SYBASE iAnywhere

MobiLink
Server Administration

Published: March 2007

Copyright and trademarks

Contents

About This Manual

Subject

This manual describes how to set up and administer MobiLink applications.

Audience

This manual is for anyone who wants to create distributed information systems. The central data source and remote data stores can be, but are not restricted to, relational database systems.

Before you begin

For a comparison of MobiLink with other SQL Anywhere synchronization and replication technologies, see "Comparing synchronization technologies" [*SQL Anywhere 10 - Introduction*].

SQL Anywhere documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere documentation

The complete SQL Anywhere documentation is available in two forms: an online form that combines all books, and as separate PDF files for each book. Both forms of the documentation contain identical information and consist of the following books:

- **SQL Anywhere 10 - Introduction** This book introduces SQL Anywhere 10—a product that provides data management and data exchange technologies, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 10 - Changes and Upgrading** This book describes new features in SQL Anywhere 10 and in previous versions of the software, as well as upgrade instructions.
- **SQL Anywhere Server - Database Administration** This book covers material related to running, managing, and configuring SQL Anywhere databases. It describes database connections, the database server, database files, backup procedures, security, high availability, and replication with Replication Server, as well as administration utilities and options.
- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **SQL Anywhere Server - SQL Reference** This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.
- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by these tools.
- **SQL Anywhere 10 - Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.
- **MobiLink - Getting Started** This manual introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Server Administration** This manual describes how to set up and administer MobiLink server-side utilities and functionality.
- **MobiLink - Client Administration** This manual describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases.
- **MobiLink - Server-Initiated Synchronization** This manual describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.

- **QAnywhere** This manual describes QAnywhere, which is a messaging platform for mobile and wireless clients as well as traditional desktop and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **SQL Anywhere 10 - Context-Sensitive Help** This manual contains the context-sensitive help for the Connect dialog, the Query Editor, the MobiLink Monitor, MobiLink Model mode, the SQL Anywhere Console utility, the Index Consultant, and Interactive SQL.
- **UltraLite - Database Management and Reference** This manual introduces the UltraLite database system for small devices.
- **UltraLite - AppForge Programming** This manual describes UltraLite for AppForge. With UltraLite for AppForge you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS, Symbian OS, or Windows CE.
- **UltraLite - .NET Programming** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This manual describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows CE, or Windows XP.
- **UltraLite - C and C++ Programming** This manual describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.

Documentation formats

SQL Anywhere provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ► Programs ► SQL Anywhere 10 ► Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on Unix operating systems, see the HTML documentation under your SQL Anywhere installation or on your installation CD.

- **PDF files** The complete set of SQL Anywhere books is provided as a set of Adobe Portable Document Format (pdf) files, viewable with Adobe Reader.

On Windows, the PDF books are accessible from the online documentation via the PDF link at the top of each page, or from the Windows Start menu (Start ► Programs ► SQL Anywhere 10 ► Online Books - PDF Format).

On Unix, the PDF books are available on your installation CD.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in uppercase, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, …]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

  [**QUOTES** { **ON** | **OFF** }]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

Operating system conventions

- **Windows** The Microsoft Windows family of operating systems for desktop and laptop computers. The Windows family includes Windows Vista and Windows XP.

- **Windows CE** Platforms built from the Microsoft Windows CE modular operating system, including the Windows Mobile and Windows Embedded CE platforms.

  Windows Mobile is built on Windows CE. It provides a Windows user interface and additional functionality, such as small versions of applications like Word and Excel. Windows Mobile is most commonly seen on mobile devices.

  Limitations or variations in SQL Anywhere are commonly based on the underlying operating system (Windows CE), and seldom on the particular variant used (Windows Mobile).

- **Unix** Unless specified, Unix refers to both Linux and Unix platforms.

File name conventions

The documentation generally adopts Windows conventions when describing operating system dependent tasks and features such as paths and file names. In most cases, there is a simple transformation to the syntax used on other operating systems.

- **Directories and path names** The documentation typically lists directory paths using Windows conventions, including colons for drives and backslashes as a directory separator. For example,

  ```
  MobiLink\redirector
  ```

  On Unix, Linux, and Mac OS X, you should use forward slashes instead. For example,

  ```
  MobiLink/redirector
  ```

  If SQL Anywhere is used in a multi-platform environment you must be aware of path name differences between platforms.

- **Executable files** The documentation shows executable file names using Windows conventions, with the suffix *.exe*. On Unix, Linux, and Mac OS X, executable file names have no suffix. On NetWare, executable file names use the suffix *.nlm*.

  For example, on Windows, the network database server is *dbsrv10.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv10*. On NetWare, it is *dbsrv10.nlm*.

- **install-dir** The installation process allows you to choose where to install SQL Anywhere, and the documentation refers to this location using the convention *install-dir*.

  After installation is complete, the environment variable SQLANY10 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*). SQLANYSH10 specifies the location of the directory containing components shared by SQL Anywhere with other Sybase applications.

  For more information on the default location of *install-dir*, by operating system, see "SQLANY10 environment variable" [*SQL Anywhere Server - Database Administration*].

- **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

After installation is complete, the environment variable SQLANYSAMP10 specifies the location of the directory containing the samples (*samples-dir*). From the Windows Start menu, choosing Programs ► SQL Anywhere 10 ► Sample Applications and Projects opens a Windows Explorer window in this directory.

For more information on the default location of *samples-dir*, by operating system, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%envvar%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$envvar* or *${envvar}*.

  Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

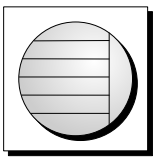
Graphic icons

The following icons are used in this documentation.

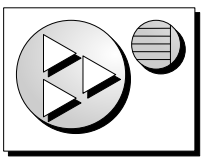
- A client application.

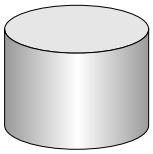
- A database server, such as SQL Anywhere.

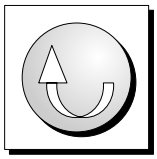
- An UltraLite application.

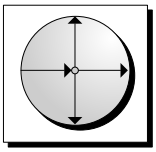
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.



- A Sybase Replication Server



- A programming interface.

Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at http://www.ianywhere.com/developer/.

If you have questions or need help, you can post messages to the Sybase iAnywhere newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere. You can find this information by entering **dbeng10 -v** at a command prompt.

The newsgroups are located on the *forums.sybase.com* news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

Newsgroup disclaimer
iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.
iAnywhere Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

Feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can email comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

Part I. Using MobiLink Technology

This part introduces MobiLink technology and describes how to use it to synchronize data between two or more data sources.

CHAPTER 1

MobiLink Consolidated Databases

Contents

Introduction to consolidated databases

Your consolidated database holds system objects that are required by MobiLink. In most cases it also holds your application data, but you can hold all or part of your application data in other forms as well.

Your consolidated database can be one of the following ODBC-compliant RDBMSs:

- SQL Anywhere
- Adaptive Server Enterprise
- Oracle
- Microsoft SQL Server
- IBM DB2 UDB

For version support information, see the MobiLink table in SQL Anywhere 10.0.1 Components by Platform.

Your SQL Anywhere installation includes a setup script for each type of RDBMS. You need to run the appropriate setup script to use that RDBMS with MobiLink. The setup script adds tables and stored procedures that are required by MobiLink.

For information about setting up each type of database as a consolidated database, see "Setting up a consolidated database" on page 6.

For information about writing synchronization scripts for particular consolidated databases, see "RDBMS-dependent synchronization scripts" on page 7.

Synchronizing to other data sources

Your MobiLink environment must have a database that has been set up as a consolidated database. However, you can synchronize data sources other than the consolidated database. You can create a hybrid application in which you synchronize to both a consolidated database and some other data source, or you can synchronize to just a consolidated database or just another data source.

See "Direct Row Handling" on page 517.

How remote tables relate to consolidated tables

Synchronization designs can specify mappings between tables and rows in the remote databases with tables and rows in the consolidated database. Typically, tables and columns in remote databases either exactly match the tables and columns in the consolidated database or are subsets of them.

Arbitrary relationships permitted

Tables in a remote database need not be identical to those in the consolidated database. Synchronized data in one remote application table can be distributed between columns in different tables, and even between tables in different consolidated databases. You specify these relationships using synchronization scripts.

Direct relationships are simple

The simplest and most common design uses a table structure in the remote database that is a subset of that in the consolidated database. Using this design, every table in the remote database exists in the consolidated

database. Corresponding tables have the same structure and foreign key relationships as those in the consolidated database.

The consolidated database frequently contains columns and tables that are not synchronized. Some of these columns or tables may be used for synchronization. For example, a timestamp column can identify new or updated rows in the consolidated database; or a shadow table can be used to track deletes. Non-synchronized columns or tables in the consolidated database can also hold information that is not required at remote sites.

Remote databases also frequently hold tables or columns that aren't synchronized.

See also

- "MobiLink Data Mappings Between Remote and Consolidated Databases" on page 585

Setting up a consolidated database

Setup scripts

To set up a database so that it can be used as a MobiLink consolidated database, you must run a setup script. Your SQL Anywhere installation includes a script for each of the supported RDBMSs. These scripts are all located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.

Note:
If you use the Create Synchronization Model wizard to create your MobiLink application, the wizard will check whether your database needs to be set up, and will automatically run the appropriate setup script. See "Introduction to MobiLink models" [*MobiLink - Getting Started*].

The MobiLink setup script adds MobiLink system tables, stored procedures, triggers, and views to your database. These tables and procedures are required for MobiLink synchronization.

For information about the MobiLink system tables that are installed, see "MobiLink Server System Tables" on page 551.

For information about the stored procedures that are installed, see "MobiLink system procedures" on page 532.

You can view each setup script in a text editor if you want to check what it does.

Caution
The database user who runs the setup scripts is given permission to update the MobiLink system tables, which is required to start the MobiLink server and to configure MobiLink. See "Required permissions" on page 20.

For instructions on how to run the setup scripts, see the section for your RDBMS:

- "SQL Anywhere consolidated database" on page 9
- "Sybase Adaptive Server Enterprise consolidated database" on page 10
- "Oracle consolidated database" on page 12
- "IBM DB2 UDB consolidated database" on page 14
- "Microsoft SQL Server consolidated database" on page 17

ODBC connection

The MobiLink server needs an ODBC connection to your consolidated database. You must configure the appropriate ODBC driver for your server and create an ODBC data source for the database on the computer where your MobiLink server is running.

For more information about MobiLink ODBC drivers, see "iAnywhere Solutions ODBC Drivers for MobiLink" on page 619.

For updated information and complete functional specifications of the ODBC drivers you can use with MobiLink, see http://www.ianywhere.com/developer/technotes/odbc_mobilink.html.

RDBMS-dependent synchronization scripts

MobiLink uses synchronization scripts to define the rules you use to synchronize data. Synchronization scripts define:

- How data uploaded from the remote database is to be applied to the consolidated database.
- What data should be downloaded from the consolidated database to the remote database.

See "Writing Synchronization Scripts" on page 213.

For a complete list of events you can write scripts for, see "Synchronization Events" on page 239.

For specific information about each type of consolidated database, see:

- "SQL Anywhere consolidated database" on page 9
- "Sybase Adaptive Server Enterprise consolidated database" on page 10
- "Oracle consolidated database" on page 12
- "IBM DB2 UDB consolidated database" on page 14
- "Microsoft SQL Server consolidated database" on page 17

.NET and Java synchronization scripts

You can write your synchronization logic in the version of the SQL language used by your database. You can also write more portable and powerful scripts using Java or .NET. Both Java and .NET offer flexibility beyond what each RDBMS provides via SQL, while also providing full SQL compatibility. When you use Java or .NET synchronization logic, you can hold session-wide variables, create user-defined procedures, integrate authentication to external servers, and so on.

For information about Java synchronization logic, see "Writing Java synchronization logic" on page 419.

For information about .NET synchronization logic, see "Writing Synchronization Scripts in .NET" on page 461.

Invoking procedures from scripts

Some databases, such as Microsoft SQL Server, require that procedure calls with parameters be written using the ODBC syntax.

```
{ CALL procedure_name( {ml param1}, {ml param2}, ... ) }
```

You can return values by defining the parameters as OUT or INOUT in the procedure definition.

CHAR columns

In many other RDBMSs, CHAR data types are fixed length and blank-padded to the full length of the string. In SQL Anywhere or UltraLite remote MobiLink databases, CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. If you are not using SQL Anywhere as your consolidated database, It is strongly recommended that you use VARCHAR in the consolidated database rather than CHAR. If you must

use CHAR, the mlsrv10 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

See "-b option" on page 35.

Data conversion

For information about the conversion of data that must take place when a MobiLink server communicates with a consolidated database that isn't SQL Anywhere, see "MobiLink Data Mappings Between Remote and Consolidated Databases" on page 585.

SQL Anywhere consolidated database

To set up SQL Anywhere to work as a MobiLink consolidated database, you must run a setup procedure that adds MobiLink system tables, stored procedures, triggers, and views that are required for MobiLink synchronization. There are multiple ways you can do this:

- Run the *syncsa.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.
- In the MobiLink plug-in in Sybase Central, choose Mode ► Admin and connect to your server database. Right-click the database name and choose Check MobiLink System Setup. If your database requires setup, you are prompted to continue.
- When you use the Create Synchronization Model wizard or Deploy Synchronization Model wizard, system setup is checked when you connect to your server database. If your database requires setup, you are prompted to continue.

The database user who runs the setup script is the only user who will have permission to change the MobiLink system tables, which is required for configuring MobiLink applications.

Setting up the ODBC driver

You must set up an ODBC DSN for your SQL Anywhere consolidated database. The ODBC driver for SQL Anywhere is installed with SQL Anywhere.

For information about the SQL Anywhere ODBC driver, see "Working with ODBC data sources" [*SQL Anywhere Server - Database Administration*].

Isolation level

See "MobiLink isolation levels" on page 131.

Sybase Adaptive Server Enterprise consolidated database

To set up Adaptive Server Enterprise to work as a MobiLink consolidated database, you must run a setup procedure that adds MobiLink system tables, stored procedures, triggers, and views that are required for MobiLink synchronization. There are multiple ways you can do this:

- Run the *syncase.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.
- In the MobiLink plug-in in Sybase Central, choose Mode ► Admin and connect to your server database. Right-click the database name and choose Check MobiLink System Setup. If your database requires setup, you are prompted to continue.
- When you use the Create Synchronization Model wizard or Deploy Synchronization Model wizard, system setup is checked when you connect to your server database. If your database requires setup, you are prompted to continue.

The database user who runs the setup script is the only user who will have permission to change the MobiLink system tables, which is required for configuring MobiLink applications.

ODBC driver

You must set up an ODBC DSN for your Adaptive Server Enterprise consolidated database using the ODBC driver that is provided with your Adaptive Server Enterprise database. See:

- http://www.ianywhere.com/developer/technotes/odbc_mobilink.html
- Your Adaptive Server Enterprise documentation

Adaptive Server Enterprise issues

- **Column sizes** To download BLOB data with a data size greater than 1024 KB (the default), you need to set the ODBC DSN setting Default Buffer Size for Long Columns to be greater than the largest expected BLOB.
- **CHAR columns** In Adaptive Server Enterprise, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (SQL Anywhere or UltraLite), CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. It is strongly recommended that you use VARCHAR in the consolidated database rather than CHAR. If you must use CHAR, the mlsrv10 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

  For more information, see "-b option" on page 35.
- **Data type mapping** The data types of columns must map correctly between your consolidated and remote database. See "Adaptive Server Enterprise data mapping" on page 586.

- **Special considerations for version 11.5 and earlier** You cannot use MobiLink system procedures such as ml_add_connection_script to add scripts longer than 255 bytes to Adaptive Server Enterprise 11.5 or earlier. To define longer scripts, use Sybase Central or direct insertion.
- **Restrictions on VARBIT** MobiLink does not support synchronizing 0 length (empty) VARBIT or LONG VARBIT values to an Adaptive Server Enterprise consolidated database. Adaptive Server Enterprise does not support a VARBIT type so these types would normally be synchronized to a VARCHAR or TEXT column on the Adaptive Server Enterprise database. On Adaptive Server Enterprise, empty string values are converted into a single space. A space is not allowed in a VARBIT column on SQL Anywhere, so an attempt to download these values causes an error on the remote database.

Isolation level

See "MobiLink isolation levels" on page 131.

Oracle consolidated database

To set up Oracle to work as a MobiLink consolidated database, you must run a setup procedure that adds MobiLink system tables, stored procedures, triggers, and views that are required for MobiLink synchronization. There are multiple ways you can do this:

- Run the *syncora.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.
- In the MobiLink plug-in in Sybase Central, choose Mode ► Admin and connect to your server database. Right-click the database name and choose Check MobiLink System Setup. If your database requires setup, you are prompted to continue.
- When you use the Create Synchronization Model wizard or Deploy Synchronization Model wizard, system setup is checked when you connect to your server database. If your database requires setup, you are prompted to continue.

The database user who runs the setup script is the only user who will have permission to change the MobiLink system tables, which is required for configuring MobiLink applications.

ODBC driver

You must set up an ODBC DSN for your Oracle consolidated database. See:

- "iAnywhere Solutions Oracle driver" on page 621
- http://www.ianywhere.com/developer/technotes/odbc_mobilink.html

Oracle issues

- **Session-wide variables** Oracle does not provide session-wide variables. You can store session-wide information in variables within Oracle packages. Oracle packages allow variables to be created, modified and destroyed; these variables may last as long as the Oracle package is current.
- **Autoincrement methods** To maintain primary key uniqueness, you can use an Oracle sequence to generate a list of keys similar to that of an autoincrement field. The CustDB sample database provides coding examples, which can be found in *Samples\MobiLink\CustDB\custora.sql*. Unlike autoincrement, however, you must explicitly reference the sequence. Autoincrement inserts a column value automatically if the column is not referenced in an INSERT statement.

  For an example of using an Oracle sequence, see "Tutorial: Using MobiLink with an Oracle 10g Consolidated Database" [*MobiLink - Getting Started*].
- **Oracle does not support empty strings** In Oracle, an empty string is treated as NULL. In SQL Anywhere and UltraLite, empty strings have a different meaning from NULL. Therefore, you should avoid using empty strings in your client databases when you have an Oracle consolidated database.
- **CHAR columns** In Oracle, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (SQL Anywhere or UltraLite), CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. It is strongly recommended that you use

VARCHAR in the consolidated database rather than CHAR. If you must use CHAR, the mlsrv10 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

See "-b option" on page 35.

- **Data type mapping** The data types of columns must map correctly between your consolidated and remote database. For details, see "Oracle data mapping" on page 600.

Isolation level

See "MobiLink isolation levels" on page 131.

IBM DB2 UDB consolidated database

MobiLink supports IBM DB2 UDB for Linux, Unix, and Windows. MobiLink does not support IBM DB2 for AS/400 or mainframe.

To set up DB2 to work as a MobiLink consolidated database, you must run a setup procedure that adds MobiLink system tables, stored procedures, triggers, and views that are required for MobiLink synchronization. There are multiple ways you can do this:

- Run the *syncdb2long.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation. Before running the file, you must copy it to another location and modify it. Instructions follow.
- In the MobiLink plug-in in Sybase Central, choose Mode ► Admin and connect to your server database. Right-click the database name and choose Check MobiLink System Setup. If your database requires setup, you are prompted to continue.
- When you use the Create Synchronization Model wizard or Deploy Synchronization Model wizard, system setup is checked when you connect to your server database. If your database requires setup, you are prompted to continue.

The database user who runs the setup script is the only user who will have permission to change the MobiLink system tables, which is required for configuring MobiLink applications.

♦ To run the DB2 setup script

1. To install MobiLink system tables using the setup script, an IBM DB2 UDB tablespace must use a minimum of 8 KB pages. If a tablespace does not use 8 KB pages, complete the following steps:

   - Verify that at least one of your buffer pools has 8 KB pages. If not, create a buffer pool with 8 KB pages.
   - Create a new tablespace and temporary tablespace that use the buffer pool with 8 KB pages.

   For more information, consult your DB2 UDB documentation.

2. Customize *syncdb2long.sql* with your connection information:

   a. Copy *syncdb2long.sql* to a new location where it can be modified and stored.

   b. The *syncdb2long.sql* script contains a default connection statement, `connect to DB2Database`. Alter this line to connect to your DB2 database. Use the following syntax:

   ```
   connect to DB2Database user userid using password ~
   ```

   where *DB2Database*, *userid*, and *password* are names you provide. (The *syncdb2long.sql* script uses the tilde character (~) as a command delimiter.)

3. Run *syncdb2long.sql*:


```
db2 -c -ec -td~ +s -v -f syncdb2long.sql
```

4. Copy *SyncDB2Long_1000.class* and *SyncDB2Long_1000.java*, which are located in the *MobiLink \setup* subdirectory of your SQL Anywhere installation, to the *FUNCTION* subdirectory of your DB2 UDB installation.

ODBC driver

You must set up an ODBC DSN for your DB2 consolidated database using the ODBC driver that is provided with your DB2 database. See:

- http://www.ianywhere.com/developer/technotes/odbc_mobilink.html
- IBM DB2 UDB documentation

DB2 UDB issues

- **Tablespace capacity** A tablespace and temporary tablespace of any DB2 UDB database that you want to use as a consolidated database must use 8 KB pages.

  In addition, there are columns that require a LONG tablespace. If there is no default LONG tablespace, the creation statements for the tables containing these columns must be qualified appropriately, as in the following example:

  ```
  CREATE TABLE ... ( ... )
  IN tablespace
  LONG IN long-tablespace
  ```

  For an example using the sample application, see "Exploring the CustDB Sample for MobiLink" [*MobiLink - Getting Started*].

- **Session-wide variables** DB2 UDB prior to version 8 does not support session-wide variables. A convenient solution is to use a base table with columns for the MobiLink user name and other session data. The base table has rows representing concurrent synchronizations.

- **User-defined procedures** DB2 UDB prior to version 8.2 requires that you compile SQL procedures into an executable library (such as a DLL). The resulting DLL/shared library must be copied to a special directory on the server. Note that you can write procedures using several different languages, including C/C++ and Java, among others.

  For an example of Java as a procedural language for DB2 UDB, see the CustDB scripts in the files *Samples\MobiLink\CustDB\custdbq.sql* and *Samples\MobiLink\CustDB\custdbq.java*.

  For more information about Java and .NET synchronization scripts, see:

  - "Writing Synchronization Scripts in Java" on page 415
  - "Writing Synchronization Scripts in .NET" on page 461

- **CHAR columns** In IBM DB2 UDB, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (SQL Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. It is strongly recommended that you use VARCHAR in the consolidated database rather than CHAR. If you must use CHAR, the mlsrv10 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

See "-b option" on page 35.

- **Data type mapping** The data types of columns must map correctly between your consolidated and remote database. For details, see "IBM DB2 UDB data mapping" on page 593.
- **Double up the quotation marks in system procedure calls** When you use a MobiLink system procedure to add scripts to your DB2 consolidated database, you need to double up the quotation marks. For example, if the script you are adding with ml_add_table_script includes the line `SET "DELETED"=''Y''` for any other consolidated database, for DB2 you would have to write this as `SET "DELETED" = ''''Y''''`.
- **Special considerations for version 5 and earlier** If you are using IBM DB2 UDB prior to version 6, column names and other identifiers are only supported up to 18 characters. This means that you must truncate the names of MobiLink system procedures. For example, to call ml_add_connection_script, use the name ml_add_connection_.

Isolation level

See "MobiLink isolation levels" on page 131.

Microsoft SQL Server consolidated database

To set up Microsoft SQL Server to work as a MobiLink consolidated database, you must run a setup procedure that adds MobiLink system tables, stored procedures, triggers, and views that are required for MobiLink synchronization. There are multiple ways you can do this:

- Run the *syncmss.sql* setup script, located in the *MobiLink\setup* subdirectory of your SQL Anywhere installation.
- In the MobiLink plug-in in Sybase Central, choose Mode ► Admin; connect to your server database; right-click the database name and choose Check MobiLink System Setup. If your database requires setup, you are prompted to continue.
- When you use the Create Synchronization Model wizard or Deploy Synchronization Model wizard, system setup is checked when you connect to your server database. If your database requires setup, you are prompted to continue.

The database user who runs the setup script is the only user who will have permission to change the MobiLink system tables, which is required for configuring MobiLink applications.

ODBC driver

You must set up an ODBC DSN for your SQL Server consolidated database using the ODBC driver that is provided with your SQL Server database. See:

- http://www.ianywhere.com/developer/technotes/odbc_mobilink.html
- Microsoft SQL Server documentation

SQL Server issues

- **SET NOCOUNT ON** For Microsoft SQL Server, you should specify SET NOCOUNT ON as the first statement in all stored procedures or SQL batches executed via ODBC. Without this option, network buffers can overflow, silently losing data. This is a known SQL Server problem.
- **Procedure calls** Microsoft SQL Server requires that procedure calls with parameters be written using the ODBC syntax:

  ```
  { CALL procedure_name( {ml param1}, {ml param2}, ... ) }
  ```

- **CHAR columns** In Microsoft SQL Server, CHAR data types are fixed length and blank-padded to the full length of the string. In MobiLink remote databases (SQL Anywhere or UltraLite) CHAR is the same as VARCHAR: values are not blank-padded to a fixed width. We strongly recommend that you use VARCHAR in the consolidated database rather than CHAR. If you must use CHAR, the mlsrv10 -b command line option can be used to remove trailing blanks from strings during synchronization. This option is important for string comparisons used to detect conflicts.

  See "-b option" on page 35.

- **Data type mapping** The data types of columns must map correctly between your consolidated and remote database. For details, see "Microsoft SQL Server data mapping" on page 609.

- **Sample database issues** The SQL Server AdventureWorks sample database contains computed columns. You can't synchronize a computed column. You can set the column to be download-only, or you can exclude the column from synchronization.
- **Implementing conflict detection in an upload_update script** The behavior of the SQL Server NOCOUNT option means that sometimes the MobiLink server cannot accurately assess how many rows were changed by an upload script. For SQL Server, it is safer to implement a stored procedure in the upload_update script for conflict detection.

Isolation level

See "MobiLink isolation levels" on page 131.

CHAPTER 2

MobiLink Server

Contents

Running the MobiLink server

All MobiLink clients synchronize through the MobiLink server. None connect directly to a database server. You must start the MobiLink server before a MobiLink client synchronizes.

For a list of mlsrv10 command line options, see "MobiLink Server Options" on page 27.

The MobiLink server opens connections, via ODBC, with your consolidated database server. It then accepts connections from remote applications and controls the synchronization process.

♦ To start the MobiLink server

- Run mlsrv10. Use the –c option to specify the ODBC connection parameters for your consolidated database.

For information about connection parameters, see "-c option" on page 37.

You must specify connection parameters. Other options are available, but are optional. These options allow you to specify how the server works. For example, you can specify a cache size and logging options.

For more information about mlsrv10 options, see "mlsrv10 syntax" on page 29.

> **Note**
> The mlsrv10 options allow you to specify how the MobiLink server works. To control what the server does, you define scripts that are invoked at synchronization events. See "Synchronization Events" on page 239.

Example

The following command starts the MobiLink server using the ODBC data source *SQL Anywhere 10 CustDB* to identify the consolidated database. Enter the entire command on one line.

```
mlsrv10
  -c "dsn=SQL Anywhere 10 CustDB;uid=DBA;pwd=sql"
  -zs MyServer
  –o mlsrv.log
  -vcr
  -x tcpip
  -xo tcpip
```

In this example, the -c option provides a connection string that contains an ODBC data source name (DSN) as well as authentication. The -zs option provides a server name. The –o option specifies that the log file should be named *mlsrv.log*. The contents of *mlsrv.log* are verbose because of the –vcr option. The –x option opens a port for version 10 clients, and the -xo option opens a port for version 8 and 9 clients.

You can also start the MobiLink server as a Windows service or Unix daemon. See "Running the MobiLink server outside the current session" on page 25.

Required permissions

You must specify a database user to connect to the MobiLink server. You specify the database user with the mlsrv10 -c option or in the DSN.

This database user must have full select, insert, update, and delete permissions on the MobiLink system tables, and must also have execute permissions on the MobiLink system procedures. By default, the database user who runs the MobiLink setup script has these permissions. If you want to use another database user to run the MobiLink server, you must grant permissions for that user on the ml_* tables and the ml_add_*_script system procedures.

For example, in a SQL Anywhere consolidated database you can grant the required permissions as follows:

```
GRANT CONNECT to DBUser IDENTIFIED BY SQL;
GRANT ALL ON dbo.ml_user to DBUser;
...
GRANT EXECUTE ON dbo.ml_add_table_script TO DBUser;
...
```

You must grant permission for each MobiLink system table and system procedure. For a list of all MobiLink system tables and system procedures, see "MobiLink Server System Tables" on page 551 and "MobiLink Server System Procedures" on page 531.

The database user also needs the appropriate permission on all tables referenced in the MobiLink scripts, as well as execute permissions on any procedures referenced in the MobiLink scripts.

For more information about setting permission, see "GRANT statement" [*SQL Anywhere Server - SQL Reference*].

For more information about setup scripts, see "Setting up a consolidated database" on page 6.

Stopping the MobiLink server

The MobiLink server is stopped from the computer where the server was started. You can stop the MobiLink server in the following ways:

- Use the MobiLink stop utility (mlstop).
- Click Shutdown on the MobiLink server window.
- On Windows, right-click the icon in the system tray and choose Shut down.
- When running on Unix without the MobiLink server window, type Q.
- Use the shutdown method in the MobiLink server API.

See also

- "MobiLink stop utility [mlstop]" on page 547
- server API for Java: "shutdown method" on page 446
- server API for .NET: "ShutDown method" on page 499

Logging MobiLink server actions

Logging the actions that the server takes is particularly useful during the development process and when troubleshooting. Verbose output is not recommended for normal operation of a production environment because it can slow performance.

Logging output to a file

Logging output is sent to the MobiLink console. In addition, you can send the output to a log file using the -o option. The following command sends output to a log file named *mlsrv.log*.

```
mlsrv10 -o mlsrv.log -c ...
```

You can control the size of log files, and specify what you want done when a file reaches its maximum size.

- Use the -o option to specify that a log file should be used.
- Use the -ot option to specify that a log file should be used when you want the previous contents of the file to be deleted before messages are sent to it.
- In addition to -o or -ot, use the -on option to specify the size at which the log file is renamed with the extension .old and a new file is started with the original name.
- In addition to -o or -ot, use the -os option to specify the size at which a new log file is started with a new name based on the date and a sequential number.

See:

- “-o option” on page 56
- “-on option” on page 57
- “-os option” on page 59
- “-ot option” on page 60

Controlling the amount of logging output

You can control what information is logged to the message log file and displayed in the MobiLink server window using the –v option. See “-v option” on page 73.

Controlling which warning messages are reported

You can also control which warning messages are reported.

For more information, see:

- “-zw option” on page 92
- “-zwd option” on page 93
- “-zwe option” on page 94

Viewing MobiLink server logs

You can view MobiLink logs in the following ways:

- In the console
- By opening the log file
- Using the MobiLink Log Viewer in Sybase Central

To view log information outside of the console, you must log the information to a file. See "Logging output to a file" on page 23.

MobiLink Server Log File Viewer

To view MobiLink server logs, open Sybase Central and choose Tools ► MobiLink 10 ► MobiLink Server Log File Viewer. You are prompted to choose a log file to view. By holding down the shift key, you can open multiple log files at the same time.

The Log File Viewer reads information that is stored in MobiLink log files. It does not connect to the MobiLink server or change the composition of log files.

The Log File Viewer allows you to filter the information that you view. In addition, it provides statistics based on the information in the log.

Running the MobiLink server outside the current session

You can set up the MobiLink server to be available all the time. To make this easier, you can run the MobiLink server for Windows and for Unix so that it remains running when you log off the computer. The way you do this depends on your operating system.

- **Unix daemon** You can run the MobiLink server as a daemon using the mlsrv10 -ud option, enabling the MobiLink server to run in the background, and to continue running after you log off.
- **Windows service** You can run the Windows MobiLink server as a service.

  To stop a MobiLink server that is running as a service, you can use mlstop, dbsvc, or the Windows Service Manager.

See also

- "-ud option" on page 70
- "Service utility (dbsvc) for Linux" [*SQL Anywhere Server - Database Administration*]
- "Running the server outside the current session" [*SQL Anywhere Server - Database Administration*]

Troubleshooting MobiLink server startup

This section describes some common problems when starting the MobiLink server.

Ensure that network communication software is running

Appropriate network communication software must be installed and running before you run the MobiLink server. If you are running reliable network software with just one network installed, this should be straightforward. You should confirm that other software requiring network communications is working properly before running the MobiLink server.

If you are running under the TCP/IP protocol, you may want to confirm that ping and telnet are working properly. The ping and telnet applications are provided with many TCP/IP protocol stacks.

Debug network communications startup problems

If you are having problems establishing a connection across a network, you can use debugging options at both the client and server to diagnose problems. The startup information appears on the server window: you can use the -o option to log the results to an output file.

See "Logging MobiLink server actions" on page 23.

CHAPTER 3

MobiLink Server Options

Contents

mlsrv10 syntax

The MobiLink server lets you synchronize remote databases or applications with an ODBC-compliant consolidated database.

Function

Start a MobiLink server.

Syntax

mlsrv10 –c "*connection-string***"** [*options*]

Option	Description
@data	Read in options from the specified environment variable or configuration file. See “@data option” on page 33.
–a	Disable automatic reconnection upon synchronization error. See “-a option” on page 34.
-b	Trim blank padding of strings. See “-b option” on page 35.
–bn *size*	Specify the maximum number of bytes to consider when comparing BLOBs for conflict detection. See “-bn option” on page 36.
–c *"keyword=value; …"*	Supply ODBC database connection parameters for your consolidated database. See “-c option” on page 37.
–cm *size*	Specify the server memory cache size. See “-cm option” on page 38.
–cn *connections*	Set the maximum number of simultaneous connections with the consolidated database server. See “-cn option” on page 39.
–cr *count*	Set the maximum number of database connection retries. See “-cr option” on page 40.
–ct *connection-timeout*	Set the length of time a connection may be unused before it is timed out. See “-ct option” on page 41.
–dl	Display all log messages on the console. See “-dl option” on page 42.
-ds *size*	Specify the maximum amount of data that can be stored for use in all restartable downloads. See “-ds option” on page 43.
-dsd	Disable snapshot isolation, which is the default download isolation level for SQL Anywhere and Microsoft SQL Server consolidated databases. See “-dsd option” on page 44.

Option	Description
-dt	Detect transactions only within the current database. See "-dt option" on page 45.
–e *filename*	Store remote error logs sent into the named file. See "-e option" on page 46.
–esu	Use snapshot isolation for uploads. See "-esu option" on page 47.
–et *filename*	Store remote error logs sent into the named file, but delete the contents first if the file exists. See "-et option" on page 48.
–f	Assume synchronization scripts do not change. See "-f option" on page 49.
–fips	Forces all secure MobiLink streams to be FIPS-compliant. See "-fips option" on page 50.
–fr	If table data scripts are missing, synchronization will not abort, but just issue a warning. See "-fr option" on page 51.
–ftr *path*	Creates a location for files that are to be used by the mlfile-transfer utility. See "-ftr option" on page 52.
-m [*filename*]	Enables QAnywhere messaging. See "-m option" on page 53.
-nc *connections*	Sets maximum number of concurrent connections. See "-nc option" on page 54.
-notifier	Starts a Notifier for server-initiated synchronization. See "-notifier option" on page 55.
–o *logfile*	Log messages to a file. See "-o option" on page 56.
-on *size*	Set maximum size for log file. See "-on option" on page 57.
–oq	Prevent the popup dialog on startup error. See "-oq option" on page 58.
–os *size*	Maximum size of output file. See "-os option" on page 59.
–ot *logfile*	Log messages to a file, but delete its contents first. See "-ot option" on page 60.
–q	Minimize the MobiLink server window. See "-q option" on page 61.

Option	Description
–r *retries*	Retry deadlocked uploads at most this many times. See "-r option" on page 62.
–rd *delay*	Set maximum delay, in seconds, before retrying a deadlocked transaction. See "-rd option" on page 63.
–s *count*	Specify the maximum number of rows to be fetched or sent at once. See "-s option" on page 64.
–sl dnet *script-options*	Set the .NET CLR options and force loading of the virtual machine on startup. See "-sl dnet option" on page 65.
–sl java *script-options*	Set the Java virtual machine options and force loading of the virtual machine on startup. See "-sl java option" on page 66.
-sm *number*	Set the maximum number of synchronizations that can be actively worked on. See "-sm option" on page 68.
-tx *count*	For transactional uploads, batches groups of transactions and commits them together. See "-tx option" on page 69.
–ud	On UNIX platforms, run as a daemon. See "-ud option" on page 70.
-ui	For Linux with X window, starts the MobiLink server in console mode if a usable display isn't available. See "-ui option" on page 71.
-ux	Opens the console. See "-ux option" on page 72.
–v [*levels*]	Controls the type of messages written to the log file. See "-v option" on page 73.
–w *count*	Set the number of database worker threads. See "-w option" on page 75.
–wu *count*	Set the maximum number of database worker threads permitted to process uploads concurrently. See "-wu option" on page 76.
–x *protocol*[**(***network-parameters***)**]	Specify the communications protocol. Optionally, specify network parameters in form *parameter=value*, with multiple parameters separated by semicolons. See "-x option" on page 77.
–xo *protocol*[**(***network-parameters***)**]	For version 8 and 9 clients, specify the communications protocol. Optionally, specify network parameters in form *parameter=value*, with multiple parameters separated by semicolons. See "-xo option" on page 82.

Option	Description
–zp	In the event of a timestamp conflict between the consolidated and remote database, this option allows timestamp values with a precision higher than the lowest-precision to be used for conflict detection purposes. See "-zp option" on page 87.
–zs *name*	Specify a server name. See "-zs option" on page 88.
–zt *number*	Specify the maximum number of processors used to run the MobiLink server. See "-zt option" on page 89.
–zu { **+** | **–** }	Controls the automatic addition of users when the authenticate_user script is undefined. See "-zu option" on page 90.
-zus	Causes MobiLink to invoke upload scripts for tables for which there is no upload. See "-zus option" on page 91.
-zw 1,…5	Controls which levels of warning message to display. See "-zw option" on page 92.
-zwd *code*	Disables specific warning codes. See "-zwd option" on page 93.
-zwe *code*	Enables specific warning codes. See "-zwe option" on page 94.

Description

The MobiLink server opens connections, via ODBC, with your consolidated database server. It then accepts connections from client applications and controls the synchronization process.

You must supply connection parameters for the consolidated database using the –c option. The command line options may be specified in any order. The –c option is shown here as the first item in a command string as a convention only. It can be anywhere in a list of options, but must precede a connection string.

Unless your ODBC data source is configured to automatically start the consolidated database, the database must be running before you start the MobiLink server.

See also

- "MobiLink Server" on page 19

@data option

Reads in options from the specified environment variable or configuration file.

Syntax

mlsrv10 –c "*connection-string***" @***data* …

Remarks

Use this option to read in mlsrv10 command line options from the specified environment variable or configuration file. If both exist with the same name that is specified, the environment variable is used.

For more information about configuration files, see "Using configuration files" [*SQL Anywhere Server - Database Administration*].

If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file.

See "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].

-a option

Instructs the MobiLink server to not reconnect on synchronization error.

Syntax

mlsrv10 –c "*connection-string***" –a** …

Remarks

Should an error occur during synchronization, the MobiLink server automatically disconnects from the consolidated database, and then re-establishes the connection. Reconnecting ensures that the following synchronization starts from a known state. When this behavior is not required, you can use this option to disable it. The maintenance of state information depends on programmer requirements and may vary depending on the ways in which the programmer configures MobiLink scripting to work with the DBMS. This applies even if that database is an Oracle, SQL Anywhere database, or other supported product. Some status information may need to be re-initialized depending on the client application.

-b option

For columns of type VARCHAR, CHAR, LONG VARCHAR, or LONG CHAR, removes trailing blanks from strings during synchronization.

Syntax

mlsrv10 –c "*connection-string***" –b** …

Remarks

Note
It is recommended that you use VARCHAR in the consolidated database rather than CHAR, so that this problem does not occur.

This option helps resolve differences between the SQL Anywhere CHAR data type and the CHAR or VARCHAR data type used by the consolidated database. The SQL Anywhere CHAR data type is equivalent to VARCHAR. However, in most consolidated databases that are not SQL Anywhere, the CHAR(n) data type is blank-padded to n characters.

When -b is specified, the MobiLink server removes trailing blanks from strings for columns of type CHAR, VARCHAR, LONG CHAR, or LONG VARCHAR if the column on the remote is a string. It does this before filtering rows that were uploaded in the current synchronization. The trimmed data is then downloaded to the remote databases.

This option can also be used to detect conflict updates. For each upload update row, the MobiLink server fetches the row from the consolidated database for the given primary key, compares the row with the pre-image of the update, and then determines whether the update is a conflict update. When -b is used, MobiLink trims trailing blanks from columns of type CHAR, VARCHAR, LONG CHAR, or LONG VARCHAR before doing the comparison.

See also

- "CHAR columns" on page 7
- "NVARCHAR data type" [*SQL Anywhere Server - SQL Reference*]

Example

If the -b option is not used, a primary-key value of 'abc' uploaded from a SQL Anywhere or UltraLite remote to a CHAR(10) column in the consolidated database will become 'abc' followed by seven blank spaces. If the same row is downloaded, then it will appear on the remote as 'abc' followed by seven spaces. If the remote database is not blank-padded, then the remote will now have two rows: both 'abc' and 'abc' followed by seven spaces. There is now a duplicate row on the remote.

If the -b option is used, a primary-key value of 'abc' uploaded from a SQL Anywhere or UltraLite remote to a CHAR(10) column in the consolidated database will become 'abc' followed by seven spaces. Seven spaces still pad the value to ten characters, but if the same row is downloaded, then MobiLink server will strip the trailing spaces, and the value will appear on the remote as 'abc'. The -b option thus fixes the duplicate row problem.

-bn option

Sets the maximum number of BLOB bytes to compare during conflict detection.

Syntax

mlsrv10 –c "*connection-string***" –bn** *size* …

Remarks

When two BLOBs contain similar or identical values, the operation of comparing them for filtering or conflict detection can be expensive due to the amount of data involved. This option tells the MobiLink server to consider only the first *size* bytes of two BLOBs when making the comparison. The default is to compare the two BLOBs in their entirety.

Under some situations, limiting the maximum amount of data compared can speed synchronization substantially; however, it can also cause errors. For example, if two large BLOBs differ only in the last few bytes, the MobiLink server may consider them identical when in fact they are not.

-c option

Specifies connection parameters for the consolidated database.

Syntax

mlsrv10 –c "*connection-string***"** …

Remarks

The connection string must give the MobiLink server information sufficient to connect to the consolidated database. The connection string is required.

The connection string must specify connection parameters in the form *keyword=value*, separated by semicolons, with no spaces between parameters.

Connection parameters must be included in the ODBC data source specification if not given in the command line. Check your RDBMS and ODBC data source to determine required connection data.

For a complete list of SQL Anywhere connection parameters, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

For information about how to hide the password, see "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].

Example

```
mlsrv10 -c "dsn=odbcname;uid=DBA;pwd=sql"
```


-cm option

Sets the maximum size for the server memory cache.

Syntax

mlsrv10 –c "*connection-string***" –cm** *size*[**k** | **m** | **g**] …

Remarks

The maximum amount of memory the server will use for holding table data, network buffers, cached download data, and other structures used for synchronization. When the server has more data than can be held in this memory pool, the data is stored on disk.

The *size* is the amount of memory to reserve in bytes. Use **k**, **m**, or **g** to specify units of kilobytes, megabytes, or gigabytes, respectively. The default is **50M**.

-cn option

Sets the maximum number of simultaneous consolidated database connections.

Syntax

mlsrv10 –c "*connection-string***" –cn** *value* …

Remarks

Specifies the maximum number of simultaneous connections that the MobiLink server should make to the consolidated database. The minimum and the default value are one greater than the number of database worker threads. A warning is issued if the supplied value is too small.

A MobiLink database connection is only used for synchronizations using one script version. When the MobiLink server is using all of the database connections that it is permitted by the -cn option, if a synchronization is pending but no database connection for its script version currently exists, the MobiLink server disconnects a connection and then creates a new database connection for the pending synchronization's script version.

A value larger than the number of worker threads may speed performance, particularly if connecting to the consolidated database is slow or if multiple script versions are in use. The optimum maximum number of database connections is the number of script versions times the number of worker threads, plus one. Connections above this optimum value will not necessarily speed synchronization, and will needlessly consume resources in both the MobiLink server and the consolidated database server.

-cr option

Sets the maximum number of database connection retries.

Syntax

mlsrv10 -c "*connection-string***" -cr** *value* …

Remarks

Set the maximum number of times that the MobiLink server will attempt to connect to the database, before quitting, when a connection goes bad. The default value is three connection retries.

-ct option

Sets the length of time, in minutes, that a connection may be unused before it is timed out and disconnected by the MobiLink server.

Syntax

mlsrv10 –c "*connection-string***" –ct** *connection-timeout …*

Remarks

MobiLink database connections that go unused for a specified amount of time are freed by the server. The timeout can be set using the -ct option. A default timeout period of 60 minutes is used.

-dl option

Displays all log messages on screen.

Syntax

mlsrv10 –c "*connection-string***" -v -dl** ...

Remarks

Display all log messages in the MobiLink server window. By default, only a subset of all messages is shown in the window when a log file is being output (using -o). In circumstances with many messages, this option can degrade performance.

See also

- "-o option" on page 56

-ds option

For use with restartable downloads. Specifies the maximum amount of data that the MobiLink server can use to store all restartable downloads.

Syntax

mlsrv10 –c "*connection-string***" –ds** *size*[**k** | **m** | **g**] …

Remarks

The MobiLink server holds download data that has not been received by the client for use in a restartable download. This option limits the amount of data that the server will hold for all the synchronizations combined. If this value is too small, the server may release download data, making it impossible to restart a download.

Use **k**, **m**, or **g** to specify units of kilobytes, megabytes, or gigabytes, respectively. The default is **10M**.

See also

- “Resuming failed downloads” on page 125
- “-dc option” [*MobiLink - Client Administration*]

-dsd option

Disables snapshot isolation.

Syntax

mlsrv10 –c "*connection-string***" –dsd** …

Remarks

When the consolidated database is SQL Anywhere (version 10 or higher) or Microsoft SQL Server (2005 or higher), the default isolation level for downloads is snapshot isolation. If the consolidated database is an earlier version of these databases, the default download isolation level is read committed.

You can also change the default isolation level in a script. However, for SQL Anywhere version 10 and Microsoft SQL Server 2005 and up databases, the isolation level is set at the start of the upload and download transactions. This means that if you set the isolation level in the begin_connection script, it may be overridden in the begin_upload and begin_download scripts.

This option only applies to SQL Anywhere version 10 and Microsoft SQL Server 2005 consolidated databases.

See also

- "MobiLink isolation levels" on page 131
- "-dt option" on page 45
- "-esu option" on page 47

-dt option

For Microsoft SQL Server databases only, causes MobiLink to detect transactions only within the current database.

Syntax

mlsrv10 –c "*connection-string***" –dt** …

Remarks

If your consolidated database is running on a Microsoft SQL Server that is also running other databases, if you are using snapshot isolation for uploads or downloads, and if your upload or download scripts do not access any other databases on the server, you should specify the MobiLink server -dt option. This option makes MobiLink ignore all transactions except ones within the current database. It increases throughput and reduces duplication of rows that are downloaded.

This option only applies to Microsoft SQL Server databases using snapshot isolation.

See also

- “MobiLink isolation levels” on page 131
- “-dsd option” on page 44
- “-esu option” on page 47

-e option

Stores error logs sent from SQL Anywhere MobiLink clients.

Syntax

mlsrv10 –c "*connection-string***" –e** *filename …*

Remarks

With no -e option, error logs from SQL Anywhere MobiLink clients are stored in a file named *mlsrv10.mle*. The -e option instructs the MobiLink server to store the error logs in the named file. By default, dbmlsync sends, on the occurrence of an error on the remote site, up to 32 kilobytes of remote log messages to a MobiLink server.

This option provides centralized access to remote error logs to help diagnose synchronization issues.

The amount of information delivered from a remote site can be controlled by the dbmlsync extended option ErrorLogSendLimit.

See also

- "-et option" on page 48
- "ErrorLogSendLimit (el) extended option" [*MobiLink - Client Administration*]

-esu option

Use snapshot isolation for uploads.

Synopsis

mlsrv10 –c "*connection-string***" –esu** …

Remarks

By default, MobiLink uses the SQL_TXN_READ_COMMITTED isolation level for uploads. In most cases, this is the optimal isolation level.

If you use snapshot isolation for uploads, you may generate conflicts on snapshot transactions during upload updates. If this happens, the MobiLink server rolls back the entire upload and retries it. In this case, you might want to adjust your settings for the MobiLink server options -r or -rd to specify the delay time between retries and the maximum number of retries.

You can change the default isolation level in a script. To change the upload isolation level, you would typically use the begin_upload script.

This option only applies to SQL Anywhere version 10 and Microsoft SQL Server 2005 consolidated databases.

See also

- “MobiLink isolation levels” on page 131
- “-dsd option” on page 44
- “-dt option” on page 45
- “-r option” on page 62
- “-rd option” on page 63

-et option

Stores error logs sent from SQL Anywhere MobiLink clients in the named file after truncating the existing file.

Syntax

mlsrv10 –c "*connection-string***" –et** *filename …*

Remarks

The -et option is the same as the -e option, except that the error log file is truncated before any new errors are added to it.

The amount of information delivered from a remote site can be controlled by the dbmlsync extended option ErrorLogSendLimit.

See also

- "ErrorLogSendLimit (el) extended option" [*MobiLink - Client Administration*]
- "-e option" on page 46

-f option

Loads synchronization scripts only once, for better performance.

Syntax

mlsrv10 –c "*connection-string***" –f** …

Remarks

Without the -f option, the MobiLink server checks to see if synchronization scripts have changed during regular operation. This checking is helpful during development, but can have an unnecessary performance impact in a production environment. With the -f option, the MobiLink server loads the synchronization scripts only once per MobiLink session..

-fips option

Forces all secure MobiLink streams to be FIPS-compliant.

Syntax

mlsrv10 –c *connection-string*" **-fips** …

Remarks

Specifying this option forces all MobiLink encryption to use FIPS-approved algorithms. You can still use unencrypted connections when the -fips option is specified, but you can't use simple encryption.

When you use this option, FIPS-approved algorithms are used for connections regardless of whether you specify them or not. For example, if you start the MobiLink server with the option -fips and the option -x tls(...;fips=no;...), the fips=no setting is ignored and the server starts with fips=yes.

Separately licensed component required
ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.
See "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

For MobiLink transport-layer security, the -fips option causes the server to use the FIPS-approved RSA encryption cipher, even if RSA without FIPS is specified. If ECC is specified, an error occurs because a FIPS-approved elliptic-curve algorithm is not available.

See also

- "Encrypting MobiLink client/server communications" [*SQL Anywhere Server - Database Administration*]
- "FIPS-approved encryption technology" [*SQL Anywhere Server - Database Administration*]

-fr option

If required table data scripts are missing, synchronization will not abort, but just issue a warning.

Syntax

mlsrv10 –c "*connection-string***" –fr** …

Remarks

By default, the MobiLink server aborts if required synchronization scripts are missing. This option causes MobiLink to issue a warning instead of aborting.

See also

- “Required scripts” on page 227
- “Upload-only and download-only synchronizations” on page 105
- “Synchronization Events” on page 239
- “Direct Row Handling” on page 517

-ftr option

Creates a location for files that are to be used by the mlfiletransfer utility.

Syntax

mlsrv10 –c "*connection-string***" –ftr** *path …*

Remarks

This option creates a file transfer root directory. Subdirectories are automatically created for every registered MobiLink user. Files that are to be transferred to users can be placed in the root directory or in subdirectories.

This option is required if you want to use the mlfiletransfer utility.

See also

- "MobiLink file transfer utility [mlfiletransfer]" [*MobiLink - Client Administration*]
- "authenticate_file_transfer connection event" on page 251

-m option

Enables QAnywhere messaging.

Syntax

mlsrv10 –c "*connection-string***" –m** [*message-properties-file*] …

Remarks

The optional *message-properties-file* is deprecated. Properties are now specified via Sybase Central and stored in the database, or are specified with server management requests.

In the *message-properties-file*, each property must appear on its own line and consist of a property name, the = character, and then a property value.

For a list of properties you can set, see “Server properties” [*QAnywhere*].

See also

- “Introducing QAnywhere” [*QAnywhere*]
- “Starting the QAnywhere server” [*QAnywhere*]
- “Server management requests” [*QAnywhere*]

-nc option

Sets the maximum number of concurrent network connections.

Syntax

mlsrv10 –c "*connection-string***" –nc** *connections* …

Remarks

The MobiLink server rejects new synchronization connections when the limit is reached.

The default is 1024.

-notifier option

Starts the Notifier for server-initiated synchronization.

Syntax

mlsrv10 –c "*connection-string***" –notifier** [*notifier-properties-file*] …

Remarks

If you specify a Notifier configuration file name, or if you do not specify a file name but you have a default Notifier properties file called *config.notifier*, the Notifier is configured using that file. This overrides any configuration information that is stored in the ml_properties table in the consolidated database.

Otherwise, MobiLink uses the configuration information that is stored in the ml_properties table in the consolidated database.

When you use the -notifier option, you start every Notifier that you have enabled.

For more information about enabling Notifiers, see “enable property” [*MobiLink - Server-Initiated Synchronization*].

See also

- “Setting properties in more than one place” [*MobiLink - Server-Initiated Synchronization*]
- “Notifier properties file” [*MobiLink - Server-Initiated Synchronization*]
- “Notifiers” [*MobiLink - Server-Initiated Synchronization*]
- “MobiLink Notification Properties” [*MobiLink - Server-Initiated Synchronization*]

-o option

Logs output messages to a MobiLink server message log file, and limits the data logged to the console window.

Syntax

mlsrv10 –c "*connection-string***" –o** *logfile* …

Remarks

Write all log messages to the specified file. Note that the MobiLink server window, if present, usually shows a subset of all messages logged.

The MobiLink server gives the full error context in its output file if errors occur during synchronization. The error context may include the following information:

- **User Name** This is the actual user name that is provided by MobiLink SQL Anywhere applications during synchronization.
- **Modified User Name** This is the user name as modified by the modify_user script.
- **Transaction** This lists the transaction the error occurs in. The transaction could be authenticate_user, begin_synchronization, upload, prepare_for_download, download, or end_synchronization.
- **Table Name** This shows the table name if it is available or NULL.
- **Row Operation** The operation could be INSERT, UPDATE, DELETE or FETCH.
- **Row Data** This shows all the column values of the row that caused the error.
- **Script Version** This is the script version currently used for synchronization.
- **Script** This is the script that caused the error.

Error context information appears in the log regardless of your chosen level of verbosity.

See also

- "-os option" on page 59
- "-dl option" on page 42
- "-ot option" on page 60
- "-on option" on page 57
- "-v option" on page 73

-on option

Specifies a maximum size for the MobiLink server message log file, after which the file is renamed with the extension .old and a new file is started.

Syntax

mlsrv10 –c "*connection-string***" –on** *size* [**k** | **m**]…

Remarks

The *size* is the maximum file size for the output log, in bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. The minimum size limit is 10 KB.

When the log file reaches the specified size, the MobiLink server renames the output file with the extension .old, and starts a new one with the original name.

> **Note**
> If the .old file already exists, it is overwritten. To avoid losing old log files, use the -os option instead.

This option cannot be used with the -os option.

See also

- “-o option” on page 56
- “-ot option” on page 60
- “-on option” on page 57
- “-os option” on page 59
- “-v option” on page 73

-oq option

On Windows, prevents the appearance of the error dialog when a startup error occurs.

Syntax

mlsrv10 –c "*connection-string***" –oq** …

Remarks

By default, the MobiLink server displays a message box dialog if a startup error occurs. The -oq option prevents this dialog from being displayed.

-os option

Sets the maximum size of the MobiLink server message log file, after which a new log file with a new name is created and used.

Syntax

mlsrv10 –c "*connection-string***" –os** *size* [**k** | **m**] …

Remarks

The *size* is the maximum file size for logging output messages. The default unit is bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. The minimum size limit is 10 KB.

Before the MobiLink server logs output messages to a file, it checks the current file size. If the log message will make the file size exceed the specified size, the MobiLink server renames the message log file to *yymmddxx.mls*, where *xx* is a number from 00 to 99, and *yymmdd* represents the current year, month, and day.

You can use this option to prune old message log files to free up disk space. The latest output is always appended to the file specified by -o or -ot.

You cannot use this option with the -on option.

Note
This option will make an unlimited number of log files. To avoid this situation, use -o or -on.

See also

- “-o option” on page 56
- “-on option” on page 57
- “-ot option” on page 60
- “-v option” on page 73

-ot option

Logs output messages to the MobiLink server message log file, but deletes the contents first.

Syntax

mlsrv10 –c "*connection-string***" –ot** *logfilename …*

Remarks

The default is to send output to the screen.

See also

- “-on option” on page 57
- “-os option” on page 59
- “-v option” on page 73
- “-o option” on page 56

-q option

Instructs MobiLink to run in a minimized window on startup.

Syntax

mlsrv10 –c "*connection-string***" –q** …

Remarks

Minimize the MobiLink server window.

-r option

Sets the maximum number of deadlock retries.

Syntax

mlsrv10 –c "*connection-string***" –r** *retries …*

Remarks

By default, MobiLink server retries uploads that are deadlocked, for a maximum of 10 attempts. If the deadlock is not broken, synchronization fails, since there is no guarantee that the deadlock can be overcome. This option allows an arbitrary retry limit to be set. To stop the server from retrying deadlocked transactions, specify **–r 0**. The upper bound on this setting is 2 to the power 32, minus one.

-rd option

Sets the maximum delay time between deadlock retries.

Syntax

mlsrv10 –c "*connection-string***" –rd** *delay* …

Remarks

When upload transactions are deadlocked, the MobiLink server waits a random length of time before retrying the transaction. The random nature of the delay increases the likelihood that future attempts will succeed. This option allows you to specify the maximum delay in units of seconds. The value 0 (zero) makes retries instantaneous, but larger values are recommended because they yield more successful retries. The default and maximum delay value is **30**.

-s option

Sets the maximum number of rows that can be uploaded at the same time.

Syntax

mlsrv10 –c "*connection-string***" –s** *count* …

Remarks

Set the maximum number of rows that can be inserted, updated, or deleted at the same time to *count*.

The MobiLink server sends upload rows to the consolidated database through the ODBC driver. This option controls the number of rows sent to the database server in each batch. Increasing this value can speed up processing of the upload stream and reduce network time. However, with a higher setting the MobiLink server may require more resources for applying the upload stream.

The number of rows uploaded at once can be viewed in the log file as **rowset size**.

The default is 10.

-sl dnet option

Sets the .NET Common Language Runtime (CLR) options and forces the CLR to load on startup.

Syntax

mlsrv10 –c "*connection-string***" –sl dnet** *options …*

Remarks

Sets options to pass directly to the .NET CLR. The options are:

Option	Description
-D*name=value*	Set an environment variable. For example, `-Dsynchtype=far -Dextra_rows=yes` For more information, see the .NET framework class System.Environment.
-MLAutoLoadPath=*path*	Set the location of base assemblies. Only works with private assemblies. To tell MobiLink where assemblies are located, use this option or -MLDomConfigFile, but not both. When you use -MLAutoLoadPath, you cannot specify a domain in the event script. The default is the current directory.
-MLDomConfigFile=*file*	Set the location of base assemblies. Use when you have shared assemblies, or you don't want to load all assemblies in the directory, or you can't use MLAutoLoadPath for some other reason. To tell MobiLink where assemblies are located, use -MLDomConfigFile or -MLAutoLoadPath, but not both.
-MLStartClasses= *classnames*	At server startup, load and instantiate user-defined start classes in the order listed.
-clrConGC	Enable concurrent garbage collection in the CLR.
-**clrFlavor=**(**wks** | **svr**)	Flavor of the .NET CLR to load. The flavor is **svr** for server and **wks** for workstation. By default, **wks** is loaded.
-clrVersion=*version*	Version of the .NET CLR to load. This must be prefixed with **v**. For example, **v1.0.3705** loads the directory *\Microsoft.NET \Framework\v1.0.3705*.

To display this list of options, use the following command:

```
mlsrv10 -sl dnet (?)
```

See also

- "Writing Synchronization Scripts in .NET" on page 461

-sl java option

Sets the Java virtual machine options and forces the virtual machine to load on startup.

Syntax

mlsrv10 –c "*connection-string***" –sl java (** *options* **)** …

Remarks

Sets -jrepath and other options to pass directly to the Java virtual machine. The options are:

Option	Description
-hotspot | **-server** | **-classic**	Override the default choice for the Java VM to use.
–cp *location*;…	Specify a set of directories or jar files in which to search for classes. Instead of -cp, you can also use -classpath.
–D*name*=*value*	Set a system property. For example, `-Dsynchtype=far -Dextra_rows=yes`
–DMLStartClasses=*classname*, …	At server startup, load and instantiate user-defined start classes in the order listed.
–jrepath *path*	Override the default JRE path, which is the *install-dir\Sun\jre150* directory.
–verbose [**:class** |**:gc** | **:jni**]	Enable verbose output.
–X *vm-option*	Set a VM-specific option as described in the file *install-dir\Sun\jre150\bin\client\Xusage.txt*.

To display a list of Java options you can use, type:

```
java
```

Unix notes

Options must be enclosed in brackets. These can be round brackets, as shown in the syntax above, or curly braces { }.

The -jrepath option is only available on Windows. On UNIX, if you want to load a specific JRE, you should set the LD_LIBRARY_PATH (LIBPATH on AIX, SHLIB_PATH on HP-UX) to include the directory containing the JRE. The directory must be listed before any of the SQL Anywhere installation directories.

On Unix, the -cp options must be separated with colons.

See also

- "Writing Synchronization Scripts in Java" on page 415

Examples

For example, on Windows the following partial mlsrv10 command line sets the Java virtual machine option that enables system asserts:

```
mlsrv10 -sl java (-cp ;\myclasses; -esa) ...
```

On Windows, the following partial mlsrv10 command line defines the LDAP_SERVER system property:

```
mlsrv10 -sl java ( -cp ;\myclasses; -DLDAP_SERVER=huron-ldap ) ...
```

The following partial mlsrv10 command line works on Unix:

```
mlsrv10 -sl java { -cp .:$CLASSPATH:/opt/myclasses:/opt/my.jar: }
```


-sm option

Sets the maximum number of synchronizations that can be actively worked on, which limits the maximum number of network connections as well.

Syntax

mlsrv10 –c "*connection-string***" –sm** *number …*

Remarks

The MobiLink server performs all of the following tasks simultaneously:

1. Read upload data from the network and unpack it.
2. Apply uploads to the consolidated database.
3. Fetch rows to be downloaded from the consolidated database.
4. Pack download data and send it to remote databases.

The number of synchronizations for each task is limited as follows:

- The number of synchronizations doing tasks 2 and 3 is less than or equal to the setting for the mlsrv10 -w option.
- The number of synchronizations doing task 2 is less than or equal to the setting for the mlsrv10 -wu option.
- The number of synchronizations doing all four tasks is less than or equal to the setting for the -sm option.

Higher values for -sm, especially when much greater than -w, allow the MobiLink server to perform more network tasks (1 and 4) than database tasks (2 and 3). This can help ensure that a database worker doesn't have to wait for tasks when network performance might otherwise be a bottleneck. This can improve throughput. However, if -sm is set too high, the MobiLink server can allocate more memory than is directly available, causing the virtual memory paging of the operating system to be activated, which in turn causes memory to be swapped to disk—significantly decreasing throughput.

See also

- "-w option" on page 75
- "-wu option" on page 76

-tx option

When using transactional uploads, this option batches groups of transactions and commits them together.

Syntax

mlsrv10 –c "*connection-string***" –tx** *count* …

Remarks

Use this option to improve performance when doing transactional uploads.

count can be any non-negative value. The default is 1, which means commit every transaction separately. Use a value of zero to perform one commit after all transactions have been uploaded.

See also

- "-tu option" [*MobiLink - Client Administration*]

-ud option

Instructs MobiLink to run as a daemon.

Syntax

mlsrv10 –c "*connection-string***" –ud** …

Remarks

UNIX platforms only.

See also

- "Running the MobiLink server outside the current session" on page 25

-ui option

For Linux with X window server support, starts the MobiLink server in console mode if a usable display isn't available.

Syntax

mlsrv10 –c "*connection-string***" –ui** …

Remarks

When this option is used, the MobiLink server will start whether or not the X window server starts. First, mlsrv10 will try to start with X Windows. If this fails, it will start with a console UI.

When -ui is specified, the server attempts to find a usable display. If it cannot find one, for example because the X window server isn't running, then the MobiLink server starts in console mode.

-ux option

For Linux, opens a console where messages are displayed.

Syntax

mlsrv10 –c "*connection-string* **-ux** …

Remarks

When -ux is specified, the MobiLink server must be able to find a usable display. If it cannot find one, for example because the DISPLAY environment variable is not set or because the X window server is not running, the MobiLink server fails to start.

To run the console in quiet mode, use -q.

On Windows, the console appears automatically.

See also

- "-q option" on page 61

-v option

Allows you to specify what information is logged to the message log file and displayed in the synchronization window.

Syntax

mlsrv10 –c "*connection-string***" –v**[*levels*] …

Remarks

This option is particularly useful when dbmlsync transaction-level uploads are used.

This option controls the type of messages written to the message log file.

If you specify –v alone, the MobiLink server writes a minimal amount of information about each synchronization.

The values of levels are as follows. You can use one or more of these options at once; for example, -vnrsu.

- **+** Turn on all logging options that increase verbosity.
- **c** Show the content of each synchronization script when it is invoked. This level implies s.
- **e** Show system event scripts. These system event scripts are used to maintain MobiLink system tables as well as SQL scripts that control the upload.
- **f** Show first-read errors. This will log errors caused when load-balancing devices check for server liveness by making connections that don't send any data, and thus result in failed synchronizations.

  For TCP/IP connections, you might be better off using the TCP/IP option **ignore**. For more information, see "-x option" on page 77.
- **h** Show the remote schema as uploaded during synchronization.
- **n** Show row-count summaries.
- **p** Show progress offsets.
- **r** Display the column values of each row uploaded or downloaded.
- **s** Show the name of each synchronization script as it is invoked.
- **t** Show the translated SQL that results from scripts that are written in ODBC canonical format. This level implies c. The following example shows the automatic translation of a statement for SQL Anywhere.

  ```
  I. 02/11 11:02:14. [102]: begin_upload synch2
  { call SynchLogLine( ?, ?, 'begin_upload' ) }
  I. 02/11 11:02:14. [102]: Translated SQL:
  call SynchLogLine( ?, ?, 'begin_upload' )
  ```

  The following example shows the translation of the same statement for Microsoft SQL Server.

  ```
  I. 02/11 11:03:21. [102]: begin_upload synch2
  { call SynchLogLine( ?, ?, 'begin_upload' ) }
  I. 02/11 11:03:21. [102]: Translated SQL:
  EXEC SynchLogLine ?, ?, 'begin_upload'
  ```


- **u** Show undefined table scripts. This may help new users understand the synchronization process.

-w option

Sets the number of database worker threads.

Syntax

mlsrv10 –c "*connection-string***" –w** *count* …

Remarks

Each worker thread accepts synchronization requests one at a time.

Each worker thread uses one connection to the consolidated database. The MobiLink server opens one additional connection for administrative purposes. Hence, the minimum number of connections from the MobiLink server to the consolidated database is *count* + 1.

The number of database worker threads has a strong influence on MobiLink synchronization throughput, and you need to run tests to determine the optimum number for your particular synchronization setup. The number of worker threads determines how many synchronizations can be active in the consolidated database simultaneously; the rest will be queued waiting for worker threads to become available. Adding worker threads should increase throughput, but it will also increase the possibility of contention between the active synchronizations. At some point adding more worker threads will decrease throughput, because the increased contention outweighs the benefit of overlapping synchronizations.

The value set for this option is also the default setting for the -wu option, which can be used to limit the number of threads that can simultaneously upload to the consolidated database. This is useful if the optimum number of worker threads for downloading is larger than the optimum number for uploading. The best throughput may be achieved with a large number of worker threads (via -w) with a small number allowed to apply uploads simultaneously (via -wu). In general, the optimum number for -wu depends on the consolidated database, and is relatively independent of the processing or network speeds for the remote databases. Therefore, when you increase the number of threads with -w, you may want to use -wu to restrict the number that can upload simultaneously. For more information, see “-wu option” on page 76.

The default number of worker threads is **5**.

See also

- “-wu option” on page 76
- “-sm option” on page 68

-wu option

Sets the maximum number of database worker threads that can apply uploads to the consolidated database simultaneously.

Syntax

mlsrv10 –c "*connection-string***" –wu** *count …*

Remarks

Use the -wu option to limit the number of worker threads that can simultaneously apply uploads to the consolidated database. When the limit is reached, a worker thread that is ready to apply its upload to the consolidated database must wait until another finishes its upload.

The most common cause of contention in the consolidated database is having too many worker threads applying uploads simultaneously. Downloads cause far less contention, so they are limited only be the mlsrv10 -w option. for this reason, the -w setting must be greater than or equal to the -wu setting.

By default, all worker threads can apply uploads simultaneously. The number of worker threads that are used is set by the -w option. The default is **5**.

Example

In a pilot setup using a LAN and remote databases on PCs, you find that the optimum number of worker threads is approximately 10 for both upload-only and download-only synchronizations, and that corresponds to 100% CPU utilization on the consolidated database. With fewer worker threads you find that throughput is less and the CPU utilization for the consolidated database is lower. With more worker threads, throughput does not increase because the consolidated database is already processing as fast as it can with 10 workers.

See also

- “-w option” on page 75
- “-sm option” on page 68

-x option

Sets network protocol and protocol options for MobiLink clients. These are used by the MobiLink server to listen for synchronization requests.

Separately licensed component required
ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.
See "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

Syntax

mlsrv10 –c "*connection-string***" –x** *protocol*[*protocol-options*] …

protocol : **tcpip** | **tls** | **http** | **https**

protocol-options : **(** *option*=*value*; ... **)**

Default

The default is **TCPIP** with port **2439**.

Parameters

The allowed values of *protocol* are as follows:

- **tcpip** Accept connections via TCP/IP.
- **tls** Accept connections via TCP/IP using transport-layer security.
- **http** Accept connections via the standard Web protocol.
- **https** Accept connections via a variant of HTTP that handles secure transactions. The HTTPS protocol implements HTTP over SSL/TLS using RSA or ECC encryption.

You can also specify the following network protocol options, in the form *option*=*value*. You should separate multiple options with semicolons.

- **TCP/IP options**
  If you specify the **tcpip** protocol, you can optionally specify the following protocol options:

TCP/IP protocol option	Description
host=*hostname*	The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.

TCP/IP protocol option	Description
ignore=*hostname*	A host name or IP number that will be ignored by the MobiLink server if it makes a connection. This option allows you to ignore requests from load balancers at the lowest possible level, preventing excessive output in the MobiLink server log and MobiLink Monitor output files. You can specify multiple hosts to ignore; for example `-x tcpip(ignore=lb1;ignore=123.45.67.89)`. If you specify multiple instances of -x on a command line, the host is ignored on all instances; for example, if you specify -x tcpip(ignore=1.1.1.1) -x http, then connections for 1.1.1.1 are ignored on both the TCP/IP and the HTTP streams. However, this does not affect connections via the -xo option.
port=*portnumber*	The socket port number on which the MobiLink server should listen. The default port is **2439**, which is the IANA registered port number for the MobiLink server.

- **Options for TCP/IP with transport-layer security**
  If you specify the **tls** protocol, which is TCP/IP with transport-layer security, you can optionally specify the following protocol options:

TLS protocol options	Description
fips={**yes**|**no**}	If you specify the TLS protocol with tls_type=rsa, you can specify fips=yes to accept connections using the TCP/IP protocol and FIPS-approved algorithms for encryption. FIPS connections use separate FIPS 140-2 certified software. Servers using RSA encryption without FIPS are compatible with clients using RSA with FIPS, and servers using RSA with FIPS are compatible with clients using RSA without FIPS.
host=*hostname*	The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.
ignore=*hostname*	A host name or IP number that will be ignored by the MobiLink server if it makes a connection. This option allows you to ignore requests from load balancers at the lowest possible level, preventing excessive output in the MobiLink server log and MobiLink Monitor output files. You can specify multiple hosts to ignore; for example `-x tcpip(ignore=lb1;ignore=123.45.67.89)`.
port=*portnumber*	The socket port number on which the MobiLink server should listen. The default port is **2439**, which is the IANA registered port number for the MobiLink server.

TLS protocol options	Description
tls_type={**rsa**|**ecc**}	If you specify the TCP/IP protocol as tls, you can specify either elliptic-curve cryptography (ecc) or RSA encryption (rsa). For backward compatibility, **ecc** can also be specified as **certicom**. The default tls_type is **rsa**. When you use TLS, you must specify a certificate and certificate password: ♦ **certificate=certificate_file** Specify the path and file name of the certificate that is to be used for server authentication. ♦ **certificate_password=password** Specify the password for the certificate. See "Starting the MobiLink server with transport-layer security" [*SQL Anywhere Server - Database Administration*].

♦ **HTTP options**
If you specify the **http** protocol, you can optionally specify the following protocol options:

HTTP options	Description
buffer_size=*number*	The maximum body size for an HTTP message sent from MobiLink server, in bytes. Changing the option will decrease or increase the amount of memory allocated for sending HTTP messages. The default is **65535** bytes.
host=*hostname*	The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.
port=*portnumber*	The socket port number on which the MobiLink server should listen. The port number must match the port the MobiLink server is setup to monitor. The default port is **80**.
version=*http-version*	The MobiLink server automatically detects the HTTP version used by a client. This parameter is a string specifying the default version of HTTP to use in case the server cannot detect the method used by the client. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

♦ **HTTPS options** The HTTPS protocol uses RSA or ECC digital certificates for transport-layer security. If you specify FIPS encryption, the protocol uses separate FIPS 140-2 certified software that is compatible with https.
For more information, see "Starting the MobiLink server with transport-layer security" [*SQL Anywhere Server - Database Administration*].If you specify the **https** protocol, you can optionally specify the following protocol options:

HTTPS options	Description
buffer_size=*number*	The maximum body size for an HTTPS message sent from MobiLink server, in bytes. Changing the option will decrease or increase the amount of memory allocated for sending HTTPS messages. The default is **65535** bytes.

HTTPS options	Description
certificate=*server-certificate*	The path and file name of the certificate that is to be used for server authentication. For HTTPS, this must be an RSA certificate.
certificate_password=*password*	An optional parameter that specifies a password for the certificate. See "Transport-Layer Security" [*SQL Anywhere Server - Database Administration*].
fips={**yes**|**no**}	You can specify fips=yes to accept connections using the HTTPS protocol and FIPS-approved algorithms for encryption. FIPS connections use separate FIPS 140-2 certified software. Servers using RSA encryption without FIPS are compatible with clients using RSA with FIPS, and servers using RSA with FIPS are compatible with clients using RSA without FIPS.
host=*hostname*	The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.
port=*portnumber*	The socket port number on which the MobiLink server should listen. The port number must match the port the MobiLink server is set up to monitor. The default port is **443**.
tls_type={**rsa**|**ecc**}	If you specify the TCP/IP protocol as tls, you can specify either elliptic-curve cryptography (ecc) or RSA encryption (rsa). For backward compatibility, **ecc** can also be specified as **certicom**. The default tls_type is **rsa**. When you use transport-layer security, you must specify a certificate and certificate password: ♦ **certificate=certificate_file** Specify the path and file name of the certificate that is to be used for server authentication. ♦ **certificate_password=password** Specify the password for the certificate. See "Starting the MobiLink server with transport-layer security" [*SQL Anywhere Server - Database Administration*].
version=*http-version*	The MobiLink server automatically detects the HTTP version used by a client. This parameter is a string specifying the default version of HTTP to use in case the server cannot detect the method used by the client. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

Example

The following command line sets the port to 12345:

```
mlsrv10 -c "dsn=SQL Anywhere 10 CustDB;uid=DBA;pwd=sql" -x tcpip(port=12345)
```

The following example specifies the type of security (RSA), the server certificate, and the password protecting the server's private key:


```
mlsrv10 -c "dsn=my_cons"
 -x tls(tls_type=rsa;certificate=c:\test
\serv_rsa1.crt;certificate_password=pwd)
```


-xo option

Sets network protocol and protocol options for version 8 and 9 MobiLink clients.

Syntax

mlsrv10 –c "*connection-string***"**
–xo *protocol*[*protocol-options*] …

protocol-options : **(** *keyword***=***value*; ... **)**

Remarks

Specify communications protocol through which to communicate with client applications. The default is **TCPIP** with port **2439**.

The allowed values of *protocol* are as follows:

- **tcpip** Accept connections from applications via TCP/IP.
- **http** Accept connections via the standard Web protocol. Client applications can pick their HTTP version and the MobiLink server adjusts on a per-connection basis.
- **https** Accept connections via a variant of HTTP that handles secure transactions. The HTTPS protocol implements HTTP over SSL/TLS using RSA encryption, and is compatible with any other HTTPS server.
- **https_fips** Accept connections using the HTTPS protocol and FIPS-approved algorithms for encryption. HTTPS_FIPS uses separate FIPS 140-2 certified software. Servers using rsa_tls are compatible with clients using rsa_tls_fips, and servers using rsa_tls_fips are compatible with clients using rsa_tls.

Optionally, you can also specify network protocol options, in the form *option***=***value*. You should separate multiple options with semicolons. Which options you specify depends on the protocol you choose.

- **TCP/IP options** If you specify the **tcpip** protocol, you can optionally specify the following protocol options:
  - **backlog=number-of-connections** The maximum number of remote connections before MobiLink server should reject new synchronization requests, causing synchronization to fail on the client side. By specifying a backlog size, you can prevent clients from waiting to synchronize when the server is busy. If you do not specify a backlog size, MobiLink server accepts as many connections as possible, potentially reaching or exceeding the operating system's limit on network connections. This may cause slow or erratic behavior.

    When a client attempts to synchronize with a MobiLink server that has accepted its maximum number of remote connections, it receives the error code -85 (SQLE_COMMUNICATIONS_ERROR). The client application should handle this error and try to connect again in a few minutes.

    For more information about SQLE_COMMUNICATIONS_ERROR, see "Communication error" [*SQL Anywhere 10 - Error Messages*].

If you are using MobiLink in an environment where thousands of simultaneous synchronizations are possible, use the backlog option to specify a maximum number of remote connections that is less than the operating system limit. See "Plan for operating system limitations" on page 138.

- **host=hostname** The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.
- **ignore=hostname** A host name or IP number that will be ignored by the MobiLink server if it makes a connection. This option allows you to ignore requests from load balancers at the lowest possible level, preventing excessive output in the MobiLink server log and MobiLink Monitor output files. You can specify multiple hosts to ignore; for example `-x tcpip (ignore=lb1;ignore=123.45.67.89)`.
- **liveness_timeout=n** The amount of time, in seconds, after the last communication with a client before MobiLink aborts the synchronization. A value of 0 means that there is no timeout. This option is only effective if download acknowledgement on the client is set to off (the default). The default is **120** seconds.
- **port=portnumber** The socket port number on which the MobiLink server should listen. The default port is **2439**, which is the IANA registered port number for the MobiLink server.

> **Note**
> The mlsrv10 -x and -xo options both use the same default port and -x is started even if you don't specify it, so you must change the port for -xo unless you change it in the -x option.

- **security=cipher(keyword=value;...)** All communication through this connection is to be encrypted and authenticated using transport-layer security. *Cipher* can be one of:

Cipher	Description
rsa_tls	RSA encryption.
rsa_tls_fips	RSA encryption that is FIPS-approved. rsa_tls_fips uses separate FIPS 140-2 certified software but is compatible with clients using https (and version 9.0.2 or later).
ecc_tls	Elliptic-curve cryptography. For backward compatibility, **ecc_tls** can also be specified as **certicom_tls**.

The security parameters are **certificate** (the path and file name of the certificate that is to be used for server authentication), and **certificate_password**. You must use a certificate that matches the cipher suite you choose.

For more information, see "Starting the MobiLink server with transport-layer security" [*SQL Anywhere Server - Database Administration*].

> **Separately licensed component required**
> ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.
> See "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

- **HTTP options** If you specify the **http** protocol, you can optionally specify the following protocol options:
  - **backlog=number-of-connections** The maximum number of remote connections before MobiLink server should reject new synchronization requests, causing synchronization to fail on the client side. By specifying a backlog size, you can prevent clients from waiting to synchronize when the server is busy. If you do not specify a backlog size, MobiLink server accepts as many connections as possible, potentially reaching or exceeding the operating system's limit of network connections. This may cause slow or erratic behavior.

    When a client attempts to synchronize with a MobiLink server that has accepted its maximum number of remote connections, it receives the error code -85 (SQLE_COMMUNICATIONS_ERROR). The client application should handle this error and try to connect again in a few minutes.

    For more information about SQLE_COMMUNICATIONS_ERROR, see "Communication error" [*SQL Anywhere 10 - Error Messages*].

    If you are using MobiLink in an environment where thousands of simultaneous synchronizations are possible, use the backlog option to specify a maximum number of remote connections that is less than the operating system limit. For more information, see "Plan for operating system limitations" on page 138.
  - **buffer_size=number** The maximum body size for an HTTP message sent from MobiLink server, in bytes. Changing the option will decrease or increase the amount of memory allocated for sending HTTP messages. The default is **65535** bytes.
  - **contd_timeout=seconds** The number of seconds the MobiLink server waits to receive the next part of a partially completed synchronization before the synchronization is abandoned. You can tune this option to free MobiLink server resources when the wait time indicates that the client will never continue the connection. The default value is **30** seconds.
  - **host=hostname** The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.
  - **port=portnumber** The socket port number on which the MobiLink server should listen. The port number must match the port the MobiLink server is set up to monitor. The default port is **80**.

    Note
    The mlsrv10 -x and -xo options both use the same default port and -x is started even if you don't specify it, so you must change the port for -xo unless you change it in the -x option.
  - **session_key={cookie|header}**
    Creates an alternative to the JSESSIONID for tracking connections. This may be necessary if your corporate infrastructure already uses the JSESSIONID.
  - **unknown_timeout=seconds** The number of seconds the MobiLink server waits to receive HTTP headers on a new connection before the synchronization is abandoned. You can tune this option to free MobiLink server resources when the wait time indicates that a network failure has occurred. The default value is **30** seconds.

- **version=http-version** The MobiLink server automatically detects the HTTP version used by a client. This parameter is a string specifying the default version of HTTP to use in case the server cannot detect the method used by the client. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

- **HTTPS or HTTPS_FIPS options** The https protocol uses RSA digital certificates for transport-layer security. The https_fips protocol uses separate FIPS 140-2 certified software but is compatible with https.

  For more information, see "Starting the MobiLink server with transport-layer security" [*SQL Anywhere Server - Database Administration*].

  Separately licensed component required
  ECC encryption and FIPS-certified encryption require a separate license. All strong encryption technologies are subject to export regulations.
  See "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

  If you specify the **https** protocol, you can optionally specify the following protocol options:

  - **backlog=number-of-connections** The maximum number of remote connections before MobiLink should reject new synchronization requests, causing synchronization to fail on the client side. By specifying a backlog size, you can prevent clients from waiting to synchronize when the server is busy. If you do not specify a backlog size, clients will attempt to synchronize regardless of the size of the backlog.

  - **buffer_size=number** The maximum body size for an HTTPS message sent from MobiLink server, in bytes. Changing the option will decrease or increase the amount of memory allocated for sending HTTPS messages. The default is **65535** bytes.

  - **contd_timeout=seconds** The number of seconds the MobiLink server waits to receive the next part of a partially completed synchronization before the synchronization is abandoned. You can tune this option to free MobiLink server resources when the wait time indicates that the client will never continue the connection. The default value is **30** seconds.

  - **host=hostname** The host name or IP number on which the MobiLink server should listen. The default value is **localhost**.

  - **port=portnumber** The socket port number on which the MobiLink server should listen. The port number must match the port the MobiLink server is set up to monitor. The default port is **443**.

    Note
    The mlsrv10 -x and -xo options both use the same default port and -x is started even if you don't specify it, so you must change the port for -xo unless you change it in the -x option.

  - **certificate** The path and file name of the certificate that is to be used for server authentication. This must be an RSA certificate.

  - **certificate_password** An optional parameter that specifies a password for the certificate.

For more information about security, see "Transport-Layer Security" [*SQL Anywhere Server - Database Administration*].

- **session_key={cookie|header}**
  Creates an alternative to the JSESSIONID for tracking connections. This may be necessary if your corporate infrastructure already uses the JSESSIONID.
- **unknown_timeout=seconds** The number of seconds the MobiLink server waits to receive HTTP headers on a new connection before the synchronization is abandoned. You can tune this option to free MobiLink server resources when the wait time indicates that a network failure has occurred. The default value is **30** seconds.
- **version=http-version** The MobiLink server automatically detects the HTTP version used by a client. This parameter is a string specifying the default version of HTTP to use in case the server cannot detect the method used by the client. You have a choice of **1.0** or **1.1**. The default value is **1.1**.

Example

The following command line sets the backlog size to 10 connections.

```
mlsrv10 -c "dsn=SQL Anywhere 10 CustDB;uid=DBA;pwd=sql" -x http(backlog=10)
```

-zp option

Adjusts which timestamp values will be used for conflict detection purposes.

Syntax

mlsrv10 –c "*connection-string***" –zp**

Remarks

In the event of a timestamp conflict between the consolidated and remote database, this option allows timestamp values with a precision higher than the lowest precision to be used for conflict detection purposes. The option is useful when timestamps in the consolidated database are more precise than in the remote, as updated timestamps on the remote can cause conflicts in the next synchronization. The option allows MobiLink to ignore these conflicts. When there is a precision mismatch and -zp is not used, a per synchronization and a schema sensitive per table warning are written to the log to advertise the -zp option. Another per synchronization warning is also added to advise users to adjust the timestamp precision on the remote database where possible.

-zs option

Specifies a MobiLink server name for mlstop.

Syntax

mlsrv10 –c "*connection-string***" –zs** *name*

Remarks

The name that is specified may include ASCII letters and numbers, but no other characters.

When mlstop is used to shut down a MobiLink server started with the -zs option, you must specify the server name on the mlstop command line. For example, `mlstop myMLserver`. Shutdown may only be initiated from the computer where the MobiLink server is installed.

See also

- "MobiLink stop utility [mlstop]" on page 547

-zt option

Specifies the maximum number of processors used to run the MobiLink server.

Syntax

mlsrv10 –c "*connection-string***" –zt** *number*

Remarks

This option may be required for some ODBC drivers. It also gives you fine control of processor resources.

This option can only be used on Windows operating systems. The default is the number of processors on the computer.

-zu option

Controls the automatic addition of users when the authenticate_user script is undefined.

Syntax

mlsrv10 –c "*connection-string***" –zu**{ + | - } …

Remarks

If this is supplied as -zu+, then unrecognized MobiLink user names are added to the ml_user table on first synchronizing. If the argument is supplied as -zu-, or not supplied, unrecognized user names are prevented from synchronizing.

This option is useful during development to register users. It is not recommended for deployed applications.

See also

- "MobiLink Users" [*MobiLink - Client Administration*]
- "MobiLink user authentication utility [mluser]" on page 548
- "authenticate_user connection event" on page 256

-zus option

Causes the MobiLink server to invoke upload scripts for a table even when no rows are uploaded for the table.

Syntax

mlsrv10 –c "*connection-string***" –zus** …

Remarks

By default, if no rows are uploaded for a table, the MobiLink server will not invoke upload scripts for that table, even if they are defined. This option overrides the default behavior and causes the MobiLink server to call upload scripts for a table even if no rows are uploaded.

-zw option

Controls which levels of warning message to display.

Syntax

mlsrv10 –c "*connection-string***" –zw** *levels*

Remarks

MobiLink has five levels of warning messages:

Level	Description
0	Suppress all warning messages
1	Server and high ODBC level: warning messages when the MobiLink server starts
2	Synchronization and user level: warning messages when a synchronization starts
3	Schema level: warning messages when a MobiLink server is processing a client schema
4	Script and lower ODBC level: warning messages when a MobiLink server fetches, prepares, or executes scripts
5	Table or row level: warning messages when a MobiLink server performs table operations in an upload or download

To specify the level of warning messages you want reported, you can separate levels with a comma, or separate a range with two dots. For example, **-zw 1..3,5** is the same as **-zw 1,2,3,5**.

The reporting of messages has a slight impact on performance. Levels with a higher number tend to produce more messages.

If -zw is used more than once in the same command line, MobiLink recognizes only the last instance. If settings of -zw, -zwd, and -zwe conflict, MobiLink gives priority to -zwe, then -zwd, then -zw.

The default is **1,2,3,4,5**, which indicates that all levels of warning message should be displayed.

-zwd option

Disables specific warning codes.

Syntax

mlsrv10 –c "*connection-string***" –zwd** *code*, …

Remarks

You can disable specific warning codes so that they will not be reported, even though other codes of the same level are reported.

For a complete list of warning message codes, see "MobiLink Server Warning Messages" [*SQL Anywhere 10 - Error Messages*].

If -zwd is used more than once in the same command line, MobiLink accumulates the settings. If settings of -zw, -zwd, and -zwe conflict, MobiLink gives priority to -zwe, then -zwd, then -zw.

-zwe option

Enables specific warning codes.

Syntax

mlsrv10 –c "*connection-string***" –zwe** *code*, …

Remarks

You can enable specific warning codes so that they will be reported even though you have disabled other codes of the same level using -zw.

For a complete list of warning message codes, see "MobiLink Server Warning Messages" [*SQL Anywhere 10 - Error Messages*].

If -zwe is used more than once on the same command line, MobiLink accumulates the settings. If settings of -zw, -zwd, and -zwe conflict, MobiLink gives priority to -zwe, then -zwd, then -zw.

CHAPTER 4

Synchronization Techniques

Contents

MobiLink development tips

Adding synchronization functionality to an application adds an added level of complexity to your application. The following tips may be useful.

When you are adding synchronization to a prototype application, it can be difficult to see which components are causing problems. When developing a prototype, temporarily hard code INSERT statements in your application to provide data for testing and demonstration purposes. Once your prototype is working correctly, enable synchronization and discard the temporary INSERT statements.

Start with straightforward synchronization techniques. Operations such as a simple upload or download require only one or two scripts. Once those are working correctly, you can introduce more advanced techniques, such as timestamps, primary key pools, and conflict resolution.

MobiLink and primary keys

In a synchronization system, the primary key is the only way to identify the same row in different databases (remote and consolidated) and the only way to detect conflicts. Therefore, MobiLink applications must adhere to the following rules:

- Every table that is to be synchronized must have a primary key.
- Never update the values of primary keys.
- Primary keys must be unique across all synchronized databases.

See "Maintaining unique primary keys" on page 106.

Timestamp-based downloads

The timestamp method is the most useful general technique for efficient synchronization. This technique involves tracking the last time that each user synchronized and only downloading rows that have changed since then.

MobiLink maintains a timestamp value indicating when each MobiLink user last downloaded data. This value is called the **last download time**.

See "Using last download times in scripts" on page 98.

♦ To implement timestamp-based synchronization for a table

1. At the consolidated database, add a column that holds the most recent time the row was modified. The column is typically declared as follows:

DBMS	last modified column
SQL Anywhere	`timestamp DEFAULT timestamp`
Adaptive Server Enterprise	`datetime`
Microsoft SQL Server	`datetime`
Oracle	`date`
IBM DB2 UDB	`timestamp`

2. In scripts for the download_cursor and download_delete_cursor events, compare the first parameter to the value in the timestamp column.

Example

The following table declaration and scripts implement timestamp-based synchronization on the Customer table in the Contact sample:

♦ Table definition:

```
CREATE TABLE "DBA"."Customer"(
   "cust_id"  integer NOT NULL DEFAULT GLOBAL AUTOINCREMENT,
   "name"  char(40) NOT NULL,
   "rep_id"  integer NOT NULL,
   "last_modified" timestamp NULL DEFAULT timestamp,
   "active"  bit NOT NULL,
   PRIMARY KEY ("cust_id") )
```

♦ download_delete_cursor script:

```
SELECT cust_id
FROM Customer JOIN SalesRep
ON Customer.rep_id = SalesRep.rep_id
WHERE Customer.last_modified >= {ml s.last_table_download}
    AND ( SalesRep.ml_username != {ml s.username}
          OR Customer.active = 0 )
```


- download_cursor script:

```
SELECT cust_id, Customer.name, Customer.rep_id
FROM Customer KEY JOIN SalesRep
WHERE Customer.last_modified >= {ml s.last_table_download}
    AND SalesRep.ml_username = {ml s.username}
    AND Customer.active = 1
```

See "Synchronization logic source code" [*MobiLink - Getting Started*] and "Synchronizing contacts in the Contact sample" [*MobiLink - Getting Started*].

Using last download times in scripts

The last download timestamp is provided as a parameter to many MobiLink events. The last download timestamp is the value obtained from the consolidated database during the last successful synchronization immediately prior to the download phase. If the current MobiLink user has never synchronized, or has never synchronized successfully, this value is set to 1900-01-01.

See "How download timestamps are generated and used" on page 99.

If you have multiple publications and have synchronized them at different times, then you can have two different last download timestamps. For this reason, there are two script parameter names for last download timestamps:

- **last_table_download** is the last download timestamp for a table.
- **last_download** is the last time all tables were synchronized. It is the earliest last_table_download value for any table.

When you use question marks instead of named parameters in MobiLink scripts, the correct value is always used.

Caution
If you are using a SQL Anywhere consolidated database and the column holding last modified information is of type DEFAULT TIMESTAMP, then the column should not be synchronized. If your remote databases require such a column, a different column name should be used. Otherwise, the default timestamp value may be overridden by the uploaded value, and will not contain the time that the row was last modified on the consolidated database.

See also

- "Script parameters" on page 221

Example

```
SELECT cust_id, Customer.name, Customer.rep_id
FROM Customer KEY JOIN SalesRep
WHERE Customer.last_modified >= {ml s.last_table_download}
    AND SalesRep.ml_username = {ml s.username}
    AND Customer.active = 1
```


How download timestamps are generated and used

MobiLink generates and uses a timestamp for timestamp-based downloads as follows:

- After an upload is committed and immediately before invoking the prepare_for_download event, the MobiLink server fetches the current time from the consolidated database and saves the value. This timestamp value represents the start time of the current download; the next synchronization should only download data that changes after this time.
- The MobiLink server sends this timestamp value as part of the download, and the client stores it. SQL Anywhere clients store it in the ISYSSYNC system table.
- The next time the client synchronizes, it uses the timestamp value for the **last_download_timestamp** that it sends with the upload.
- The MobiLink server passes the last_download_timestamp that the client just uploaded into your download_cursor and download_delete_cursor. Your cursor can then select changes with timestamps that are newer or equal to the last last_download_timestamp to ensure that only new changes are downloaded.

In some rare circumstances, you may want to modify the last_download_timestamp. For example, if you accidentally delete all the data on a remote database, you can resynchronize it by defining a modify_last_download_timestamp connection script to reset the value for the last download timestamp. There is another event, called modify_next_last_download_timestamp, which you can use to reset the timestamp not for the current synchronization but for the next synchronization.

See also

- "Using last download times in scripts" on page 98
- "modify_last_download_timestamp connection event" on page 357
- "modify_next_last_download_timestamp connection event" on page 360

Dealing with daylight savings time

Daylight savings time can cause problems in a distributed database system if data is synchronized during the hour that the time changes. In fact, you can lose data. This is only an issue in the autumn when the time goes back and there is a one-hour period that can be ambiguous.

To deal with daylight savings time, you have three possible solutions:

- Ensure that the consolidated database server is using UTC time.
- Turn off daylight savings time on the consolidated database server.
- Shut down for an hour when the time changes.

Snapshot synchronization

Timestamp-based synchronization is appropriate for most synchronizations. However, occasionally you may want to update a snapshot of your data.

Snapshot synchronization of a table is a complete download of all relevant rows in the table, even if they have been downloaded before. This is the simplest synchronization method, but can involve unnecessarily large data sets being exchanged, which can limit performance.

You can use snapshot synchronization for downloading all the rows of the table, or in conjunction with a partitioning of the rows. See "Partitioning rows among remote databases" on page 102.

When to use snapshot synchronization

The snapshot method is typically most useful for tables that have both the following characteristics.

- **Relatively few rows** When there are few rows, the overhead associated with downloading all of them is small.
- **Rows that change frequently** When most rows in a table change frequently, there is little to be gained by explicitly excluding those that have not changed since the last synchronization.

A table that holds a list of exchange rates could be suited to this approach because there are relatively few currencies, but the rates of most change frequently. Depending on the nature of the business, a table that holds prices, a list of interest rates, or current news items could all be candidates.

♦ To implement snapshot-based synchronization

1. Leave the upload scripts undefined unless remote users update the values.
2. If the table may have rows deleted, write a download_delete_cursor script that deletes all the rows from the remote table, or at least all rows no longer required. Do not delete the rows from the consolidated database; rather, mark them for deletion. You must know the row values to delete them from the remote database.

   See "Writing download_delete_cursor scripts" on page 234.
3. Write a download_cursor script that selects all the rows you want to include in the remote table.

Deleting rows

Rather than deleting rows from the consolidated database, mark them for deletion. You must know the row values to delete them from the remote database. Select only unmarked rows in the download_cursor script and only marked rows in the download_delete_cursor script.

The download_delete_cursor script is executed before the download_cursor script. If a row is to be included in the download, you need not include a row with the same primary key in the delete list. When a downloaded row is received at the remote location, it replaces a preexisting row with the same primary key.

See "Writing scripts to download rows" on page 232.

An alternative deletion technique

Rather than delete rows from the remote database using a download_cursor script, you can allow the remote application to delete the rows. For example, immediately following synchronization, you could allow the application to execute SQL statements that delete the unneeded rows.

Rows deleted by the application are ordinarily uploaded to the MobiLink server upon the next synchronization, but you can prevent this upload using the STOP SYNCHRONIZATION DELETE statement. For example,

```
STOP SYNCHRONIZATION DELETE;
DELETE FROM table-name
    WHERE expiry_date < CURRENT TIMESTAMP;
COMMIT;
START SYNCHRONIZATION DELETE;
```

See "Writing download_delete_cursor scripts" on page 234.

Snapshot example

The ULProduct table in the sample application is maintained by snapshot synchronization. The table contains relatively few rows, and for this reason, there is little overhead in using snapshot synchronization.

1. There is no upload script. This reflects a business decision that products cannot be added at remote databases.
2. There is no download_delete_cursor, reflecting an assumption that products are not removed from the list.
3. The download_cursor script selects the product identifier, price, and name of every current product. If the product is pre-existing, the price in the remote table will be updated. If the product is new, a row will be inserted in the remote table.

   ```
   SELECT prod_id, price, prod_name
   FROM ULProduct
   ```

For another example of snapshot synchronization in a table with very few rows, see "Synchronizing sales representatives in the Contact sample" [*MobiLink - Getting Started*].

Partitioning rows among remote databases

Each MobiLink remote database can contain a different subset of the data in the consolidated database. This means that you can write your synchronization scripts so that data is **partitioned** among remote databases.

The partitioning can be disjoint, or it can contain overlaps. For example, if each employee has their own set of customers, with no shared customers, the partitioning is **disjoint**. If there are shared customers who appear in more than one remote database, the partitioning contains **overlaps**.

Partitioning is implemented in the download_cursor and download_delete_cursor scripts for the table, which define the rows to be downloaded to the remote database. Each of these scripts takes a MobiLink user name as a parameter. By defining your scripts using this parameter in the WHERE clause, each user gets the appropriate rows.

Disjoint partitioning

Partitioning is controlled by the download_cursor and download_delete_cursor scripts for each table involved in synchronization. These scripts take two parameters, a last download timestamp and the MobiLink user name supplied in the call to synchronize.

♦ To partition a table among remote databases

1. Include in the table definition a column containing the synchronization user name in the consolidated database. You need not download this column to remote databases.
2. Include a condition in the WHERE clause of the download_cursor and download_delete_cursor scripts requiring this column to match the script parameter.

   The script parameter can be represented by a question mark or a named parameter in the script. For example, the following download_cursor script partitions the Contact table by employee ID.

   ```
   SELECT id, contact_name
   FROM Contact
   WHERE last_modified >= {ml s.last_table_download}
   AND emp_id = {ml s.username}
   ```

   See “download_cursor table event” on page 294 and “download_delete_cursor table event” on page 297.

Example

The primary key pool tables in the CustDB sample application are used to supply each remote database with its own set of primary key values. This technique is used to avoid duplicate primary keys, and is discussed in “Using primary key pools” on page 110.

A necessary feature of the method is that primary key-pool tables must be partitioned among remote databases in a disjoint fashion.

One key-pool table is ULCustomerIDPool, which holds primary key values for each user to use when they add customers. The table has three columns:

- **pool_cust_id** A primary key value for use in the ULCustomer table. This is the only column downloaded to the remote database.
- **pool_emp_id** The employee who owns this primary key.
- **last_modified** This table is maintained using the timestamp technique, based on the last_modified column.

  For information on timestamp synchronization, see "Timestamp-based downloads" on page 97.

The download_cursor script for this table is as follows.

```
SELECT pool_cust_id
FROM ULCustomerIDPool
WHERE last_modified >= {ml s.last_table_download}
    AND pool_emp_id = {ml s.username}
```

When not using a variable or named parameter, you can use a join or sub-selection that includes the **?** placeholder.

See "Synchronizing customers in the Contact sample" [*MobiLink - Getting Started*] and "Synchronizing contacts in the Contact sample" [*MobiLink - Getting Started*].

Partitioning with overlaps

Some tables in your consolidated database may have rows that belong to many remote databases. Each remote database has a subset of the rows in the consolidated database and the subset overlaps with other remote databases. This is frequently the case with a customer table. In this case, there is a many-to-many relationship between the table and the remote databases and there will usually be a table to represent the relationship. The scripts for the download_cursor and download_delete_cursor events need to join the table being downloaded to the relationship table.

Example

The CustDB sample application uses this technique for the ULOrder table. The ULEmpCust table holds the many-to-many relationship information between ULCustomer and ULEmployee.

Each remote database receives only those rows from the ULOrder table for which the value of the emp_id column matches the MobiLink user name.

The SQL Anywhere version of the download_cursor script for ULOrder in the CustDB application is as follows:

```
SELECT o.order_id, o.cust_id, o.prod_id,
   o.emp_id, o.disc, o.quant, o.notes, o.status
FROM ULOrder o , ULEmpCust ec
WHERE o.cust_id = ec.cust_id
    AND ec.emp_id = {ml s.username}
    AND ( o.last_modified >= {ml s.last_table_download}
        OR ec.last_modified >= {ml s.last_table_download})
    AND     ( o.status IS NULL
        OR o.status != 'Approved' )
    AND ( ec.action IS NULL )
```


This script is fairly complex. It illustrates that the query defining a table in the remote database can include more than one table in the consolidated database. The script downloads all rows in ULOrder for which all of the following are true:

- the cust_id column in ULOrder matches the cust_id column in ULEmpCust
- the emp_id column in ULEmpCust matches the synchronization user name
- the last modification of either the order or the employee-customer relationship was later than the most recent synchronization time for this user
- the status is anything other than **Approved**

The action column on ULEmpCust is used to mark columns for delete. Its purpose is not relevant to the current topic.

The download_delete_cursor script is as follows.

```
SELECT o.order_id
FROM ULOrder o, ULEmpCust ec
WHERE o.cust_id = ec.cust_id
  AND ec.emp_id = {ml s.username}
  AND ( o.last_modified >= {ml s.last_table_download} OR
        c.last_modified >= {ml s.last_table_download} )
  AND ( o.status IS NULL OR
        o.status != 'Approved' )
  AND ( ec.action IS NULL )
```

This script deletes all approved rows from the remote database.

Partitioning child tables

The example in "Partitioning with overlaps" on page 103 illustrates how to partition tables based on a criterion in some other table.

Some tables in your remote database may have disjoint subsets or overlapping subsets, but do not contain a column that determines the subset. These are child tables that usually have a foreign key (or a series of foreign keys) referencing another table. The referenced table has a column that determines the correct subset.

In this case, the download_cursor script and the download_delete_cursor script need to join the referenced tables and have a WHERE clause that restricts the rows to the correct subset.

For an example, see "Synchronizing contacts in the Contact sample" [*MobiLink - Getting Started*].

Upload-only and download-only synchronizations

By default, synchronization is bi-directional: data is both uploaded and downloaded. However, you can choose to do only an upload or only a download.

Synchronization model note
This topic provides information for how to set up upload-only and download-only synchronization when you create your MobiLink synchronization system in your database. You can also specify upload-only or download-only if you create a synchronization model in Sybase Central.

SQL Anywhere remote databases

- **Upload** To perform upload-only synchronization, use the dbmlsync option -uo or the extended option UploadOnly. See:
  - "-uo option" [*MobiLink - Client Administration*]
  - "UploadOnly (uo) extended option" [*MobiLink - Client Administration*]
- **Download** To perform download-only synchronization, use the dbmlsync option -ds or the extended option DownloadOnly. See:
  - "-ds option" [*MobiLink - Client Administration*]
  - "DownloadOnly (ds) extended option" [*MobiLink - Client Administration*]

  SQL Anywhere remote databases can also use download-only publications. This approach to downloads is different from download-only synchronizations. See "Download-only publications" [*MobiLink - Client Administration*].

UltraLite remote databases

- **Upload** To perform upload-only synchronization, use the Upload Only synchronization parameter.

  See "Upload Only synchronization parameter" [*MobiLink - Client Administration*].
- **Download** To perform download-only synchronization, use the Download Only synchronization parameter.

  See "Download Only synchronization parameter" [*MobiLink - Client Administration*].

Maintaining unique primary keys

Every table that is to be synchronized must have a primary key, and the primary key must be unique across all synchronized databases. The values of primary keys should not be updated.

It is often convenient to use a single column as the primary key for tables. For example, each customer should be assigned a unique identification value. If all the sales representatives work in an environment where they can maintain a direct connection to the database, assigning these numbers is easily accomplished. Whenever a new customer is inserted into the customer table, automatically add a new primary key value that is greater than the last value.

In a disconnected environment, assigning unique values for primary keys when new rows are inserted is not as easy. When a sales representative adds a new customer, she is doing so to a remote copy of the Customer table. You must prevent other sales representatives, working on other copies of the Customer table, from using the same customer identification value.

This section describes the following ways to solve the problem of how to generate unique primary keys:

- "Using composite keys" on page 106
- "Using UUIDs" on page 106
- "Using global autoincrement" on page 107
- "Using primary key pools" on page 110

Using composite keys

The MobiLink remote ID uniquely defines a remote database within a synchronization system. Therefore, an easy way to create a unique primary key is to create a composite primary key that includes the MobiLink remote ID as part of its value. If you maintain unique MobiLink user names, you could use the user name instead of the remote ID.

See "Remote IDs" [*MobiLink - Client Administration*].

Using UUIDs

You can ensure that primary keys are unique by using the newid( ) function to create universally unique values for your primary key. The resulting UUIDs can be converted to a string using the uuidtostr( ) function, and converted back to binary using the strtouuid( ) function.

UUIDs, also known as GUIDs, are unique across all computers. However, the values are completely random and so cannot be used to determine when a value was added, or the order of values. UUID values are also considerably larger than the values required by other methods (including global autoincrement), and require more table space in both the primary and foreign key tables. Indexes on tables using UUIDs are also less efficient.

See also

SQL Anywhere databases:

- "The NEWID default" [*SQL Anywhere Server - SQL Usage*]
- "NEWID function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "UNIQUEIDENTIFIER data type" [*SQL Anywhere Server - SQL Reference*]

UltraLite databases:

- "Primary key uniqueness in UltraLite" [*MobiLink - Client Administration*]
- "NEWID function [Miscellaneous]" [*UltraLite - Database Management and Reference*]

Example

The following SQL Anywhere CREATE TABLE statement creates a primary key that is universally unique:

```
CREATE TABLE customer (
   cust_key UNIQUEIDENTIFIER NOT NULL
            DEFAULT NEWID( ),
   rep_key VARCHAR(5),
   PRIMARY KEY(cust_key))
```

Using global autoincrement

In SQL Anywhere and UltraLite databases, you can set the default column value to be GLOBAL AUTOINCREMENT. You can use this default for any column in which you want to maintain unique values, but it is particularly useful for primary keys.

♦ To use global autoincrement columns

1. Declare the column as a global autoincrement column.

   When you specify default global autoincrement, the domain of values for that column is partitioned. Each partition contains the same number of values. For example, if you set the partition size for an integer column in a database to 1000, one partition extends from 1001 to 2000, the next from 2001 to 3000, and so on.

   See "Declaring default global autoincrement" on page 108.

2. Set the global_database_id value.

   SQL Anywhere supplies default values in a database only from the partition uniquely identified by that database's number. For example, if you assign a database the identity number 10 and the partition size is 1000, the default values in that database would be chosen in the range 10001–11000. Another copy of the database, assigned the identification number 11, would supply default value for the same column in the range 11001–12000.

   See "Setting the global database ID" on page 108.

Declaring default global autoincrement

You can set default values in your database by selecting the column properties in Sybase Central, or by including the DEFAULT GLOBAL AUTOINCREMENT phrase in a CREATE TABLE or ALTER TABLE statement.

Optionally, the partition size can be specified in parentheses immediately following the AUTOINCREMENT keyword. The partition size may be any positive integer, although the partition size is generally chosen so that the supply of numbers within any one partition will rarely, if ever, be exhausted.

For columns of type INT or UNSIGNED INT, the default partition size is $2^{16} = 65536$; for columns of other types the default partition size is $2^{32} = 4294967296$. Since these defaults may be inappropriate, especially if our column is not of type INT or BIGINT, it is best to specify the partition size explicitly.

For example, the following SQL statement creates a simple table with two columns: an integer that holds a customer identification number and a character string that holds the customer's name. The partition size is set to 5000.

```
CREATE TABLE customer (
    id   INT          DEFAULT GLOBAL AUTOINCREMENT (5000),
    name VARCHAR(128) NOT NULL,
    PRIMARY KEY (id)
)
```

See also

- SQL Anywhere: "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- UltraLite: "UltraLite CREATE TABLE statement" [*UltraLite - Database Management and Reference*]

Setting the global database ID

When deploying an application, you must assign a different identification number to each database. You can create and distribute the identification numbers by a variety of means. One method is to place the values in a table and download the correct row to each database based on some other unique property, such as remote ID.

♦ To set the global database identification number

- In SQL Anywhere, you set the global ID of a database by setting the value of the public option global_database_id. The identification number must be a non-negative integer. See "global_database_id option [database]" [*SQL Anywhere Server - Database Administration*].

  In UltraLite, you set the global ID of a database by setting the global_id option. See "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*].

How default values are chosen

The global database ID is set with the public option global_database_id in SQL Anywhere, and with the global_id option in UltraLite.

The global database id option in each database must be set to a unique, non-negative integer. The range of default values for a particular database is $pn + 1$ to $p(n + 1)$, where p is the partition size and n is the value

of the global database ID. For example, if the partition size is 1000 and global database ID is set to 3, then the range is from 3001 to 4000.

SQL Anywhere and UltraLite choose default values by applying the following rules:

- If the column contains no values in the current partition, the first default value is $pn + 1$, where p is the partition size and n is the value of the global database ID.
- If the column contains values in the current partition, but all are less than $p(n + 1)$, the next default value will be one greater than the previous maximum value in this range.
- Default column values are not affected by values in the column outside of the current partition; that is, by numbers less than $pn + 1$ or greater than $p(n + 1)$. Such values may be present if they have been replicated from another database via MobiLink synchronization.

If the global database ID is set to the default value of 2147483647, a null value is inserted into the column. Should null values not be permitted, the attempt to insert the row causes an error. This situation arises, for example, if the column is contained in the table's primary key.

Because the global database ID cannot be set to negative values, the values chosen are always positive. The maximum identification number is restricted only by the column data type and the partition size.

Null default values are also generated when the supply of values within the partition has been exhausted. In this case, a new global database ID value should be assigned to the database to allow default values to be chosen from another partition. Attempting to insert the null value causes an error if the column does not permit nulls. To detect that the supply of unused values is low and handle this condition, you can create an event of type GlobalAutoincrement.

Should the values in a particular partition become exhausted, you can assign a new global database ID to that database. You can assign new database ID numbers in any convenient manner. However, one possible technique is to maintain a pool of unused database ID values. This pool is maintained in the same manner as a pool of primary keys.

You can set an event handler to automatically notify the database administrator (or carry out some other action) when the partition is nearly exhausted. For SQL Anywhere databases, see "Defining trigger conditions for events" [*SQL Anywhere Server - Database Administration*].

See also

- "Setting the global database ID" on page 108
- SQL Anywhere: "global_database_id option [database]" [*SQL Anywhere Server - Database Administration*]
- UltraLite: "UltraLite global_database_id option" [*UltraLite - Database Management and Reference*]

Example

In a SQL Anywhere database, the following statement sets the database identification number to 20.

```
SET OPTION PUBLIC.global_database_id = 20
```

If the partition size for a particular column is 5000, default values for this database are selected from the range 100001–105000.

Using primary key pools

One efficient means of solving this problem of unique primary keys is to assign each user of the database a pool of primary key values that can be used as the need arises. For example, you can assign each sales representative 100 new identification values. Each sales representative can freely assign values to new customers from his or her own pool.

♦ To implement a primary key pool

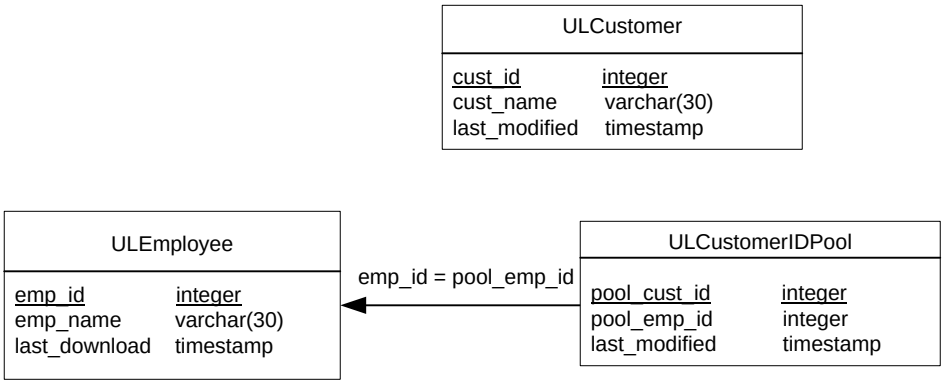
1. Add a new table to the consolidated database and to each remote database to hold the new primary key pool. Apart from a column for the unique value, these tables should contain a column for a user name, to identify who has been given the right to assign the value.
2. Write a stored procedure to ensure that each user is assigned enough new identification values. Assign more new values to remote users who insert many new entries or who synchronize infrequently.
3. Write a download_cursor script to select the new values assigned to each user and download them to the remote database.
4. Modify the application that uses the remote database so that when a user inserts a new row, the application uses one of the values from the pool. The application must then delete that value from the pool so it is not used a second time.
5. Write an upload script. The MobiLink server will then delete rows from the consolidated pool of values that a user has deleted from his personal value pool in the remote database.
6. Write an end_upload script to call the stored procedure that maintains the pool of values. Doing so has the effect of adding more values to the user's pool to replace those deleted during upload.

Example

The sample application allows remote users to add customers. It is essential that each new row has a unique primary key value, and yet each remote database is disconnected when data entry is occurring.

The ULCustomerIDPool holds a list of primary key values that can be used by each remote database. In addition, the ULCustomerIDPool_maintain stored procedure tops up the pool as values are used up. The maintenance procedures are called by a table-level end_upload script, and the pools at each remote database are maintained by upload_insert and download_cursor scripts.

1. The ULCustomerIDPool table in the consolidated database holds the pool of new customer identification numbers. It has no direct link to the ULCustomer table.



2. The ULCustomerIDPool_maintain procedure updates the ULCustomerIDPool table in the consolidated database. The following sample code is for a SQL Anywhere consolidated database.

```
CREATE PROCEDURE ULCustomerIDPool_maintain ( IN syncuser_id INTEGER )
BEGIN
    DECLARE pool_count INTEGER;

    -- Determine how may ids to add to the pool
    SELECT COUNT(*) INTO pool_count
    FROM ULCustomerIDPool
    WHERE pool_emp_id = syncuser_id;

    -- Top up the pool with new ids
    WHILE pool_count < 20 LOOP
      INSERT INTO ULCustomerIDPool ( pool_emp_id )
      VALUES ( syncuser_id );
      SET pool_count = pool_count + 1;
    END LOOP;
END
```

   This procedure counts the numbers presently assigned to the current user, and inserts new rows so that this user has a sufficient supply of customer identification numbers.

   This procedure is called at the end of the upload, by the end_upload table script for the ULCustomerIDPool table. The script is as follows:

```
CALL ULCustomerIDPool_maintain( {ml s.username} )
```

3. The download_cursor script for the ULCustomerIDPool table downloads new numbers to the remote database.

```
SELECT pool_cust_id
FROM ULCustomerIDPool
WHERE pool_emp_id = {ml s.username}
AND last_modified >= {ml s.last_table_download}
```

4. To insert a new customer, the application using the remote database must select an unused identification number from the pool, delete this number from the pool, and insert the new customer information using this identification number. The following embedded SQL function for an UltraLite application retrieves a new customer number from the pool.

```
bool CDemoDB::GetNextCustomerID( void )
/************************************/
{
```



```
        short    ind;

        EXEC SQL SELECT min( pool_cust_id )
        INTO :m_CustID:ind FROM ULCustomerIDPool;
        if( ind < 0 ) {
            return false;
        }
        EXEC SQL DELETE FROM ULCustomerIDPool
        WHERE pool_cust_id = :m_CustID;
        return true;
    }
```

Handling conflicts

Conflicts can arise during the upload of rows to the consolidated database. If two users modify the same row on different remote databases, a conflict is detected when the second of the rows arrives at the MobiLink server.

By default,

- If an attempt to insert a row finds that the row has already been inserted, an error is generated.
- If an attempt to delete a row finds that the row has already been deleted, the second attempt to delete is ignored.

If you need different behavior, you can implement it by defining one or more of the upload events that are described in this section.

About conflicts

Caution
Never update primary keys in synchronized tables. Updating primary keys defeats the purpose of a primary key because the key is the only way to identify the same row in different databases (remote and consolidated) and the only way to detect conflicts.

Conflicts are not the same as errors. When conflicts can occur, you should define a process to compute the correct values, or at least to log the conflict. Conflict handling is an integral part of a well-designed application.

During the download stage of a synchronization, no conflicts arise in the remote database. If a downloaded row contains a new primary key, the values are inserted into a new row. If the primary key matches that of a pre-existing row, the values in the row are updated.

Example

User1 starts with an inventory of ten items, and then sells three and updates the Remote1 inventory value to seven items. User2 sells four items and updates the Remote2 inventory to six. When Remote1 synchronizes, the consolidated database is updated to seven. When Remote2 synchronizes, a conflict is detected because the value of the inventory is no longer ten. To resolve this conflict programmatically, you need three row values:

1. The current value in the consolidated database.
2. The new row value that Remote2 uploaded.
3. The old row value that Remote2 obtained during the last synchronization.

In this case, the business logic could use the following to calculate the new inventory value and resolve the conflict:


```
current consolidated - (old remote - new remote)
-> 7 - (10-6) = 3
```

For other examples of how to handle conflicts, see:

- "Synchronizing products in the Contact sample" [*MobiLink - Getting Started*]

Detecting conflicts

When a MobiLink client sends an updated row to the MobiLink server, it includes not only the new updated values (the post-image), but also a copy of the old row values (the pre-image) obtained either in the last download or from the row values existing prior to the first upload of this row. When the pre-image does not match the current values in the consolidated database, a conflict is detected.

There are several scripts provided to detect conflicts. The MobiLink server detects conflicts only if one of the following scripts is applied:

- An upload_fetch or upload_fetch_column_conflict script.
- An upload_update script that includes all non-primary key columns in the WHERE clause.

Detecting conflicts with upload_fetch scripts

If you define an upload_fetch or upload_fetch_column_conflict script for a table, the MobiLink server compares the pre-image of an update to the values of the row returned by the upload_fetch script with the same primary key values. If values in the pre-image do not match the current consolidated values, the MobiLink server detects a conflict.

The upload_fetch script selects a single row of data from a consolidated database table corresponding to the row being updated. A typical upload_fetch script has the following syntax:

SELECT *pk1*, *pk2*, ...*col1*, *col2*, …
FROM *table-name*
WHERE *pk1* **= {ml r.***pk1***} AND** *pk2* **= {ml r.***pk2***}** …
  AND *col1* **= {ml r.***col1***} AND** *col2* **= {ml r.***col2***}** …

See "upload_fetch table event" on page 390.

The upload_fetch_column_conflict event is similar to upload_fetch, but it only detects a conflict when two users update the same column. Different users can update the same row, as long as they don't update the same column, without generating a conflict.

See "upload_fetch_column_conflict table event" on page 392.

You can have only one upload_fetch or upload_fetch_column_conflict script for each table in the remote database.

Locking the row on the consolidated database

It is possible that a row might change on the consolidated database after the upload_fetch script detects a conflict and before the conflict resolution is completed. To avoid this problem, which could result in incorrect data, you can implement the upload_fetch or upload_fetch_column_conflict scripts with a row lock.

In SQL Anywhere consolidated databases, you can use either the UPDLOCK or HOLDLOCK keywords, but UPDLOCK is better for concurrency. For example:

```
SELECT column-names from table-name WITH (UPDLOCK)
  WHERE where-clause
```

For Oracle and DB2, use FOR UPDATE. For example:

```
SELECT column-names FROM table-name
  WHERE where-clause
    FOR UPDATE
```

For Microsoft SQL Server, use HOLDLOCK. For example,

```
SELECT column-names FROM table-name WITH (HOLDLOCK)
  WHERE where-clause
```

For Adaptive Server Enterprise, use HOLDLOCK. For example,

```
SELECT column-names FROM table-name
  HOLDLOCK
  WHERE where-clause
```

Example

You define an upload_fetch script. The MobiLink server uses the script to retrieve the current row in the consolidated database and compares this row to the pre-image of the updated row. If the two rows contain identical values, there is no conflict. If the two rows differ, then a conflict is detected and MobiLink calls the upload_old_row_insert and upload_new_row_insert scripts, followed by resolve_conflict.

See "Resolving conflicts with resolve_conflict scripts" on page 116.

Detecting conflicts with upload_update scripts

To use the upload_update script to detect conflicts, include all columns in the WHERE clause:

UPDATE *table-name*
SET *col1* **= {ml r.***col1***},** *col2* **= {ml r.** *col2* **}** …
WHERE *pk1* **= {ml o.***pk1***}** **AND** *pk2* **= {ml o.***pk2***}** …
  AND *col1* **= {ml o.***col1***}** **AND** *col2* **= {ml o.***col2***}** …

In this statement, col1, col2 and so on are the non-primary key columns, while pk1, pk2 and so on are primary key columns. The values passed to the second set of non-primary key columns (o.) are the pre-image (or old values) of the updated row. The WHERE clause compares old values uploaded from the remote to current values in the consolidated database. If the values do not match, the update is ignored, preserving the values already on the consolidated database.

See "upload_update table event" on page 410.

The upload_update script is used for conflict detection only if no conflict is detected by upload_fetch or upload_fetch_column_conflict.

Scenario 1

You define scripts for the following events: upload_update, upload_old_row_insert, upload_new_row_insert, and resolve_conflict.

You define the following upload_update script:

```
UPDATE product
SET name={ml r.name}, description={ml r.description}
WHERE id={ml r.id}
   AND name={ml o.name}
   AND description={ml o.description]
```

MobiLink performs the update and then checks to see how many rows were modified. If no rows were modified, then MobiLink has detected a conflict: no row in the consolidated database matches the pre-image row. MobiLink calls the upload_old_row_insert and upload_new_row_insert scripts, followed by resolve_conflict.

See "Resolving conflicts with resolve_conflict scripts" on page 116.

Scenario 2

You do not define scripts for upload_old_row_insert, upload_new_row_insert, and resolve_conflict. Instead, you create a stored procedure to handle the conflict detection and resolution and you call it in the upload_update script.

See "Resolving conflicts with upload_update scripts" on page 118.

Resolving conflicts

You have several options for resolving conflicts:

- Resolve conflicts as they occur using temporary or permanent tables and a resolve_conflict script.

  See "Resolving conflicts with resolve_conflict scripts" on page 116.

- Resolve conflicts as they occur using an upload_update script.

  See "Resolving conflicts with upload_update scripts" on page 118.

- Resolve all conflicts at once using a table's end_upload script.

  See "end_upload table event" on page 327.

Resolving conflicts with resolve_conflict scripts

When the MobiLink server detects a conflict using an upload_fetch script, the following events take place.

- The MobiLink server inserts old row values uploaded from the remote database as defined by the upload_old_row_insert script. Typically, the old values are inserted into a temporary table.

  See "upload_old_row_insert table event" on page 398.

- The MobiLink server inserts the new row values uploaded from the remote database as defined by the upload_new_row_insert script. Typically, the new values are inserted into a temporary table.

  See "upload_new_row_insert table event" on page 395.

- The MobiLink server executes the resolve_conflict script. In this script you can either call a stored procedure, or define a sequence of steps to resolve the conflict using the new and old row values.

For more information, see "resolve_conflict table event" on page 373.

Example

In the following example, you create scripts for six events and then you create a stored procedure.

- In the begin_synchronization script, you create two temporary tables called contact_new and contact_old. (You could also do this in the begin_connection script.)
- The upload_fetch script detects the conflict.
- When there is a conflict, the upload_old_row_insert and upload_new_row_insert scripts populate the two temporary tables with the new and old data uploaded from the remote database.
- The resolve_conflict script calls the stored procedure MLResolveContactConflict to resolve the conflict.

Event	Script

begin_synchronization	`CREATE TABLE #contact_new(`
	`  id  INTEGER,`
	`  location CHAR(36),`
	`  contact_date DATE);`
	`CREATE TABLE #contact_old(`
	`  id  INTEGER,`
	`  location CHAR(36),`
	`  contact_date DATE)`

upload_fetch	`SELECT id, location, contact_date`
	`  FROM contact`
	`  WHERE id = {ml r.id}`

upload_old_row_insert	`INSERT INTO #contact_new( id, location, contact_date )` `  VALUES ( {ml r.id}, {ml r.location}, {ml r.contact_date} )`
upload_new_row_insert	`INSERT INTO #contact_old( id, location, contact_date )` `  VALUES ( {ml r.id}, {ml r.location}, {ml r.contact_date} )`
resolve_conflict	`CALL MLResolveContactConflict( )`

Event	Script
end_synchronization	DROP TABLE #contact_new; DROP TABLE #contact_old

The stored procedure MLResolveContactConflict is as follows:

```
CREATE PROCEDURE MLResolveContactConflict( )
BEGIN
  --update the consolidated database only if the new contact date
  --is later than the existing contact date
  UPDATE contact c
    SET c.contact_date = cn.contact_date
      FROM #contact_new cn
      WHERE c.id = cn.id
        AND cn.contact_date > c.contact_date;
  --cleanup
  DELETE FROM #contact_new;
  DELETE FROM #contact_old;
END
```

Resolving conflicts with upload_update scripts

Instead of using the resolve_conflict script for conflict resolution, you can call a stored procedure in the upload_update script. With this technique, you must both detect and resolve conflicts programmatically.

The stored procedure must use the format of the upload_update script with a WHERE clause that includes all columns but uses the pre-image (old) values.

The upload_update script could be as follows:

```
{CALL UpdateProduct(
   {ml o.id}, {ml o.name}, {ml o.desc}, {ml r.name}, {ml r.desc}
)
}
```

The UpdateProduct stored procedure could be:

```
CREATE PROCEDURE UpdateProduct(
  @id INTEGER,
  @preName VARCHAR(20),
  @preDesc VARCHAR(200),
  @postName VARCHAR(20),
  @postDesc VARCHAR(200) )
BEGIN
    UPDATE product
     SET name = @postName, description = @postDesc
     WHERE id = @id
       AND name = @preName
       AND description = @preDesc
    IF @@rowcount=0 THEN
        // A conflict occurred: handle resolution here.
    END IF
END
```

This approach is often easier to maintain than the approach described in “Resolving conflicts with resolve_conflict scripts” on page 116 because there is only one script to maintain and all the logic is contained in one stored procedure. However, the code of the stored procedure may be complicated if the tables columns

are nullable or if they contain BLOBs or CLOBs. Also, some RDBMSs that are supported MobiLink consolidated databases have limitations on the size of values that can be passed to stored procedures.

See:

- "Detecting conflicts with upload_update scripts" on page 115
- "upload_update table event" on page 410

Example

The following stored procedure, sp_update_my_customer, contains logic for conflict detection and resolution. It accepts old column values and new column values. This example uses SQL Anywhere features. The script could be implemented as follows.

```
{CALL sp_update_my_customer(
   {ml o.cust_1st_pk},
   {ml o.cust_2nd_pk},
   {ml o.first_name},
   {ml o.last_name},
   {ml o.nullable_col},
   {ml o.last_modified},
   {ml r.first_name},
   {ml r.last_name},
   {ml r.nullable_col},
   {ml r.last_modified}
)}

CREATE PROCEDURE sp_update_my_customer(
     @cust_1st_pk        INTEGER,
     @cust_2nd_pk        INTEGER,
     @old_first_name     VARCHAR(100),
     @old_last_name      VARCHAR(100),
     @old_nullable_col   VARCHAR(20),
     @old_last_modified  DATETIME,
     @new_first_name     VARCHAR(100),
     @new_last_name      VARCHAR(100),
     @new_nullable_col   VARCHAR(20),
     @new_last_modified  DATETIME
)

BEGIN
DECLARE @current_last_modified DATETIME;
// Detect a conflict by checking the number of rows that are
// affected by the following update. The WHERE clause compares
// old values uploaded from the remote to current values in
// the consolidated database. If the values match, there is
// no conflict. The COALESCE function returns the first non-
// NULL expression from a list, and is used in this case to
// compare values for a nullable column.


UPDATE my_customer
SET first_name                  = @new_first_name,
 last_name                        = @new_last_name,
 nullable_col                   = @new_nullable_col,
 last_modified                  = @new_last_modified

WHERE cust_1st_pk               = @cust_1st_pk
 AND cust_2nd_pk                = @cust_2nd_pk
 AND first_name                 = @old_first_name
 AND last_name                  = @old_last_name
```

```
 AND COALESCE(nullable_col, '') = COALESCE(@old_nullable_col, '')
 AND last_modified              = @old_last_modified;
...

// Use the @@rowcount global variable to determine
// the number of rows affected by the update. If @@rowcount=0,
// a conflict has occurred. In this example, the database with
// the most recent update wins the conflict. If the consolidated
// database wins the conflict, it retains its current values
// and no action is taken.

IF( @@rowcount = 0 ) THEN
// A conflict has been detected. To resolve it, use business
// logic to determine which values to use, and update the
// consolidated database with the final values.

  SELECT last_modified INTO @current_last_modified
     FROM my_customer WITH( HOLDLOCK )
     WHERE cust_1st_pk=@cust_1st_pd
        AND cust_2nd_pk=@cust_2nd_pk;


     IF( @new_last_modified > @current_last_modified ) THEN
     // The remote has won the conflict: use the values it
     // uploaded.

       UPDATE my_customer
       SET first_name    = @new_first_name,
         last_name     = @new_last_name,
         nullable_col  = @new_nullable_col,
         last_modified = @new_last_modified
       WHERE cust_1st_pk  = @cust_1st_pk
          AND cust_2nd_pk   = @cust_2nd_pk;

     END IF;
END IF;
END;
```

See:

- “COALESCE function [Miscellaneous]” [*SQL Anywhere Server - SQL Reference*]
- @@rowcount in “Global variables” [*SQL Anywhere Server - SQL Reference*]

Forced conflicts

Forced conflict resolution is a special technique that forces every uploaded row to be treated as if it were a conflict.

The MobiLink server uses forced conflict resolution when the upload_insert, upload_update, and upload_delete script are all undefined. In this mode of operation, the MobiLink server attempts to insert all uploaded rows from that table using the statements defined by the upload_old_row_insert and upload_new_row_insert scripts. In essence, all uploaded rows are then treated as conflicts. You can write stored procedures or scripts to process the uploaded values in any way you want.

Without any of the upload_insert, upload_update, or upload_delete scripts, the normal conflict-resolution procedure is bypassed. This technique has two principal uses.

- **Arbitrary conflict detection and resolution** The automatic mechanism only detects errors when updating a row, and only then when the old values do not match the present values in the consolidated database.

  You can capture the raw uploaded data using the upload_old_row_insert and upload_new_row_insert scripts, then process the rows as you see fit.

- **Performance** When the upload_insert, upload_update, and upload_delete are not defined, the MobiLink server is relieved of its normal conflict-detection tasks, which involve querying the consolidated database one row at a time. Instead, it needs only to insert the raw uploaded information using the statements defined by the upload_old_row_insert and upload_new_row_insert scripts. Performance is improved because the MobiLink server is not fetching rows across the network.

See also

- "Forced conflict statistics" on page 163

Data entry

In some databases, there are tables that are only used for data entry. One way of processing these tables is to upload all inserted rows at each synchronization, and remove them from the remote database on the download. After synchronization, the remote table is empty again, ready for another batch of data.

To achieve this model, you can upload rows into a temporary table and then insert them into a base table using an end_upload table script. The temporary table can be used in the download_delete_cursor to remove rows from the remote database following a successful synchronization.

Alternatively, you can allow the client application to the delete the rows, using the STOP SYNCHRONIZATION DELETE statement to stop the deletes being uploaded during the next synchronization.

See "STOP SYNCHRONIZATION DELETE statement [MobiLink]" [*SQL Anywhere Server - SQL Reference*].

Handling deletes

When rows are deleted from the consolidated database, there needs to be a record of the row so it can be removed from any remote databases that have the row. Two ways to do this are by using logical deletes or shadow tables.

- **Logical deletes** With this method, the row is not deleted. Data that is no longer required is marked as inactive in a status column. The WHERE clause of the download_cursor and download_delete_cursor can refer to the status of the row.

  This technique is used in the ULEmpCust table in the CustDB sample application, in which the action column holds a D for Delete. The scripts use this value to delete the record from the remote database, and delete the record from the consolidated database at the end of the synchronization. CustDB also uses this technique for the ULOrder table, and the Contact sample uses the technique on the Customer, Contact, and Product tables.

- **Shadow tables** With this method, you create a shadow table that stores the primary key values of deleted rows. When a row is deleted, a trigger can populate the shadow table. The download_delete_cursor can use the shadow table to remove rows from remote databases. The shadow table only needs to contain the primary key columns from the real table.

See "Writing download_delete_cursor scripts" on page 234.

Temporarily stopping the synchronization of deletes

Ordinarily, SQL Anywhere automatically logs any changes to tables or columns that are part of a publication with a synchronization subscription. These changes are uploaded to the consolidated database during the next synchronization.

You may, however, want to delete rows from synchronized data and not have those changes uploaded. You can do this using STOP SYNCHRONIZATION DELETE. This feature can be used to make unusual corrections, but should be used with caution as it effectively disables part of the automatic synchronization functionality. This technique is a practical alternative to deleting the necessary rows using a download_delete_cursor script

When a STOP SYNCHRONIZATION DELETE statement is executed, none of the delete operations subsequently executed on that connection are synchronized. The effect continues until a START SYNCHRONIZATION DELETE statement is executed. The effects do not nest; that is, subsequent executions of STOP SYNCHRONIZATION DELETE after the first will have no additional effect.

♦ To temporarily disable upload of deletes made through a connection

1. Issue the following statement to stop automatic logging of deletes.

   ```
   STOP SYNCHRONIZATION DELETE
   ```

2. Delete rows from the synchronized data, as required, using the DELETE statement. Commit these changes.

3. Restart logging of deletes using the following statement.

```
START SYNCHRONIZATION DELETE
```

The deleted rows will not be sent up to the MobiLink server and hence will not be deleted from the consolidated database.

See also

- SQL Anywhere clients: "STOP SYNCHRONIZATION DELETE statement [MobiLink]" [*SQL Anywhere Server - SQL Reference*]
- UltraLite clients: "UltraLite STOP SYNCHRONIZATION DELETE statement" [*UltraLite - Database Management and Reference*]

Handling failed downloads

Bookkeeping information about what is downloaded must be maintained in the download transaction. This information is updated atomically with the download being applied to the remote database.

If a failure occurs before the entire download is applied to the remote database, and if you change SendDownloadAck to ON, then the MobiLink server does not get confirmation for the download and rolls back the download transaction. Since the bookkeeping information is part of the download transaction, it is also rolled back. Next time the download is built, it will use the original bookkeeping information.

See "SendDownloadACK (sa) extended option" [*MobiLink - Client Administration*] and "Send Download Acknowledgment synchronization parameter" [*MobiLink - Client Administration*].

When testing your synchronization scripts, you should add logic to your end_download script that causes occasional failures. This will ensure that your scripts can handle a failed download.

Resuming failed downloads

Download failure is caused by a communication error during download or a user aborting the download. MobiLink has functionality that helps you recover from download failure, and may help you avoid having to retransmit the entire download. This functionality has separate implementations for SQL Anywhere and UltraLite remote databases.

SQL Anywhere remote databases

When synchronization fails during a download, none of the download is applied to the remote database. However, the part of the download that was successfully transmitted is stored in a temporary file on the remote device. You cannot access this file directly, but dbmlsync provides functionality that makes use of the file. When you use this functionality, you may be able to avoid lengthy retransmission of data. You may also be able to recover from download failure automatically.

Note
The download cannot be resumed when the SendDownloadACK extended option is set to ON (the default is OFF) or when the DownloadBufferSize extended option is set to 0 (which is also not the default).

There are three ways to implement this functionality. In all cases, the resumed download will fail if there is any new data to be uploaded, and dbmlsync will abort.

- **-dc** After a download fails, use -dc the next time you start dbmlsync to resume the download. If part of the failed download was transmitted, the MobiLink server will only transmit the remainder of the download.

  For more information, see "-dc option" [*MobiLink - Client Administration*].

- **ContinueDownload (cd) extended option** When used on the dbmlsync command line, the cd extended option works just like the -dc option. You can also store this option in the database, or use sp_hook_dbmlsync_set_extended_options to set this option in a single synchronization.

See "ContinueDownload (cd) extended option" [*MobiLink - Client Administration*] and "sp_hook_dbmlsync_set_extended_options" [*MobiLink - Client Administration*].

- **sp_hook_dbmlsync_end hook** You can use the **restart** parameter to cause a download to resume. You know a download is resumable if the **restartable download** parameter is set to true. You can also create logic in the hook to resume a download if a download file exists and is a certain size.

  See "sp_hook_dbmlsync_end" [*MobiLink - Client Administration*].

UltraLite remote databases

You can control the behavior of UltraLite applications following a failed download as follows:

- If you set the Keep Partial Download synchronization parameter to true when you synchronize, and the download fails before completion, then UltraLite applies that portion of the changes that were downloaded. UltraLite also sets the Partial Download Retained synchronization parameter to true.

  The UltraLite database may be in an inconsistent state at this point. Depending on your application, you may want to ensure that synchronization completes successfully or is rolled back before you allow changes to the data.

  See "Keep Partial Download synchronization parameter" [*MobiLink - Client Administration*] and "Partial Download Retained synchronization parameter" [*MobiLink - Client Administration*].

- To resume the download, set the Resume Partial Download synchronization parameter to true and synchronize again.

  See "Resume Partial Download synchronization parameter" [*MobiLink - Client Administration*].

  The restarted synchronization does not carry out an upload, and downloads only those changes that would have been downloaded by the failed download. That is, it completes the failed download, but does not synchronize changes made since the previous attempt. To get those changes, you would need to synchronize again once the failed download has completed, or and call Rollback Partial Download and synchronize with Resume Partial Download set to false.

  When you restart the download, many of the synchronization parameters from the failed synchronization are used again automatically. For example, the publications parameter is ignored: the synchronization downloads those publications requested on the initial download. The only parameters that must be set are the Resume Partial Download parameter (which must be set to true) and the User Name parameter. In addition, settings for the following parameters are obeyed, if set:

  - Keep Partial Download (in case of further interruption)
  - DisableConcurrency
  - Observer
  - User Data

- To roll back the changes from the failed download without resuming synchronization, call the function to roll back the changes. This function is ULRollbackPartialDownload function for embedded SQL. For UltraLite components, it is a method on the Connection object.

- **MobileVB** "RollbackPartialDownload method" [*UltraLite - AppForge Programming*].
- **UltraLite.NET** "RollbackPartialDownload method" [*UltraLite - .NET Programming*]
- **Embedded SQL** "ULRollbackPartialDownload function" [*UltraLite - C and C++ Programming*].

You may want to roll back the changes from a failed download if synchronization cannot be completed (perhaps the server or network is unavailable) and if you want to maintain a consistent set of data while letting the end user carry on working on the application.

For more information about communications errors, see SQL Anywhere 10 - Error Messages [*SQL Anywhere 10 - Error Messages*].

Note
If the send_download_ack synchronization parameter is set to true (which is not the default), the setting will be ignored for the resumed download.

Downloading a result set from a stored procedure call

You can download a result set from a stored procedure call. For example, you might currently have a download_cursor for the following table:

```
CREATE TABLE MyTable (
    pk INTEGER PRIMARY KEY NOT NULL,
    col1 VARCHAR(100) NOT NULL,
    col2 VARCHAR(20) NOT NULL
)
```

The download_cursor table script might look as follows:

```
SELECT pk, col1, col2
   FROM MyTable
      WHERE last_modified >= {ml s.last_table_download}
      AND employee = {ml s.username}
```

If you want your downloads to MyTable to use more sophisticated business logic, you can now create your script as follows, where DownloadMyTable is a stored procedure taking two parameters (last-download timestamp and MobiLink user name) and returning a result set. (This example uses an ODBC calling convention for portability):

```
{call DownloadMyTable( {ml s.last_table_download}, {ml s.username} )}
```

Following are some simple examples for each supported consolidated database. Consult the documentation for your consolidated database for full details.

The following example works with SQL Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server.

```
CREATE PROCEDURE SPDownload
    @last_dl_ts DATETIME,
    @u_name VARCHAR( 128 )
AS
BEGIN
     SELECT pk, col1, col2
       FROM MyTable
          WHERE last_modified >= @last_dl_ts
          AND employee = @u_name
END
```

The following example works with Oracle. Oracle requires that a package be defined. This package must contain a record type for the result set, and a cursor type that returns the record type.

```
Create or replace package SPInfo as
Type SPRec is record (
    pk     integer,
    col1   varchar(100),
    col2   varchar(20)
);
Type SPCursor is ref cursor return SPRec;
End SPInfo;
```

Next, Oracle requires a stored procedure with the cursor type as the first parameter. Note that the download_cursor script only passes in two parameters, not three. For stored procedures returning result sets in Oracle, cursor types declared as parameters in the stored procedure definition define the structure of the result set, but do not define a true parameter as such. In this example, the stored procedure also adds the script to the MobiLink system table.


```
Create or replace procedure
    DownloadMyTable( v_spcursor IN OUT SPInfo.SPCursor,
                     v_last_dl_ts IN DATE,
                     v_user_name IN VARCHAR ) As
Begin
    Open v_spcursor For
       select pk, col1, col2
         from MyTable
           where last_modified >= v_last_dl_ts
           and employee = v_user_name;
End;

CALL ml_add_table_script(
  'v1',
  'MyTable',
  'download_cursor',
  '{CALL DownloadMyTable(
     {ml s.last_table_download},{ml s.username} )}'
);
```

The following example works with IBM DB2 UDB.

```
CREATE PROCEDURE DownloadMyTable(
   IN last_dl_ts TIMESTAMP,
   IN u_name VARCHAR( 128 ) )
        EXTERNAL NAME 'DLMyTable!DownloadMyTable'
        RESULT SETS 1
        FENCED
        LANGUAGE JAVA PARAMETER STYLE DB2GENERAL
```

The following example is a Java implementation of the stored procedure, in DLMyTable.java. To return a result set, you must leave the result set open when the method returns:

```
import COM.ibm.db2.app.*;
import java.sql.*;

public class DLMyTable extends StoredProc
{
  public void DownloadMyTable(
      Date last_dl_ts,
      String u_name ) throws Exception
  {
      Connection  conn = getConnection();
      conn.setAutoCommit( false );
      Statement   s    = conn.createStatement();
      // Execute the select and leave it open.
      ResultSet   r    = s.executeQuery(
            "select pk, col1, col2 from MyTable"
                  + " where last_modified >= '"
                  + last_dl_ts
                  + "' and employee = '"
                  + u_name + "'" );
    }
}
```


Uploading data from self-referencing tables

Some tables are self-referencing. For example, an employee table may contain a column that lists employees and a column that lists the manager of each employee, and there may be a hierarchy of managers managing managers. These tables can pose a challenge to synchronization because the MobiLink default behavior is to coalesce all data updates on the remote database, which is efficient but which loses the order of transactions.

There are two techniques for handling this situation:

- If you are using a SQL Anywhere remote database, you can use the dbmlsync -tu option to specify that each transaction on the remote should be sent as a separate transaction.

  See "-tu option" [*MobiLink - Client Administration*].

- Add a mapping table so the order of transactions doesn't matter.

MobiLink isolation levels

MobiLink connects to a consolidated database at the most optimal isolation level it can, given the isolation levels enabled on the RDBMS. The default isolation levels are chosen to provide the best performance while ensuring data consistency.

In general, MobiLink uses the isolation level SQL_TXN_READ_COMMITTED for uploads, and if possible, it uses snapshot isolation for downloads (if that is not possible, it uses SQL_TXN_READ_COMMITTED). Snapshot isolation eliminates the problem of downloads being blocked until transactions are closed on the consolidated database.

Snapshot isolation can result in duplicate data being downloaded (if, for example, a long-running transaction causes the same snapshot to be used for a long time), but MobiLink clients automatically handle this, so the only penalty is transmission time and the processing effort at the remote.

Isolation level 0 (READ COMMITTED) is generally unsuitable for synchronization and can lead to inconsistent data.

The isolation level is set immediately after a connection to the database occurs. Some other connection setup occurs, and then the transaction is committed. The COMMIT is required by most RDBMSs so that the isolation level (and perhaps other settings) can take effect.

SQL Anywhere version 10

SQL Anywhere version 10 supports snapshot isolation. By default, MobiLink uses the SQL_TXN_READ_COMMITTED isolation level for uploads, and snapshot isolation for downloads.

MobiLink can only use snapshot isolation if you enable it in your SQL Anywhere consolidated database. If snapshot isolation is not enabled, MobiLink uses the default SQL_TXN_READ_COMMITTED.

Enabling a database to use snapshot isolation can affect performance because copies of all modified rows must be maintained, regardless of the number of transactions that use snapshot isolation. See “Enabling snapshot isolation” [*SQL Anywhere Server - SQL Usage*].

You can enable snapshot isolation for upload with the mlsrv10 -esu option, and disable snapshot isolation with the mlsrv10 -dsd option. If you need to change the MobiLink default isolation level in a connection script, you should do so in the begin_upload or begin_download scripts. If you change the default isolation level in the begin_connection script, your setting may be overridden at the start of the upload and download transactions.

See “-esu option” on page 47 and “-dsd option” on page 44.

SQL Anywhere prior to version 10

If you are using a version of SQL Anywhere prior to version 10, the default MobiLink isolation level is SQL_TXN_READ_COMMITTED. You can change the default for the entire MobiLink session in the begin_connection script, or change it for the upload and download in the begin_upload and begin_download scripts, respectively.

Adaptive Server Enterprise

For Adaptive Server Enterprise, the default MobiLink isolation level is SQL_TXN_READ_COMMITTED. You can change the default for the entire MobiLink session in the begin_connection script, or change it for the upload and download in the begin_upload and begin_download scripts, respectively.

Oracle

Oracle supports snapshot isolation, but calls it READ COMMITTED. By default, MobiLink uses the snapshot/READ COMMITTED isolation level for upload and download.

You can change the default for the entire MobiLink session in the begin_connection script, or change it for the upload and download in the begin_upload and begin_download scripts, respectively.

For the MobiLink server to be able to make the most effective use of snapshot isolation, the Oracle account used by the MobiLink server must have permission for the V_$TRANSACTION Oracle system view. If it does not, a warning is issued and rows may be missed on download. Only SYS can grant this access. The Oracle syntax for granting this access is:

```
grant select on SYS.V_$TRANSACTION to user-name
```

Microsoft SQL Server 2005 and up

Microsoft SQL Server 2005 supports snapshot isolation. By default, MobiLink uses the SQL_TXN_READ_COMMITTED isolation level for uploads, and snapshot isolation for download.

MobiLink can only use snapshot isolation if you enable it in your SQL Server consolidated database. If snapshot is not enabled, MobiLink uses the default SQL_TXN_READ_COMMITTED. See your SQL Server documentation for details.

You can enable snapshot isolation for upload with the mlsrv10 -esu option, and disable snapshot isolation with the mlsrv10 -dsd option. If you need to change the MobiLink default isolation level in a connection script, you should do so in the begin_upload or begin_download scripts. If you change the default isolation level in the begin_connection script, your setting may be overridden at the start of the upload and download transactions.

See “-esu option” on page 47 and “-dsd option” on page 44.

To use snapshot isolation on SQL Server, the user ID that you use to connect the MobiLink server to the database must have permission to access the SQL Server system table SYS.DM_TRAN_ACTIVE_TRANSACTIONS. If this permission is not granted, MobiLink uses the default level SQL_TXN_READ_COMMITTED.

If your consolidated database is running on a Microsoft SQL Server that is also running other databases, if you are using snapshot isolation for uploads or downloads, and if your upload or download scripts do not access any other databases on the server, you should specify the MobiLink server -dt option. This option makes MobiLink ignore all transactions except ones within the current database. It increases throughput and reduces duplication of rows that are downloaded.

See “-dt option” on page 45.

Microsoft SQL Server prior to version 2005

If you are using a version of Microsoft SQL Server prior to version 2005, the default MobiLink isolation level is SQL_TXN_READ_COMMITTED. You can change the default for the entire MobiLink session in

the begin_connection script, or change it for the upload and download in the begin_upload and begin_download scripts, respectively.

See also

- "-dsd option" on page 44
- "-esu option" on page 47
- "-dt option" on page 45
- "The synchronization process" [*MobiLink - Getting Started*]
- "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*]
- "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*]

CHAPTER 5

MobiLink Performance

Contents

Performance tips

Following are some suggestions to help you get the best performance out of MobiLink.

Test

The following all contribute to the throughput of your synchronization system: the type of device running your remote databases, the schema of remote databases, the data volume and synchronization frequency of your remotes, network characteristics (including for HTTP, proxies, web servers, and Redirectors), the hardware where the MobiLink server runs, your synchronization scripts, the concurrent volume of synchronizations, the type of consolidated database you use, the hardware where your consolidated database runs, and the schema of your consolidated database.

Testing is extremely important. Before deploying, you should perform testing using the same hardware and network that you plan to use for production. You should also try to test with the same number of remotes, the same frequency of synchronization, and the same data volume. During this testing you should experiment with the following performance tips.

Avoid contention

Avoid contention and maximize concurrency in your synchronization scripts.

For example, suppose a begin_download script increments a column in a table to count the total number of downloads. If multiple users synchronize at the same time, this script would effectively serialize their downloads. The same counter would be better in the begin_synchronization, end_synchronization, or prepare_for_download scripts because these scripts are called just before a commit so any database locks are held for only a short time.

See “Contention” on page 140.

For information on the transaction structure of synchronization, see “Transactions in the synchronization process” [*MobiLink - Getting Started*].

Use an optimal number of database worker threads

Use the MobiLink –w option to set the number of MobiLink database worker threads to the smallest number that gives you optimum throughput. You will need to experiment to find the best number for your situation.

A larger number of database worker threads can improve throughput by allowing more synchronizations to access the consolidated database at the same time.

Keeping the number of database worker threads small reduces the chance of contention in the consolidated database, the number of connections to the consolidated database, and the memory required for optimal caching.

See:

- “Number of database worker threads” on page 141
- “-w option” on page 75
- “-wu option” on page 76

Enable the client-side download buffer for SQL Anywhere clients

For SQL Anywhere clients, a download buffer allows a MobiLink worker thread to transmit the download without waiting for the client to apply the download. The download buffer is enabled by default. However, the download buffer cannot be used if download acknowledgement is enabled (see next bullet).

See the "DownloadBufferSize (dbs) extended option" [*MobiLink - Client Administration*].

Do not enable download acknowledgement

By default, download acknowledgement is not enabled. This frees up MobiLink database worker threads that otherwise would be waiting for confirmation of successful download from the client, which also frees up the connection that the database worker thread is using. It also makes it possible for the MobiLink server to buffer the downloads.

See the "SendDownloadACK (sa) extended option" [*MobiLink - Client Administration*].

Avoid synchronizing unnecessary BLOBs

It is inefficient to include a BLOB in a row that is synchronized frequently. To avoid this, you can create a table that contains BLOBs and a BLOB ID, and reference the ID in the table that needs to be synchronized.

Set maximum number of database connections

Set the maximum number of MobiLink database connections to be your number of synchronization script versions times the number of MobiLink worker threads, plus one. This reduces the need for MobiLink to close and create database connections. You set the maximum number of connections with the mlsrv10 -cn option.

See "MobiLink database connections" on page 142 and "-cn option" on page 39.

Have sufficient physical memory

Ensure that the computer running the MobiLink server has enough physical memory to accommodate the cache in addition to its other memory requirements.

The number of synchronizations being actively processed is not limited by the number of database worker threads. The MobiLink server can unpack uploads and send downloads for a large number of synchronizations simultaneously. For best performance, it is very important that the MobiLink server has a large enough memory cache to process these synchronizations without paging to disk. Use the -cm option to set the memory cache for the MobiLink server.

See "-cm option" on page 38.

Use sufficient processing power

You should dedicate enough processing power to MobiLink so that the MobiLink server processing is not a bottleneck. Typically the MobiLink server requires significantly less CPU than the consolidated database. However, using Java or .NET row handling adds to the MobiLink server processing requirement. In practice, network limitations or database contention are more likely to be bottlenecks.

Use minimum logging verbosity

Use the minimum logging verbosity that is compatible with your business needs. By default, verbose logging is off, and MobiLink does not write its log to disk. You can control logging verbosity with the -v option, and enable logging to a file with the -o or -ot options.

As an alternative to verbose log files, you can monitor your synchronizations with the MobiLink Monitor. The Monitor does not need to be on the same computer as the MobiLink server, and a Monitor connection has a negligible effect on MobiLink server performance. See "MobiLink Monitor" on page 145.

Plan for operating system limitations

Operating systems restrict the number of concurrent connections a server can support over TCP/IP. If this limit is reached, which may occur when over 1000 clients attempt to synchronize at the same time, the operating system may exhibit unexpected behavior, such as unexpectedly closing connections and rejecting additional clients that attempt to connect. To prevent this behavior, use the -sm option to specify a maximum number of remote connections that is less than the operating system limit.

When a client attempts to synchronize with a MobiLink server that has accepted its maximum number of concurrent synchronizations as specified by the -sm option, the client receives the error code -85 (SQLE_COMMUNICATIONS_ERROR). The client application should handle this error and try to connect again in a few minutes.

For more information about the -sm option, see "-sm option" on page 68.

For more information about SQLE_COMMUNICATIONS_ERROR, see "Communication error" [*SQL Anywhere 10 - Error Messages*].

Java or .NET vs. SQL synchronization logic

No significant throughput difference has been found between using Java or .NET synchronization logic vs. SQL synchronization logic. However, Java and .NET synchronization logic have some extra overhead per synchronization and require more memory.

In addition, SQL synchronization logic is executed on the computer that runs the consolidated database, while Java or .NET synchronization logic is executed on the computer that runs the MobiLink server. Thus, Java or .NET synchronization logic may be desirable if your consolidated database is heavily loaded.

Synchronization using direct row handling imposes a heavier processing burden on the MobiLink server, so you may need more RAM, perhaps more disk space, and perhaps more CPU power, depending on how you implement direct row handling.

Priority synchronization

If you have some tables that you need to synchronize more frequently than others, create a separate publication and subscription for them. When using synchronization models in Sybase Central, you can do this by creating more than one model. You can synchronize this priority publication more frequently than other publications, and synchronize other publications at off-peak times.

Download only the rows you need

Take care to download only the rows that are required, for example by using timestamp synchronization instead of snapshot. Downloading unneeded rows is wasteful and adversely affects synchronization performance.

Optimize script execution

The performance of your scripts in the consolidated database is an important factor. It may help to create indexes on your tables so that the upload and download cursor scripts can efficiently locate the required rows. However, too many indexes may slow uploads.

When you use the Create Synchronization Model wizard in Sybase Central to create your MobiLink applications, an index is automatically defined for each download cursor when you deploy the model.

For large uploads, estimate the number of rows

For SQL Anywhere clients, you can significantly improve the speed of uploading a large number of rows by providing dbmlsync with an estimate of the number of rows that will be uploaded. You do this with the dbmlsync -urc option.

See "-urc option" [*MobiLink - Client Administration*].

Key factors influencing MobiLink performance

The overall performance of any system, including throughput for MobiLink synchronization, is usually limited by a bottleneck at one point in the system. For MobiLink synchronization, the following might be the bottlenecks limiting synchronization throughput:

- **The performance of the consolidated database** Of particular importance for MobiLink is the speed at which the consolidated database can execute the MobiLink scripts. Multiple database worker threads might execute scripts simultaneously, so for best throughput you need to avoid database contention in your synchronization scripts.
- **The number of MobiLink database worker threads** A smaller number of threads will involve fewer database connections, less chance of contention in the consolidated database and less operating system overhead. However, too small a number may leave clients waiting for a free database worker thread, or have fewer connections to the consolidated database than it can overlap efficiently.
- **The bandwidth for client-to-MobiLink communications** For slow connections, such as those over dial-up or wide-area wireless networks, the network may cause clients and MobiLink servers to wait for data to be transferred.
- **The client processing speed** Slow client processing speed is more likely to be a bottleneck in downloads than uploads, since downloads involve more client processing as rows and indexes are written.
- **The bandwidth for MobiLink to consolidated communication** This is unlikely to be a bottleneck if both MobiLink and the consolidated database are running on the same computer, or if they are on separate computers connected by a high-speed network.
- **The speed of the computer running the MobiLink server** If the processing power of the computer running MobiLink is slow, or if it does not have sufficient memory for the MobiLink worker threads and buffers, then MobiLink execution speed could be a synchronization bottleneck. The MobiLink server's performance depends little on disk speed as long as the buffers and worker threads fit in physical memory.

Tuning MobiLink for performance

The key to achieving optimal MobiLink synchronization throughput is to have multiple synchronizations occurring simultaneously and executing efficiently. To enable multiple simultaneous synchronizations, MobiLink uses pools of worker threads for different tasks. One pool is dedicated to reading upload data from the network and unpacking it. Another pool of threads, called **database worker threads**, applies the upload to the consolidated database and fetches data to be downloaded from the consolidated database. Another pool of worker threads is dedicated to packing and sending the download data to the remote databases. Each database worker thread uses a single connection to the consolidated database for applying and fetching changes, using your synchronization scripts.

Contention

The most important factor is to avoid database contention in your synchronization scripts. Just as with any other multi-client use of a database, you want to minimize database contention when clients are simultaneously accessing a database. Database rows that must be modified by each synchronization can

increase contention. For example, if your scripts increment a counter in a row, then updating that counter can be a bottleneck.

Synchronization requests are accepted (up to the limit specified by the -sm option) and the uploaded data is read and unpacked so that it is ready for a database worker thread. If there are more synchronizations than database worker threads, the excess are queued, waiting for a free database worker thread.

You can control the number of database worker threads and connections, but MobiLink will always ensure that there is at least one connection per database worker thread. If there are more connections than database worker threads, the excess connections will be idle. Excess connections may be useful with multiple script versions, as discussed below.

Number of database worker threads

Other than contention in your synchronization scripts, the most important factor for synchronization throughput is the number of database worker threads. The number of database worker threads controls how many synchronizations can proceed simultaneously in the consolidated database.

Testing is vital to determine the optimum number of database worker threads.

Increasing the number of database worker threads allows more overlapping synchronizations to access the consolidated database, as well as increased throughput, but it also increases resource and database contention between the overlapping synchronizations, and potentially increases the time for individual synchronizations. As the number of database worker threads is increased, the benefit of more simultaneous synchronizations becomes outweighed by the cost of longer individual synchronizations, and adding more worker threads decreases throughput. Experimentation is required to determine the optimal number of database worker threads for your situation, but the following may help to guide you.

For uploads, performance testing shows that the best throughput happens with a relatively small number of database worker threads: in most cases, three to ten worker threads. Variation depends on factors like the type of consolidated database, data volume, database schema, the complexity of the synchronization scripts, and the hardware used. The bottleneck is usually due to contention between database worker threads executing the SQL of your upload scripts at the same time in the consolidated database.

For downloads, when download acknowledgement is used the optimum number of worker threads depends on the client-to-MobiLink bandwidth and the processing speed of clients. For slower clients, more worker threads are needed to get optimal download performance. This is because downloads involve more client processing and less consolidated database processing than uploads. When download acknowledgement is used, database worker threads block until the remote database finishes applying the download.

When download acknowledgements are not used (the default), the client-to-MobiLink bandwidth is less influential because a database worker thread is free to process other synchronizations while other threads send the download. Thus the number of database worker threads is less critical.

Many downloads can be sent concurrently—far more than the number of database worker threads. For optimal download performance, it is important for the MobiLink server to have enough RAM to buffer these downloads. Otherwise the download will be paged to disk and download performance may degrade. To specify the MobiLink server memory cache size, use the -cm option.

See "-cm option" on page 38.

If the MobiLink server starts paging to disk (possibly because of too many downloads being processed concurrently), consider using the -sm option to either decrease the number of database worker threads or limit the total number of synchronizations being actively processed.

See "-sm option" on page 68.

Leaving download acknowledgement off (the default) can reduce the optimal number of database worker threads for download, because database worker threads do not have to wait for clients to apply downloads.

See the "SendDownloadACK (sa) extended option" [*MobiLink - Client Administration*].

To get both the best download throughput and the best upload throughput, MobiLink provides two options. You can specify a total number of database worker threads to optimize downloads. You can also limit the number that can simultaneously apply uploads to optimize upload throughput.

The -w option controls the total number of database worker threads. The default is five.

The -wu option limits the number of database worker threads that can simultaneously apply uploads to the consolidated database. By default, all database worker threads can apply uploads simultaneously, but that can cause severe contention in the consolidated database. The -wu option lets you reduce that contention while still having a larger number of database worker threads to optimize the fetching of download data. The -wu option only has an effect if the number is less than the total number of database worker threads.

See "-w option" on page 75 and "-wu option" on page 76.

MobiLink database connections

MobiLink creates a database connection for each database worker thread. You can use the -cn option to specify that MobiLink create a larger pool of database connections, but any excess connections will be idle unless MobiLink needs to close a connection or use a different script version.

There are two cases where MobiLink will close a database connection and open a new one. The first case is if an error occurs. The second case is if the client requests a synchronization script version, and none of the available connections have already used that synchronization version.

Note
Each database connection is associated with a script version. To change the version, the connection must be closed and reopened.

If you routinely use more than one script version, you can reduce the need for MobiLink to close and open connections by increasing the number of connections. You can eliminate the need completely if the number of connections used for synchronizations is the number of worker threads times the number of script versions.

An example of tuning MobiLink for two script versions is given in the following command line:

```
mlsrv10 -c "dsn=SQL Anywhere 10 Demo" -w 5 -cn 10
```

Since the maximum number of database connections used for synchronizations is the number of script versions times the number of worker threads, setting -cn to 10 ensures that database connections are not closed and opened excessively.

See "-cn option" on page 39.

Monitoring MobiLink performance

There are a variety of tools available to help you monitor the performance of your synchronizations.

The MobiLink Monitor is a graphical tool for monitoring synchronizations. It allows you to see the time taken by every aspect of the synchronization.

See "MobiLink Monitor" on page 145.

In addition, there are a number of MobiLink scripts that are available for monitoring synchronizations. These scripts allow you to use performance statistics in your business logic. You may, for example, want to store the performance information for future analysis, or alert a DBA if a synchronization takes too long. For more information, see:

- "download_statistics connection event" on page 300
- "download_statistics table event" on page 303
- "synchronization_statistics connection event" on page 376
- "synchronization_statistics table event" on page 379
- "time_statistics connection event" on page 382
- "time_statistics table event" on page 385
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405

CHAPTER 6

MobiLink Monitor

Contents

Introduction to the MobiLink Monitor

The MobiLink Monitor is a MobiLink administration tool that provides you with detailed information about the performance of your synchronizations.

When you start the Monitor and connect it to a MobiLink server, the Monitor begins to collect statistical information about all synchronizations that occur in that monitoring session. The Monitor continues to collect data until you disconnect it or shut down the MobiLink server.

You can view the data in tabular or graphical form in the Monitor interface. You can also save the data in binary format for viewing with the Monitor later, or in *.csv* format to open in another tool, such as Microsoft Excel; or you can export it to an ODBC data source such as a MobiLink-supported relational database.

Monitor output allows you to see a wide variety of information about your synchronizations. For example, you can quickly identify synchronizations that result in errors, or that meet other criteria that you specify. You can identify possible contention in synchronization scripts by checking to see if synchronizations of differing durations have phases that end around the same time (because synchronizations are waiting for a previous phase to finish before they can continue).

The MobiLink Monitor can be used routinely in development and production, because monitoring does not degrade performance, particularly when the Monitor is run on a different computer from the MobiLink server.

Starting the MobiLink Monitor

You can have multiple instances of the Monitor running for each MobiLink server.

♦ **To start monitoring data**

1. Start your consolidated database and MobiLink server, if they are not already running.
2. From the Start menu, choose Programs ► SQL Anywhere 10 ► MobiLink ► MobiLink Monitor.

   Alternatively, you can type **mlmon** at a command prompt. For details, see below.

   The Connect to MobiLink Server dialog appears.
3. A Monitor connection starts like a synchronization connection to the MobiLink server. For example, if you started the MobiLink server with -zu+ then it doesn't matter what user ID you use here. For all MobiLink Monitor sessions, the script version is set to for_ML_Monitor_only.

   The Connect to MobiLink Server dialog should be completed as follows:

   - **Host** is the network name or IP address of the computer where the MobiLink server is running. By default, it is the computer where the Monitor is running.
   - **Protocol** should be set to the same network protocol and port as the MobiLink server is using for synchronization requests.
   - **Port** should be set to the same network port as the MobiLink server is using for synchronization requests.
   - **Encryption** If you chose HTTPS or TLS for the protocol, this box is enabled. You can choose an encryption type from the dropdown list.
   - **Additional Protocol Options** allows you to set optional parameters. You can set the following parameters, separated by semicolon if you need to specify multiple parameters:
     - **buffer_size=***number*
     - **client_port=***nnnn*
     - **client_port=***nnnn-mmmmm*
     - **persistent=**{**0**|**1**} (HTTP and HTTPS only)
     - **proxy_host=***proxy_hostname* (HTTP and HTTPS only)
     - **proxy_port=***proxy_portnumber* (HTTP and HTTPS only)
     - **url_suffix=***suffix* (HTTP and HTTPS only)
     - **version=***HTTP-version-number* (HTTP and HTTPS only)

   See "MobiLink Client Network Protocol Options" [*MobiLink - Client Administration*].
4. Start synchronizing.

The data appears in the Monitor as it is collected.

Starting mlmon on the command line

Command line options allow you to have the Monitor open a file or connect to a MobiLink server on startup. Use the following syntax:

mlmon [*connect-options* | *inputfile*.{ **mlm** | **csv** }]

where:

connect-options can be one or more of the following:

- **-u** *ml_username* (required to connect to the MobiLink server)
- **-p** *password*
- **-x** { **tcpip** | **tls** | **http** | **https** } [**(** *keyword=value*;... **)**]

  The keyword=value pairs can be the host, protocol, and Additional Network Parameters as described above. The -x option is required to connect to the MobiLink server.
- **-o** *outputfile*.{ **mlm** | **csv** }

  The -o option closes the Monitor at the end of the connection and saves the session in the specified file.

You can type **mlmon -?** to view the mlmon syntax.

Starting the MobiLink Monitor on Unix

The following steps can be used if you are using a version of Linux that supports the Linux Desktop icons and if you chose to install them when you installed SQL Anywhere 10.

♦ To start the MobiLink Monitor (Linux Desktop icons)

1. Open the SQL Anywhere 10 folder on your desktop.
2. Double-click MobiLink Monitor.

   The MobiLink Monitor opens and the Connect to MobiLink Server dialog appears.
3. Enter the information to connect to the MobiLink server as described above.

Note
The following steps assume that you have already sourced the SQL Anywhere utilities. See "Setting environment variables on Unix and Mac OS X" [*SQL Anywhere Server - Database Administration*].

♦ To start the MobiLink Monitor (Unix command line)

1. In a terminal session, enter the following command:

   ```
   mlmon
   ```


The MobiLink Monitor opens and the Connect to MobiLink Server dialog appears.

2. Enter the information to connect to the MobiLink server as described above.

Stopping the MobiLink Monitor

♦ **To stop the MobiLink Monitor**

1. In the Monitor, choose Monitor ► Disconnect from MobiLink Server. This stops the collection of data.

   You can also stop collecting data by shutting down the MobiLink server or closing the Monitor.

   Before closing the Monitor, you can save the data for the session. See "Saving Monitor data" on page 158.

2. When you are ready to close the Monitor, choose File ► Close.

Using the MobiLink Monitor

The Monitor has the following panes:

- **Details Table** is the top pane. It is a spreadsheet that shows the total time taken by each synchronization, with a breakdown showing the amount of time taken by each part of the synchronization.

  See "Details Table pane" on page 150.

- **Utilization Graph** is the second pane. It provides a graphical representation of queue lengths for different queues on the MobiLink server. The same scale is used for the Graph pane and Chart pane. The scale at the bottom of the Graph pane represents time. You can select the data that is displayed in the Utilization Graph by drawing the Marquee tool around data in the Overview pane, or by choosing View ► Go To.

  See "Utilization Graph pane" on page 151.

- **Chart** is the third pane. It provides a graphical representation of synchronizations. The scale at the bottom of this pane represents time. You can select the data that is displayed in the Chart by drawing the Marquee tool around data in the Overview pane, or by choosing View ► Go To.

  See "Chart pane" on page 154.

- **Overview** is the bottom pane. It shows an overview of all synchronizations in the session. This pane has a small box called the Marquee tool that you can use to select the data that appears in the Chart and Utilization Graph panes.

  See "Overview pane" on page 155.

In addition, there is an Options dialog that you can use to customize the display, and properties dialogs for viewing more detailed information. See:

- "Options dialog" on page 155
- "Session properties" on page 155
- "Sample properties" on page 156
- "Synchronization properties" on page 157

Details Table pane

The Details Table provides information about the duration of each part of the synchronization. All times are measured by the MobiLink server. Some times may be non-zero even when you do not have the corresponding script defined.

You can choose the columns that appear in the Details Table pane by opening Tools ► Options and then opening the Table tab. For a description of the statistics that are available, see "MobiLink statistical properties" on page 161.

The following columns appear by default:

- **Sync** Identifies each synchronization. This ID is assigned by the MobiLink server, and not by the Monitor, so it will not necessarily start at 1 in any given Monitor session and will not be received in numerical order. You can see the same IDs in the Synchronization properties sheet. See "Synchronization properties" on page 157.
- **User** Identifies the synchronization user.
- **Version** The version of the synchronization script.

  See "Script versions" on page 225.
- **Start_Time** The date and time when the MobiLink server started the synchronization. (This may be later than when the synchronization was requested by the client.)
- **Duration** The total duration of the synchronization, in seconds.
- **Verify_Upload** The time in seconds for MobiLink to validate the synchronization request, validate the user name, and validate the password (if your synchronization setup requires authentication).
- **Preload_Upload** The time in seconds for MobiLink to receive the uploaded data from the client.
- **Begin_Sync** The time in seconds to run your begin_synchronization script, if one was run.
- **Upload** The time in seconds to apply the upload to the consolidated database. This is the time between the begin_upload script and the end_upload script.
- **Prepare_for_Download** The time in seconds to run your prepare_for_download script, if one was run.
- **Download** The time in seconds to download the data. This is the time between the begin_download script and the end_download script. If download acknowledgement is enabled, this includes the time to apply the download on the remote database and return acknowledgement.
- **End_Sync** The time in seconds to run the end_synchronization script, if one was run.

To sort the table by a specific column, click the column heading. If new data is appearing in the Monitor, it will be sorted as it is added.

You can close the Details Table pane by clearing Details Table in the View menu.

Utilization Graph pane

The Utilization Graph is the second pane from the top. It displays server statistics for several types of work queue.

For more information about the data available in this pane, see "Using the Utilization Graph" on page 152.

The Utilization Graph uses the same horizontal scrollbar, horizontal time labels, and zoom level as the Chart. This means that an instant in time lines up vertically between the Graph and the Chart.

There are multiple ways to select the data that is displayed in the graph:

- From the View menu, choose Go To.
- In the Overview pane, move the Marquee tool. The Marquee tool is the small box that appears in the Overview pane. See "Marquee tool" on page 155.

You can double-click an area of the Utilization Graph to bring up a Sample Properties dialog that shows the details of the sample interval it represents. The sample interval is about a second long. See "Sample properties" on page 156.

Using the Utilization Graph

To see the values for the Utilization Graph, as well as to customize the output, choose Tools ► Options and open the Graph tab. This tab identifies the Utilization graph queues by color, and allows you to customize the graph.

Properties

- **TCP/IP work queue** This queue represents work done by the low-level network layer in the MobiLink server. This layer is responsible for both reading and writing packets from and to the network. The queue is full of read and write requests. It grows when it gets notified of incoming data to read off the network and/or when told by the stream worker to write to the network.

  If this queue gets backed up, it is usually due to a backlog of either reads or writes—sometimes both but usually one or the other. Reads can get backed up if the server is using a lot of RAM and memory pages are being swapped in and out a lot. Consider getting more RAM. Writes can get backed up if the network connection between the clients and server is slow. If this queue is the only queue that is backed up, look at the CPU usage. If CPU usage is high, suspect read/memory problems. If CPU usage is low, you may have slow writes. Consider using a faster network.

- **Stream work queue** For version 10 clients only. This queue represents work done by the high-level network layer in the MobiLink server. This layer is responsible for higher-level network protocol work such as HTTP, encryption, and compression. This queue grows when lots of reads come in from the TCP/IP layer and/or when lots of write requests come in from the Command processor layer. If this queue is the only queue that is backed up, consider removing some network protocols, such as HTTP or compression. If this isn't possible, consider reducing the number of concurrent syncs allowed using the -sm option.

  See "-sm option" on page 68.

- **Heartbeat work queue** This queue represents the layer in MobiLink server that is responsible for sending pulsed events within the server. This layer is responsible, for example, for triggering the one-per-second pulses of samples to the connected MobiLink Monitors.

  It is highly unlikely that this queue gets backed up so it is not visible by default.

- **Command processor work queue** This represents work done by the MobiLink server to both interpret internal MobiLink protocol commands and apply these commands to the consolidated database. This queue grows when lots of requests come in. Request types include synchronization requests, Listener

requests, mlfiletransfer requests, and so on. The queue also grows when the consolidated database is busy working on synchronizations, yet more synchronization requests keep coming in.

If this queue is the only queue that is backed up, look at the CPU usage. If CPU usage is high, the volume of requests may be too high. Consider reducing the number of concurrent syncs allowed, using the -sm option. If CPU usage is low, look to improving the performance of the consolidated database.

See "-sm option" on page 68.

- **Busy database worker threads** This value indicates how hard the MobiLink server is pushing the consolidated database. Each unit in the value represents a database worker thread that is doing something in the database. There is no distinction between inserts, updates, deletes, or selects. When this value is zero, the server is not operating on the consolidated database.

  When this count is high (when it is close to the maximum set with the mlsrv10 -w option), the MobiLink server is pushing the consolidated database as hard as it can. In this case, if your throughput is satisfactory, there is nothing to do. If your throughput is not satisfactory, consider increasing the number of database worker threads via the -w option. Note that a higher -w value leads to greater contention between connections. This is particularly bad when all connections are performing uploads, so you may need to use the mlsrv10 -wu option to set a lower limit for database workers doing uploads. If you cannot seem to find settings for -w and -wu that provide adequate throughput, examine your synchronization scripts for possible contention issues. Finally, you can consult your RDBMS documentation for ways to improve the overall performance of your consolidated database.

Scale

This column tells you the current scale for each property.

The vertical scale on the Utilization Graph always goes from 0 to 100. This represents zero to one hundred percent of the scale. Each value has its own scale. By default, all scales are 5, meaning that values are expected to be in a range of 0 to 20, scaled (by 5) to the range 0 to 100. If a value becomes greater than 20, the scale is automatically adjusted such that the largest value is at 100.

To determine the maximum value in the display, divide the scale into 100. For example, if the TCP/IP work queue scale is 2.381, then the maximum value is (100 / 2.381) = 42. The actual maximum isn't usually important. What is important is that values towards the top of the graph are approaching the largest currently-known value for the given property—in other words, the peak load for that property as observed in the current monitoring session.

When the graphs are consistently towards the top of the display and you notice that synchronization throughput is down, you may have a performance problem that needs investigation. Similarly, if one or more values creeps upward over time without diminishing, then there is likely a performance problem. note that the graphs may often be towards the top of the display with MobiLink server performing well. This just means that MobiLink server is busy and doing its job well.

Antialiasing

One of your customization choices is antialiasing. Antialiasing makes the graph look better, but can be slower to draw.

Chart pane

The Chart pane presents the same information as the Details Table, but in graphical format. The bars in the Chart represent the length of time taken by each synchronization, with subsections of the bars representing the phases of the synchronization.

Viewing data

Click a synchronization to select that synchronization in the Details Table.

Double-click a synchronization to open the Synchronization Session Properties for the synchronization. See "Synchronization properties" on page 157.

Grouping data by user or compactly

You can group the data by user. Choose View ► By User.

Alternatively, you can view the data in a compact mode that shows all active synchronizations in as few rows as possible. Choose View ► Compact View. In Compact View, the row numbers are meaningless.

Zooming in on data

There are several ways to select the data that is visible in the Chart pane:

- **Zoom options** There are zoom options in the View menu and zoom buttons on the toolbar that allow you to zoom in and out. To have a synchronization fill the available space, use Zoom to Selection.
- **Scrollbar** Click the scrollbar at the bottom of the Chart pane and slide it.
- **Go To dialog** Open this dialog by choosing View ► Go To. The Go To dialog appears.

  Start Date & Time lets you specify the start time for the data that appears in the Chart pane. If you change this setting, you must specify at least the year, month, and date of the date-time.

  Chart Range lets you specify the duration of time that is displayed. The chart range can be specified in milliseconds, seconds, minutes, hours, or days. The chart range determines the granularity of the data: a smaller length of time means that more detail is visible.

- **Marquee tool** In the Overview pane, move the Marquee tool. The Marquee tool is the small box that appears in the Overview pane. See "Marquee tool" on page 155.

Time axis

At the bottom of the Chart pane there is a scale showing time periods. The format of the time is readjusted automatically depending on the span of time that is displayed. You can always see the complete date-time by hovering your cursor over the scale.

Default color scheme

You can view or set the colors in the Chart pane by opening the Options dialog (available from the Tools menu). The default color scheme for the Chart pane uses lime green for uploads, coral red for downloads, and blue for begin and end phases, with a darker shade for earlier parts of a phase.

For information about setting colors, see "Options dialog" on page 155.

Overview pane

The Overview pane shows an overview of the entire Monitor session. You can navigate through the session using the Marquee tool, which is the box inside the Overview pane.

Active synchronizations, completed synchronizations, and failed synchronizations are represented with colors. To set the colors, open the Monitor, choose Tools ► Options, and then click the Overview tab.

See "Options dialog" on page 155.

You can close the Overview pane by deselecting the Overview Pane in the View menu.

You can also separate the Overview pane from the rest of the Monitor window. In the Options dialog, open the Overview tab and clear the Keep Overview Window Attached to Main Window checkbox.

Marquee tool

The Marquee tool is the small box that appears in the Overview pane. You can use the Marquee tool to see different data, or to see data at different granularity. The area represented within the box is displayed in the Chart and Graph panes. You can use the Marquee tool as follows:

- Click in the Overview pane to move the Marquee tool (and thus move the start time of the data shown in the Chart or Utilization Graph).
- Drag in the Overview pane to redraw the Marquee tool to change the Marquee tool's location and size (and thus change the start time and the range of data). If you make the marquee box smaller, you shorten the interval of the visible data in the Chart, which makes more detail visible.

The default color for the outline of the Marquee tool is black. To change the color of the Marquee tool, choose File ► Options, open the Overview tab, and select a color in the Marquee section.

Options dialog

Options allow you to specify a number of settings, including colors and patterns for the graphical display in the Chart pane (the middle pane of the MobiLink Monitor) and the Overview pane (the bottom pane).

To open the Options dialog, open the Monitor and choose Tools ► Options.

Restoring defaults

To restore default settings, delete the file *.mlMonitorSettings*. This file is stored in your user profiles directory.

Session properties

The Session Properties dialog provides statistics about the monitoring session. It provides property values for the entire time that the Monitor has been running. To open the Session Properties dialog, open the Monitor and choose File ► Properties.

Session Properties has two tabs: General and Statistics.

The General tab provides basic information about the monitoring session.

The Statistics tab shows the same statistics as Sample Properties. See "Sample properties" on page 156.

Sample properties

The Sample Properties dialog provides detailed statistics for time intervals. Each time interval is about one second long. Samples are numbered by the Monitor to reflect the order in which they were received.

You can customize the appearance of the Graph to hide properties, but all properties appear in the Sample Properties dialog. If you have hidden a property, it is identified as Hidden in the Sample Properties; otherwise the color is shown.

To open the Sample Properties dialog, click in the Graph pane for the time period that you want to examine.

Sample properties has two tabs: Sample and Range.

The Sample tab provides information for the one-second time interval it represents. The properties that are displayed are for the end of the time interval.

The Range tab shows averages for the entire range of samples that were visible when the property sheet was opened (the horizontal range that is visible in the Overview). The Range statistics are not calculated until you click the Calculate button in the Range tab.

Sample Properties contains the following information:

- **Sample** Samples are numbered by the Monitor to reflect the order in which they were received. On the Sample tab, this shows the sample number. On the Range tab it shows the range of samples.
- **Start time and end time** On the Sample tab, this reflects a sample time period of approximately one second.
- **Statistics - Color column** The color used in the graph for this property.
- **Statistics - Property column** Shows the queue length for the sample or average queue for the range. This column displays the following types of property:
  - **TCP/IP work queue** This lists the number of buffers waiting to be filled by reading from the network plus the number of buffers waiting to be written to the network. The actual number is not very meaningful, but large numbers may indicate network-related bottlenecks.
  - **Stream work queue** For version 10 clients only. This queue represents work done by the high-level network layer in the MobiLink server. This layer is responsible for higher-level network protocol work such as HTTP, encryption, and compression.
  - **Heartbeat work queue** This is the queue length for periodic internal MobiLink server tasks other than synchronizations.
  - **Command processor work queue** This is the queue length for performing database tasks and interpreting or preparing communications with MobiLink clients. The actual number is not very meaningful, but large numbers may indicate database-related bottlenecks.

Note: The names of the properties are obtained from the MobiLink server or .mlm file and are in the language of the MobiLink server. Some characters may not display properly if the Monitor is using a different language.

- **Statistics - Value column** The property value.
- **Statistics - Limit column** The maximum allowed value for the property. This is useful so that the graph can use a scale of 0-100% for all properties. For properties such as page faults that are essentially unbounded, the limit is ignored.

Synchronization properties

Double-click a synchronization in either the Details Table or the Chart to see properties for that synchronization.

You can choose to see statistics for all tables (which is the sum for all tables in the synchronization), or for individual tables. The dropdown list provides a list of the tables that were involved in the synchronization.

For an explanation of the statistics in Synchronization Properties, see "MobiLink statistical properties" on page 161.

Saving Monitor data

You can save the data from a Monitor session as a binary file (.mlm), as a text file with comma-separated values (.csv), as tables in a relational database, or as a Microsoft Excel file.

Saving to file

To save the data as a file, choose File ► Save As.

- Save the data as a binary (.mlm) file if you want to view the saved data in the MobiLink Monitor. To reopen, choose File ► Open. The binary file format is the only format that preserves all monitored information.
- Save the data as a comma separated file (.csv) if you want to view it in another tool, such as Microsoft Excel. This will only save the information in the synchronization property sheets, except per table information and the session end time. You can also open a .csv file in the Monitor.

  In the .csv file format, time durations are stored in milliseconds.

You can specify that you want data to be saved automatically to a file. To do this, choose Tools ► Options, and enter an output file name on the General tab. The output file is overwritten by new data.

Exporting to a relational database or Excel

You can also export Monitor data using an ODBC connection. You can export to any relational database that is supported by MobiLink, as well as to Excel.

When you export data, all the columns in your Monitor session are exported, as well as a column called export_time that identifies the time the export was performed. Data from the graph is not exported.

The data source must have quoted identifiers enabled, because some of the columns are reserved words. The Monitor enables quoted identifiers automatically for SQL Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server databases. If the quoted identifiers option is not enabled, the export will fail.

♦ To export the data to a database or Excel

1. After collecting Monitor information, disconnect from the MobiLink server.
2. In the MobiLink Monitor, choose File ► Export to Database.

   The Export to Database dialog appears.
3. Select options for the output.

   - You can name the two tables that will be created to hold the data, or use the defaults (mlm_by_sync and mlm_by_table). If the tables do not exist, they will be created by the Monitor. For Excel output, the two table names identify the two worksheets that are created.
   - Choose whether you want to overwrite data in existing tables. If you do not choose to overwrite the data, new data is appended to existing data.
4. Click OK.

A Connect dialog appears that allows you to connect to the database or Excel spreadsheet using ODBC.

Customizing your statistics

The Watch Manager allows you to visibly distinguish synchronizations that meet criteria that you specify. For example, you might want to highlight big synchronizations, long synchronizations, small synchronizations that take a long time, or synchronizations that receive warnings.

To open the Watch Manager, open the Monitor and then click Tools ► Watch Manager.

The left pane of the Watch Manager contains a list of all available watches. The right pane contains a list of active watches. To add or remove a watch from the active list, select a watch in the left pane and click the appropriate button.

There are three predefined watches (Active, Completed, and Failed). You can edit predefined watches to change the way they are displayed, and you can deactivate them by removing them from the right pane.

No synchronizations are displayed in the Chart unless they meet the conditions of a watch. If you disable all watches (by removing them from the Current Watches list), then no synchronizations are shown in the Chart or Overview.

The order of watches in the right pane is important. Watches that are closer to the top of the list are processed first. Use the Move Up and Move Down buttons to organize the order of watches in the right pane.

You can use the predefined watches, and create other watches. To edit a watch condition, remove it and then add the new watch condition.

When a new Monitor connects to the same MobiLink server, it will show up a short synchronization in any Monitors that are already connected. The Monitor synchronization has the version name for_ML_Monitor_only. You can hide this Monitor synchronization with a watch.

♦ To create a new watch

1. In the Watch Manager, click New.

   The New Watch dialog appears.
2. Give the watch a name in the Name box.
3. Select a property, comparison operator, and value.

   For a complete list of properties, see "MobiLink statistical properties" on page 161.
4. Click Add. (You must click Add to save the settings.)
5. If desired, select another property, operator, and value, and click Add.
6. Select a pattern for the watch in the Chart pane. (The Chart pane is the middle pane in MobiLink Monitor.)
7. Select a color for the watch in the Overview pane. (The Overview pane is the bottom pane in the MobiLink Monitor.)

MobiLink statistical properties

Following is a list of the properties that are available in the MobiLink Monitor. These statistics can be viewed in the New Watch dialog, the Details Table pane, or the Synchronization Properties. In Synchronization Properties, the property names do not contain underscores.

For more information about the New Watch dialog, see "Customizing your statistics" on page 160.

For more information about the Details Table, see "Details Table pane" on page 150.

For more information about the Synchronization Properties dialog, see "Synchronization properties" on page 157.

Synchronization statistics

MobiLink statistical properties return the following information for synchronizations when not using forced conflict mode.

For information about forced conflict mode, see "Forced conflict statistics" on page 163.

Property	Description
active	True if the synchronization is in progress.
authenticate_user	Total time to perform user authentication, including executing the authenticate_* events.
begin_sync	Total time for the begin_synchronization event.
completed	True if the synchronization completed successfully.
conflicted_deletes	Always zero.
conflicted_inserts	Always zero.
conflicted_updates	Number of update rows that caused conflict. A row is included only when a resolve conflict script was successfully called for it.
connection_retries	Number of times the MobiLink server retried the connection to the consolidated database.
download	Total time for the download.
download_bytes	Amount of memory used within the MobiLink server to store the download. This provides a good indication of the impact on server memory of a synchronization.
download_deleted_rows	Number of row deletions fetched from the consolidated database by the MobiLink server (using download_delete_cursor scripts).
download_errors	Number of errors that occurred during the download.

Property	Description
download_fetched_rows	Number of rows fetched from the consolidated database by the MobiLink server (using download_cursor scripts).
download_filtered_rows	Number of fetched rows that were not downloaded to the MobiLink client because they matched rows that the client uploaded.
download_warnings	Number of warnings that occurred during the download.
duration	Total time for the synchronization, as measured by the MobiLink server.
end_sync	Total time for the end_synchronization event.
ignored_deletes	Number of upload delete rows that caused errors while the upload_delete script was invoked, when the handle_error or handle_odbc_error are defined and returned 1000, or when there is no upload_delete script defined for the given table.
ignored_inserts	The total number of upload insert rows that were ignored. They were ignored because 1) there is no upload_insert script in normal mode or no upload_new_row_insert script in forced conflict mode; or 2) errors occurred when the MobiLink server was invoking the corresponding script and the handle_error or handle_odbc_error event returned 1000.
ignored_updates	Number of upload update rows that caused conflict but a resolve conflict script was not successfully called or no upload_update script was defined.
prepare_for_download	Total time for the prepare_for_download event.
start_time	Date-time (in ISO-8601 extended format) for the start of the synchronization.
sync	A number uniquely identifying the synchronization within the Monitor session.
sync_deadlocks	Number of deadlocks in the consolidated database that were detected for the synchronization.
sync_errors	Total number of errors that occurred for the synchronization.
sync_request	The time the synchronization request was first noticed by the MobiLink server.
sync_tables	Number of client tables that were involved in the synchronization.
sync_warnings	Number of warnings that occurred for the synchronization.
upload	Total time for data to be uploaded to the consolidated database.

Property	Description
upload_bytes	Amount of memory used within the MobiLink server to store the upload. This provides a good indication of the impact on server memory of a synchronization.
upload_deadlocks	Number of deadlocks in the consolidated database that were detected during the upload.
upload_deleted_rows	Number of rows that were successfully deleted from the consolidated database.
upload_errors	Number of errors that occurred during the upload.
upload_inserted_rows	Number of rows that were successfully inserted in the consolidated database.
upload_updated_rows	Number of rows that were successfully updated in the consolidated database.
upload_warnings	Number of warnings that occurred during the upload.
user	MobiLink user name.
version	Name of the synchronization version.

Forced conflict statistics

When you are in forced conflict mode, MobiLink statistical properties return the following information.

Statistical property	Description
conflicted_deletes	Number of upload delete rows that were successfully inserted into the consolidated database using the upload_old_row_insert script.
conflicted_inserts	Number of upload insert rows that were inserted into the consolidated database using the upload_new_row_insert script.
conflicted_updates	Number of upload update rows that were successfully applied using the upload_new_row_insert or upload_old_row_insert scripts.
ignored_deletes	Number of upload delete rows that caused errors while the upload_old_row_insert script was invoked, when the handle_error or handle_odbc_error are defined and returned 1000, or when there is no upload_old_row_insert script defined for the given table.
ignored_inserts	Number of upload insert rows that caused errors while the upload_new_row_insert script was invoked, when the handle_error or handle_odbc_error are defined and returned 1000, or when there is no upload_new_row_insert script defined for the given table.

Statistical property	Description
ignored_updates	Number of upload update rows that caused errors while the upload_new_row_insert or upload_old_row_insert scripts were invoked, when the handle_error or handle_odbc_error are defined and returned 1000, or when there is no upload_new_row_insert and upload_old_row_insert script defined for the given table.
upload_deleted_rows	Always zero.
upload_inserted_rows	Always zero.
upload_updated_rows	Always zero.

CHAPTER 7

Synchronizing Through a Web Server with the Redirector

Contents

Introduction to the Redirector

MobiLink includes a web server extension called the **Redirector** that routes requests and responses between a client and the MobiLink server. A plug-in such as this is also commonly called a **reverse proxy**.

The main reason for routing requests through a web server is to use existing web server and firewall configurations for HTTP or HTTPS synchronization. However, a web server can operate as a proxy without the Redirector. The Redirector is most useful when you have more than one MobiLink server.

See "Options when using a web server" on page 167.

Using the Redirector, you can configure your web server to route specific URL requests to one or more computers running the MobiLink server.

Web servers can be configured to pass requests with specific URLs or ranges of URLs to extension programs commonly written in the form of Perl CGI scripts, DLLs, or other extension mechanisms. These extension programs may access external data sources and provide responses for the web server to deliver to its clients.

Load balancing and failover

The Redirector implements load balancing and failover using a simple round robin algorithm (servers are chosen in a fixed cyclic order). The Redirector pings each MobiLink server and stops sending requests to a server that is not responding. The Redirector detects when a MobiLink server is running again and resumes sending requests at that time.

HTTPS synchronization

When you specify the HTTPS protocol on a MobiLink client, HTTPS is used for the connection between the remote database and web server: HTTP headers are encrypted over TLS using RSA encryption before being sent to or from the web server, and the web server decrypts the HTTPS and sends HTTP to MobiLink via the Redirector. All Redirectors support this version of HTTPS, in which HTTPS is only used for the connection between the MobiLink client and the web server.

The HTTPS protocol is slower than other secure protocols.

Full HTTPS

For some Redirectors (such as the Apache Redirector, the ISAPI Redirector, and the NSAPI Redirector on Windows), the Redirector offers an option to re-encrypt the stream as HTTPS and send it to the MobiLink server.

For a list of Redirectors that support HTTPS from the Redirector to the MobiLink server, see "full HTTPS" in MobiLink Redirector Supported Web Servers.

Supported web servers

Plug-ins are provided for the following web servers:

Redirector plug-in	...supports
ISAPI Redirector	Microsoft web servers
NSAPI Redirector	Sun One (Netscape) and iPlanet web servers on Windows and Unix

Redirector plug-in	...supports
Servlet Redirector	Web servers that support the Java Servlet API 2.3, including Apache Tomcat and Sun One web servers on UNIX
Native Apache Redirector	Apache web server
M-Business Anywhere Redirector	M-Business Anywhere web server

For more information about Redirector support, see:

- Redirector: SQL Anywhere 10.0.1 Components by Platform
- MobiLink Redirector Supported Web Servers

Options when using a web server

The Redirector is one way to route MobiLink synchronization through a web server. The Redirector is particularly useful for synchronizing across a firewall or with multiple MobiLink servers.

The main reason for routing requests through a web server is to use existing web server and firewall configurations for HTTP or HTTPS synchronization. The Redirector is most useful when you have more than one MobiLink server.

You can also route synchronizations through a web server without using the Redirector. In this case, you might configure your web server as a proxy to route synchronizations to a MobiLink server. For more information on how to do this with your web server, see your web server documentation.

The following table contains recommendations to help you decide how best to route your MobiLink synchronizations.

	Direct connection possible	Direct connection not possible
One MobiLink server	Use TCP/IP instead of HTTP	Use an HTTP or HTTPS proxy to pass messages through the web server to the MobiLink server
Multiple MobiLink servers	Use the Redirector with HTTP or HTTPS	Use the Redirector with HTTP or HTTPS

See “Synchronizing Through a Web Server with the Redirector” on page 165.

Setting up the Redirector

The following sections describe how to configure your web server to manage synchronization requests.

♦ Overview of the configuration process

1. Configure the MobiLink server.

   See "Configuring MobiLink clients and servers for the Redirector" on page 169.

2. Modify the Redirector configuration file. There are two ways to do this, depending on whether you are using a Redirector that supports server groups or one that does not support server groups. See:

   - "Configuring Redirector properties (for Redirectors that support server groups)" on page 172
   - "Configuring Redirector properties (for Redirectors that don't support server groups)" on page 174

3. Perform web server-specific configuration.

   See one of the following:

   - "NSAPI Redirector for Netscape/Sun web servers on Windows" on page 177
   - "ISAPI Redirector for Microsoft web servers" on page 182
   - "Servlet Redirector" on page 184
   - "Apache Redirector" on page 187
   - "M-Business Anywhere Redirector" on page 189

4. Configure MobiLink clients.

   See "Configuring MobiLink clients and servers for the Redirector" on page 169.

Configuring MobiLink clients and servers for the Redirector

This section describes how to configure MobiLink clients and the MobiLink server for synchronization through a web server. The following procedures set the parameters required for requests directed through web servers.

MobiLink clients

- **To configure MobiLink clients (SQL Anywhere and UltraLite)**

1. Specify the communication type for MobiLink clients to HTTP or HTTPS.

   For more information about setting the communication type for SQL Anywhere clients, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

   For more information about setting the communication type for UltraLite clients, see "Network protocol options for UltraLite synchronization streams" [*MobiLink - Client Administration*].

2. Specify the following HTTP/HTTPS synchronization protocol options for MobiLink clients:

   - **host** the name or IP address of the web server.
   - **port** the web server port accepting HTTP or HTTPS requests.
   - **url_suffix** This setting depends on the type of Redirector you are using:
     - For the ISAPI Redirector:

       *exe_dir***iaredirect.dll/ml/**[*server-group***/**]

       where *exe_dir* is the location of *iaredirect.dll*, and *server-group* is optionally the name of the group.

     - For NSAPI Redirectors:

       *mlredirect***ml/**[*server-group***/**]

       where *mlredirect* is a name mapped in your *obj.conf* file.

     - For the servlet Redirector:

       iaredirect/ml/

     - For the native Redirector for Apache, set this to whatever you chose in the Redirector's <location> tag in the *httpd.conf* file. For example, if the location is `<Location / iaredirect/ml>`, then the url_suffix is:

       `iaredirect/ml/`

     - For the M-Business Anywhere Redirector, set this to whatever you chose in the Redirector's <location> tag in the *sync.conf* file. For example, if the location is `<Location / iaredirect/ml>`, then the url_suffix is:


```
iaredirect/ml/
```

See "url_suffix" [*MobiLink - Client Administration*].

For more information about setting protocol options for UltraLite clients, see "Network protocol options for UltraLite synchronization streams" [*MobiLink - Client Administration*].

For more information about setting protocol options for SQL Anywhere clients, see "MobiLink client network protocol options" [*MobiLink - Client Administration*].

MobiLink server

♦ **To configure MobiLink servers**

1. For Redirectors that support HTTPS, you can start the MobiLink server with the HTTPS protocol. For a list of Redirectors that support HTTPS, see MobiLink Redirector Supported Web Servers.

   For Redirectors that do not support HTTPS, the MobiLink server must be started with the HTTP protocol to use HTTP or HTTPS for communication between the client and the proxy. Theses Redirectors cannot use HTTPS directly.

   For example, the HTTP protocol may be specified on the mlsrv10 command line as follows:

   ```
   mlsrv10 -x http
   ```

   See "-x option" on page 77.

2. In addition, you may want to set the following parameter for the MobiLink server:

   - **port** for the HTTP protocol, MobiLink defaults to port 80. For the HTTPS protocol, MobiLink defaults to port 443. If the MobiLink server is running on the same machine as the web server, port 80 is normally in use by the web server. If this is the case you must specify a different port. For example, you could use port 2439, which is the Internet Assigned Numbers Authority (IANA)-registered port number for the MobiLink server.

   For more information about port, see "-x option" on page 77.

Configuring Redirector properties

There are different properties for Redirectors that support server groups and Redirectors that don't support server groups.

MobiLink server groups

You can partition your MobiLink servers into server groups. This allows you to have groups of clients that access distinct groups of MobiLink servers.

For a list of Redirectors that support server groups, see MobiLink Redirector Supported Web Servers.

You create a server group by adding a section to your Redirector configuration file (*redirector_server_group.config*) with the group name in square brackets followed by settings for that group. At a minimum, a group must specify one ML directive. You can also set the ML_CLIENT_TIMEOUT option for a group. You reference the group in the url_suffix option on your client.

You can create a default server group by specifying a group with no name before you specify any named groups in the file. This default group is useful for backward compatibility. It is used when the client does not specify a server group name in their url_suffix option.

See "url_suffix" [*MobiLink - Client Administration*].

You can also specify default settings for all server groups for the SLEEP and LOG_LEVEL properties. These can be specified anywhere in the configuration file.

Supporting old and new clients

If your MobiLink server needs to support old (version 8 or 9) remote databases as well as version 10 and up remote databases, then you need to open at least two ports: you use the mlsrv10 -x option to open a port for new clients, and you use the mlsrv10 -xo option to open a port for old clients. If you are also using the Redirector, you need to set up server groups so that the Redirector directs clients to the appropriate port.

In a typical Redirector setup, you would start multiple MobiLink servers. In the simplest case, you have one MobiLink server running with two ports, opened with -x and -xo, and you create two server groups, one for each. Following is a partial mlsrv10 command line that opens two ports for the MobiLink server:

```
mlsrv10 -c "dsn=YourDSN" -x http(port=111) -xo http(port=222)
```

You add sections to your Redirector configuration file for the two server groups:

```
[v10service]
    ML="host=mySrv.myCorp.com;port=111"
[v9service]
    ML="host=mySrv.myCorp.com;port=222"
```

When you start your clients, you specify the url_suffix option with the name of the server group. For example, for SQL Anywhere clients and an ISAPI web server, part of the dbmlsync command line for the version 10 clients would be:

```
dbmlsync -e "adr='host=somehost;port=5001;url_suffix=scripts/iaredirect.dll/
ml/v10service'"...
```


Part of the dbmlsync command line for your version 9 clients would be:

```
dbmlsync -e "adr='host=somehost;port=5001;url_suffix=scripts/iaredirect.dll/
ml/v9service'"...
```

See also

- "-x option" on page 77
- "-xo option" on page 82
- "Configuring Redirector properties (for Redirectors that support server groups)" on page 172
- "url_suffix" [*MobiLink - Client Administration*]

Example

Following is a sample *redirector_server_group.config* file, showing some typical settings and the creation of server groups.

```
#
# Set up the default server group:
#
  ML="host=mySrv1.myCorp.com;port=222"
  ML="host=mySrv2.myCorp.com;port=222"
#
# Set up a server group named myOldGroup:
#
[myOldGroup]
  ML="host=myOldSrv1.myCorp.com;port=111"
  ML="host=myOldSrv2.myCrop.com;port=111"
  ML_CLIENT_TIMEOUT=30
#
# Set up a server group named myNewGroup:
#
[myNewGroup]
  ML="host=myNewSrv1.myCorp.com;port=333"
  ML="host=myNewSrv2.myCorp.com;port=555"
  ML_CLIENT_TIMEOUT=240
#
# Set up a server group named mlSecureGroup:
#
[theirSecureGroup]
  ML="https=true;Srv1.Corp.com;trusted_certificates=c:\Corp\publicRoot.crt"
  ML="https=true;Srv2.Corp.com;trusted_certificates=c:\Corp\publicRoot.crt"
#
# Set global properties:
#
    LOG_LEVEL=5
    SLEEP=15
```

Configuring Redirector properties (for Redirectors that support server groups)

This section describes generic web server configuration steps to configure Redirector properties. It applies to Redirectors that support server groups.

For a list of Redirectors that support server groups, see MobiLink Redirector Supported Web Servers.

For information about server groups, see "MobiLink server groups" on page 171.

♦ To configure Redirector properties

1. Complete the steps in "Configuring MobiLink clients and servers for the Redirector" on page 169.
2. Configure a **Redirector configuration file**. A template file called *redirector_server_group.config* is provided in the *MobiLink\redirector* subdirectory of your SQL Anywhere installation. The easiest way to configure a Redirector configuration file is to modify *redirector_server_group.config*.

   The following rules apply to the Redirector configuration file:

   - The maximum line length is 2000 characters.
   - Comments start with the hash character (#).
   - For the ISAPI Redirector, the configuration file must be named *redirector.config* and must be in the same directory as *iaredirect.dll*.

   You can set the following directives in this file:

   - **Server groups** To create server groups, you create sections in *redirector_server_group.config* that start with a server group name in square brackets, and then define the server group.

     See "MobiLink server groups" on page 171.
   - **LOG_LEVEL** Used to control the amount of output written to the log file. Values are 0 to 7, with higher numbers generating more output. By default, the log file is called *redirector.log* and is located in the same place as the Redirector configuration file. For NSAPI Redirectors, you can change the name and location in magnus.conf using the logFile directive.
   - **ML** There are two ways that you can use the ML directive:
     - For Redirectors that do not support HTTPS from the Redirector to the MobiLink server or when you are not using HTTPS, you can use the ML directive to specify the list the computers running the MobiLink server, in the form `ML=host:port`. To specify multiple computers, you repeat this syntax on separate lines. For example:

       ```
       ML=209.123.123.1:8080
       ML=myCompany.com:8081
       ```
     - If your Redirector supports HTTPS from the Redirector to the MobiLink server and you are using HTTPS, you should specify MobiLink client network protocol options in a semicolon-separated list, as follows:

       ML="https=true;*network-client-options*;...**"**

       For example,

       ```
       ML="https=true;host=My-pc;port=82;trusted_certificates=rsaroot.crt"
       ```

       For a list of network client options, see "MobiLink Client Network Protocol Options" [*MobiLink - Client Administration*].

       For a description of HTTPS support in the Redirector, see "Full HTTPS" on page 166.

For a list of Redirectors that support HTTPS from the Redirector to the MobiLink server, see MobiLink Redirector Supported Web Servers.

Your MobiLink server must be started with the same protocol and port number as it is given in your ML directive. If there is a difference, you must stop the MobiLink server and restart it with the correct information.

- **ML_CLIENT_TIMEOUT** Used to ensure that the MobiLink server can detect duplicate synchronizations from the same remote database. The timeout should be set to the maximum timeout of any client using the same server group. If you set this property to 0, resynchronization to a different server is allowed immediately. The default value is 240 seconds.
- **SLEEP** Used to set the interval in seconds at which the Redirector checks that the servers are functioning. The Redirector checks one server, waits for the amount of time set in this option, checks the next server, and so on in a cycle. For example, `SLEEP=10`. SLEEP is case sensitive. The default is 20 seconds.

3. Copy the Redirector configuration file to the web server.

   If the MobiLink server is not installed on the same computer as the web server, copy the Redirector configuration file to the computer that holds the web server (or to a drive that computer has access to).

   For ISAPI web servers, copy the Redirector configuration file to the directory *Inetpub\scripts* and rename it *redirector.config*.

   For other web servers, you can copy the Redirector configuration file to any directory.

4. Complete web server-specific configuration in one of the following sections:

   - "ISAPI Redirector for Microsoft web servers" on page 182
   - "NSAPI Redirector for Netscape/Sun web servers on Windows" on page 177

Example

Following is a sample Redirector configuration file. This file specifies the following:

- The Redirector should sleep for 10 seconds after checking that a server is functioning.
- The three computers running MobiLink servers that are able to process requests.

```
SLEEP=10
ML=myServ-pc:80
ML=209.123.123.1:8080
ML=myCompany.com:8081
```

Configuring Redirector properties (for Redirectors that don't support server groups)

This section describes generic web server configuration steps to configure Redirector properties. It applies to Redirectors that don't support server groups.

For a list of Redirectors that support server groups, see MobiLink Redirector Supported Web Servers.

♦ To configure Redirector properties

1. Complete the steps in "Configuring MobiLink clients and servers for the Redirector" on page 169.
2. Copy *redirector.config* to the web server.

   The file *redirector.config* is provided with the MobiLink server installation, in the *MobiLink \redirector* subdirectory of your SQL Anywhere installation.

   If the MobiLink server is not installed on the same computer as the web server, copy *redirector.config* to the computer that holds the web server.
3. Configure the Redirector configuration file.

   To configure communications between the web server and MobiLink server, you must edit the file *redirector.config* on the computer that holds the web server.

   The following rules apply to *redirector.config*:

   - The maximum line length is 300 characters.
   - Comments start with the hash character (#).
   - You cannot include spaces or tabs in the directive definitions.

   You can set the following directives in this file:

   - **LOG_LEVEL** Used to control the amount of output written to the log file. Values are 0, 1, and 2, with 1 being the default and 2 generating the most output. For the Apache Redirector, this setting has no effect; set the log level in the LogLevel section of the Apache configuration file, *httpd.conf*.
   - **ML** ML is case sensitive. There are two ways that you can use the ML directive.

     For Redirectors that do not support HTTPS or when you are not using HTTPS, you can use the ML directive to specify the list the computers running the MobiLink server, in the form `ML=host:port`. To specify multiple computers, you repeat this syntax on separate lines. For example:

     ```
     ML=209.123.123.1:8080
     ML=myCompany.com:8081
     ```

     If your Redirector supports HTTPS from the Redirector to the MobiLink server, you can specify MobiLink client network protocol options in a semicolon-separated list, as follows:

     ML="https=true;*network-client-options***;..."**

     For example,

     ```
     ML="https=true;host=My-pc;port=82;trusted_certificates=rsaroot.crt"
     ```

     For a list of network client options, see "MobiLink Client Network Protocol Options" [*MobiLink - Client Administration*].

For a list of Redirectors that support HTTPS from the Redirector to the MobiLink server, see MobiLink Redirector Supported Web Servers.

Your MobiLink server must be started with the same protocol and port number as it is given in your ML directive. If there is a difference, you must stop the MobiLink server and restart it with the correct information.

- **ML_CLIENT_TIMEOUT** Used to ensure that each step of a single synchronization is directed to the same MobiLink server. The Redirector maintains an association between client and server for the duration of ML_CLIENT_TIMEOUT. This value is also used to ensure that the MobiLink server can detect duplicate synchronizations from the same remote database. The value of this parameter should be greater than the longest step in any user's synchronization.

  The default value is 600 seconds (ten minutes).

- **SLEEP** Used to set the interval in seconds at which the Redirector checks that the servers are functioning. The default is 1800 (30 minutes). For example, `SLEEP=3600`. SLEEP is case sensitive.

4. Complete web server-specific configuration in one of the following sections:

   - "NSAPI Redirector for Netscape/Sun web servers on Unix" on page 180
   - "Servlet Redirector" on page 184
   - "Apache Redirector" on page 187
   - "M-Business Anywhere Redirector" on page 189

Example

Following is a sample *redirector.config* file. This file specifies the following:

- The Redirector should check every 1800 seconds that the servers are functioning.
- The three computers running MobiLink servers that are able to process requests. When you specify multiple servers, load balancing is automatically enabled.

```
SLEEP=1800
ML=myServ-pc:80
ML=209.123.123.1:8080
ML=myCompany.com:8081
```


NSAPI Redirector for Netscape/Sun web servers on Windows

The NSAPI Redirector is provided for the Sun Java System web server, which was previously known as Sun One and the Netscape iPlanet Enterprise Edition web server.

For information about version support, see Redirector in SQL Anywhere 10.0.1 Components by Platform.

To use this Redirector on Unix, see "NSAPI Redirector for Netscape/Sun web servers on Unix" on page 180.

To use the Redirector with Netscape/Sun web servers on other platforms, you can use the servlet Redirector. See "Servlet Redirector" on page 184.

♦ To configure the NSAPI Redirector

1. Complete the steps in "Configuring Redirector properties (for Redirectors that support server groups)" on page 172.
2. If necessary, copy the file *iaredirect.dll* to the computer that holds the web server. This file is installed with the MobiLink server, in the *MobiLink\redirector\web-server* subdirectory of your SQL Anywhere installation, where *web-server* is the name of your NSAPI web server.
3. If your web server is on a separate computer from the Redirector, you must copy the following files to that computer and ensure that they are in your path. What files you need depends on what, if any, encryption you are using. The following file locations are relative to your SQL Anywhere installation:

Setup	Files required
All	♦ *win32\dblgen10.dll*[1] ♦ *win32\dbicu10.dll* ♦ *win32\dbicudt10.dll*
ECC encryption	♦ *win32\mlcecc10.dll*
RSA encryption	♦ *win32\mlcrsa10.dll*
RSA encryption with FIPS	♦ *win32\mlcrsafips10.dll* ♦ *win32\sbgse2.dll*

[1]For French, German, Japanese, and Chinese editions, substitute en with fr, de, ja, and zh, respectively.

For information about how to change the language, see "Understanding the locale language" [*SQL Anywhere Server - Database Administration*].

4. Update the NSAPI web server configuration files *magnus.conf* and *obj.conf* as follows.

Sample file provided
Sample copies of *magnus.conf* and *obj.conf*, preconfigured for the MobiLink server, are provided in the *MobiLink\redirector\web-server* subdirectory of your SQL Anywhere installation, where *web-server* is the name of your NSAPI web server.

Update the following sections of the files *magnus.conf* and *obj.conf*.

- In *magnus.conf*, specify where *iaredirect.dll* and the Redirector configuration file are located.

  At the end of the Init section, add the following text, where *location* is the actual location of the files. (*iaredirect.dll* and the Redirector configuration file can be in different locations, although both must be on the same computer as the web server or a drive that is accessible to the web server.)

  ```
  Windows:
  Init fn="load-modules" shlib="location/iaredirect.dll"
  funcs="redirector,initialize_redirector"
  Init fn="initialize_redirector" configFile="location/redirector.config"
  ```

- In *obj.conf*, specify the name of the Redirector to be used in URLs.

  At the beginning of the "default object" section, add the following text. This section should appear exactly as provided below, except that you can change *mlredirect* to whatever you want. All requests of the form *http://host:port/mlredirect/ml/** will be sent to one of the MobiLink servers running with the Redirector.

  ```
  <Object name=default>
  NameTrans fn="assign-name" from="/mlredirect/ml/*" name="redirectToML"
  ```

- In *obj.conf*, specify the objects that are called by the Redirector. After the "default object" section, add the following section:

  ```
  <Object name="redirectToML">
  Service fn="redirector" serverType="ml"
  </Object>
  ```

Example

Following is an example of the section of *magnus.conf* that you need to customize.

```
Init fn="load-modules" shlib="D:/iaredirect.dll"
funcs="redirector,initialize_redirector"
Init fn=" initialize_redirector " configFile="D:/redirector.config"
```

Following is an example of the sections of *obj.conf* that are important to the Redirector:

```
<Object name=default>
NameTrans fn="assign-name" from="/mlredirect/ml/*" name="redirectToML"
...
<Object name="redirectToML">
Service fn="redirector" serverType="ml"
</Object>
```

♦ To test your configuration

1. Call the Redirector using the following syntax:


```
http://host:port/mlredirect/ml/
```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink server.

NSAPI Redirector for Netscape/Sun web servers on Unix

The NSAPI Redirector is provided for the Sun Java System web server, which was previously known as Sun One and the Netscape iPlanet Enterprise Edition web server.

For information about version support, see Redirector.

To use this Redirector on Windows, see "NSAPI Redirector for Netscape/Sun web servers on Windows" on page 177.

To use the Redirector with Netscape/Sun web servers on other platforms, you can use the servlet Redirector. See "Servlet Redirector" on page 184.

♦ To configure the NSAPI Redirector

1. Complete the steps in "Configuring Redirector properties (for Redirectors that don't support server groups)" on page 174.
2. If necessary, copy the file *iaredirect.so* to the computer that holds the web server. This file is installed with the MobiLink server, in the *MobiLink/redirector/web-server* subdirectory of your SQL Anywhere installation, where *web-server* is the name of your NSAPI web server.
3. Update the NSAPI web server configuration files *magnus.conf* and *obj.conf* as follows.

> **Sample file provided**
> Sample copies of *magnus.conf* and *obj.conf*, preconfigured for the MobiLink server, are provided in the *MobiLink/redirector/web-server* subdirectory of your SQL Anywhere installation, where *web-server* is the name of your NSAPI web server. You can use these sample files to confirm where the following sections fit in to the file.

Update the following sections of the files *magnus.conf* and *obj.conf*.

- In *magnus.conf*, specify where *iaredirect.so* and *redirector.config* are located.

  At the end of the Init section, add the following text, where *location* is the actual location of the files. (*iaredirect.so* and *redirector.config* can be in different locations, although both must be on the same computer as the web server.)

  ```
  Solaris:
  Init fn="load-modules" shlib="location/iaredirect.so"
  funcs="redirector,initialize_redirector"
  Init fn="initialize_redirector" configFile="location/redirector.config"
  ```

- In *obj.conf*, specify the name of the Redirector to be used in URLs.

  At the beginning of the "default object" section, add the following text. This section should appear exactly as provided below, except that you can change *mlredirect* to whatever you want. All requests of the form *http://host:port/mlredirect/ml/** will be sent to one of the MobiLink servers running with the Redirector.


```
<Object name=default>
NameTrans fn="assign-name" from="/mlredirect/ml/*" name="redirectToML"
```

- In *obj.conf*, specify the objects that are called by the Redirector. After the "default object" section, add the following section:

```
<Object name="redirectToML">
Service fn="redirector" serverType="ml"
</Object>
```

Example

Following is an example of the section of *magnus.conf* that you need to customize.

```
Init fn="load-modules" shlib="location/iaredirect.so"
funcs="redirector,initialize_redirector"
Init fn=" initialize_redirector " configFile="location/redirector.config"
```

Following is an example of the sections of *obj.conf* that are important to the Redirector.

```
<Object name=default>
NameTrans fn="assign-name" from="/mlredirect/ml/*" name="redirectToML"
...
<Object name="redirectToML">
Service fn="redirector" serverType="ml"
</Object>
```

♦ To test your configuration

1. Call the Redirector using the following syntax:

   ```
   http://host:port/mlredirect/ml/
   ```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink server.

ISAPI Redirector for Microsoft web servers

If you are using a Microsoft web server, you can use the ISAPI version of the Redirector.

For information about version support, see Redirector in SQL Anywhere 10.0.1 Components by Platform.

♦ To configure ISAPI Redirector for Microsoft web servers

1. Complete the steps in "Configuring Redirector properties (for Redirectors that support server groups)" on page 172.
2. Copy the file *iaredirect.dll* to *Inetpub\scripts* on the computer that holds the web server.

   The file *iaredirect.dll* is installed with the MobiLink server, in *MobiLink\redirector\IIS5* under your SQL Anywhere installation directory.

   The directory *Inetpub\scripts* is in the Microsoft web server installation directory.
3. If your web server is on a separate computer from the Redirector, you must copy the following files to that computer and ensure that they are in your path. What files you need depends on what, if any, encryption you are using. The following file locations are relative to your SQL Anywhere installation:

Setup	Files required
All	♦ *win32\dblgen10.dll*[1] ♦ *win32\dbicu10.dll* ♦ *win32\dbicudt10.dll*
ECC encryption	♦ *win32\mlcecc10.dll*
RSA encryption	♦ *win32\mlcrsa10.dll*
RSA encryption with FIPS	♦ *win32\mlcrsafips10.dll* ♦ *win32\sbgse2.dll*

   [1]For French, German, Japanese, and Chinese editions, substitute en with fr, de, ja, and zh, respectively.

   For information about how to change the language, see "Understanding the locale language" [*SQL Anywhere Server - Database Administration*].
4. If your configuration is not successful, check the following:

   - The directory *Inetpub\scripts* should be created during the web server installation with execute permissions.
   - You can put your Redirector configuration file and *iaredirect.dll* in a different directory only if you use Internet Information Services to give execute permissions to the directory.
   - You must have a virtual directory that points to the *Inetpub\scripts* directory. If you do not, you must open Internet Information Services and manually create a virtual directory. This virtual directory should point to *Inetpub\scripts* and have Execute Permissions set to Scripts and Executables. See the IIS online help for instructions.

Note

- **To test your configuration**

1. Call the ISAPI Redirector using the following syntax:

   *protocol***://***host*[**:***port*]**/***exec_dir***/iaredirect.dll/ml/**

   where:

   - **protocol** is **http** or **https**.
   - **host** is the host name of the web server.
   - **port** is the port on which the web server is listening, if it is not the default port.
   - **exec_dir** is the directory where you installed the Redirector DLL, *iaredirect.dll*. The default directory is *scripts*.

     For example,

     ```
     http://server:8080/scripts/iaredirect.dll/ml/
     ```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink server.

Servlet Redirector

The servlet Redirector is provided for web servers that support the Java servlet specification version 2.3 and higher. The following procedure is an example of how to set up the servlet Redirector for Tomcat version 5.5.9 and Apache 2.0.55.

For information about version support, see Redirector in SQL Anywhere 10.0.1 Components by Platform.

There is also a native Redirector for Apache web servers. For more information, see "Apache Redirector" on page 187.

Overview

This section describes how to install the servlet version of the Redirector to work on an Apache web server in conjunction with the Tomcat servlet container.

Installation requires the following steps:

♦ To configure the servlet Redirector for Apache Tomcat

1. Complete the steps in "Configuring Redirector properties (for Redirectors that don't support server groups)" on page 174.
2. Install the servlet version of the Redirector in Tomcat.
3. Configure the Apache web server to run as a proxy.

Install the servlet Redirector in Tomcat

In the following procedure, *%CATALINA_HOME%* is the root directory of your Tomcat installation.

♦ To install the servlet Redirector in Tomcat

1. Install Tomcat as a standalone server.
2. Optionally, set the required Tomcat HTTP port.

   Tomcat binds to port 8080 by default. If there is a conflict, perhaps because another web server is using this port,

   - Open the file: *%CATALINA_HOME%/conf/server.xml*.
   - Search for 8080 (which is in a <Connector> tag).
   - Change it to a port that is not in use.
3. Install the servlet Redirector as a web application.

   - Copy *iaredirect.war* file to *%CATALINA_HOME%/webapps*.
   - Shut down and restart Tomcat.

     Tomcat expands the war file and creates the directory *iaredirect* for the Redirector web application.

- Edit the file *%CATALINA_HOME%/webapps/iaredirect/WEB-INF/web.xml*. Search for **redirector.config** (in an <init-param> tag), and correct the path for the *redirector.config* file.

  Change the entry **redirector.config** to read *drive*`:/`*path*`/redirector.config`. Even on Windows operating systems, use a forward slash as a path separator, as in *d:/redirector.config*.

- Shut down and restart Tomcat for the changes to take effect.

  Once the changes have taken effect, you no longer need the war file in the deployed location.

- The Redirector can now be invoked through the following URL:

  http://*tc-host*:*tc-port*/iaredirect/ml/

  where *tc-host* is the machine and *tc-port* the port on which Tomcat is listening.

Configure the Apache web server as a proxy

In the following procedure, *%APACHE_HOME%* is the root directory of your Apache installation.

♦ To configure the Apache web server as a proxy

1. Install the Apache web server.
2. Optionally, change the Apache web server port:

   - Edit the file *%APACHE_HOME%/conf/httpd.conf* and change the **Port** setting to the desired port.

3. Configure Apache to run as a proxy:

   In *%APACHE_HOME%/conf/httpd.conf*, add the following directives:

   ```
   LoadModule proxy_module module-path/mod_proxy.so
   LoadModule proxy_connect_module module-path/mod_proxy_connect.so
   LoadModule proxy_http_module module-path/mod_proxy_http.so
   ```

   where *module-path* is the location of the module. For example, the path may be *modules/mod_proxy.so* (the default).

4. Configure Apache to forward Redirector URLs to Tomcat.

   In *%APACHE_HOME%/conf/httpd.conf*, add the following directive so that Apache forwards URLs of the form http://localhost/iaredirect/* to the Tomcat 5 Connector listening on port 8080:

   ```
   ProxyPass /iaredirect http://localhost:8080/iaredirect
   ```

   The port number must match the port number used for Tomcat. If Tomcat and Apache are not running on the same computer, provide the computer name where Tomcat is running instead of **localhost**.

Verifying your setup

♦ To check your configuration

1. Call the Redirector using the following syntax:


```
http://host:port/iaredirect/ml/
```

2. Check the log file to see if the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink server.

Apache Redirector

The following setup instructions are written for Apache.

For information about version support, see Redirector in SQL Anywhere 10.0.1 Components by Platform.

If you are using Tomcat, you can also use the servlet Redirector. For more information, see "Servlet Redirector" on page 184.

♦ To configure the Apache Redirector

1. Complete the steps in "Configuring Redirector properties (for Redirectors that don't support server groups)" on page 174.
2. Copy the file *mod_iaredirect.dll* or *mod_iaredirect.so* to the appropriate directory in your web server, as follows:
   - For Apache on Windows, the file *mod_iaredirect.dll* is located in the *MobiLink\redirector\apache\v20* subdirectory of your SQL Anywhere installation. Copy this file to the *%apache-home%\modules* directory on the computer that holds the web server.
   - For Apache for Solaris or Linux, the file *mod_iaredirect.so* is located in the *MobiLink/redirector/apache/v20/* subdirectory of your SQL Anywhere installation. Copy it to the *$APACHE_HOME/modules* directory on the computer that holds the web server.
3. If your web server is on a separate computer from the Redirector, you must copy the following files to that computer and ensure that they are in your path (Windows) or shared path (Unix). What files you need depends on what, if any, encryption you are using. The following file locations are relative to your SQL Anywhere installation:

Setup	Files required
ECC encryption	♦ Windows: win32\mlcecc10.dll ♦ Unix: lib32/libmlcecc10_r.so
RSA encryption	♦ Windows: win32\mlcrsa10.dll ♦ Unix: lib32/libmlcrsa10_r.so
RSA encryption with FIPS	♦ Windows: win32\mlcrsafips10.dll and win32\sbgse2.dll ♦ Unix: lib32/libmlcrsafips10_r.so and libsbgse2_r.so

4. Update the Apache web server configuration file *httpd.conf* as follows.
   - In the LoadModule section, add the following line for Windows:

     ```
     LoadModule iaredirect_module modules/mod_iaredirect.dll
     ```

     or the following line for Solaris and Linux:

     ```
     LoadModule iaredirect_module modules/mod_iaredirect.so
     ```

   - Add the following section to the file:


```
<Location /iaredirect/ml>
   SetHandler iaredirect-handler
   iaredirectorConfigFile location/redirector.config
</Location>
```

where */iaredirect/ml* is the relative URL path that you will use to invoke the Redirector, and *location* is the directory where *redirector.config* is located.

- If you are using Apache on Solaris or Linux, you may also want to add the following optional directives to the <Location> section you just created:
  - **MaxSyncUsers *number*** The maximum number of MobiLink users synchronizing through the Redirector. This number is used to allocate necessary resources to the Redirector. This number cannot be less than 60. The default is 1000. Only change this setting if the default number of users is less than the actual number.
  - **ShmemDiagnosis on|off** If set to on, allows debugging of the memory resource. The default is off.

5. To help with debugging, you may want to increase the amount of logging information that the Redirector outputs. To do this, modify the LogLevel directive in *httpd.conf* and set it to **LogLevel info**. The log level can be (from most to least verbose): debug, info, notice, warn, error, crit, alert, and emerg.

Example

Following are examples of the sections of *httpd.conf* that configure the Apache web server to route requests to the MobiLink server. This example works for Windows. For UNIX and Linux, change *mod_iaredirect.dll* to *mod_iaredirect.so*.

```
LoadModule iaredirect_module modules/mod_iaredirect.dll
...

<Location /iaredirect/ml>
    SetHandler iaredirect-handler
    iaredirectorConfigFile c:/redirector.config
</Location>
```

- **To test your configuration**

1. Call the Redirector using the following syntax:

   ```
   http://host:port/iaredirect/ml/
   ```

   where *iaredirect/ml* is the relative URL path you specified in the <Location> tag of *httpd.conf*.

2. Check the log file to see if the Redirector logged a request. The default location of the log file is *$APACHE_HOME/logs/error.log*.

   Note: This test does not make a connection to the MobiLink server.

M-Business Anywhere Redirector

The following setup instructions are written for M-Business Anywhere on Windows, Solaris, or Linux.

For information about version support, see Redirector in SQL Anywhere 10.0.1 Components by Platform.

♦ To configure the M-Business Anywhere Redirector

1. Complete the steps in "Configuring Redirector properties (for Redirectors that don't support server groups)" on page 174.
2. Copy the file *mod_iaredirect.dll* or *mod_iaredirect.so* to the M-Business Anywhere *\bin* directory on the computer that holds the web server. This file is located in the *MobiLink\redirector \MBusinessAnywhere* subdirectory of your SQL Anywhere installation.
3. If your web server is on a separate computer from the Redirector, you must copy the following files to that computer and ensure that they are in your path (Windows) or shared path (Unix). What files you need depends on what, if any, encryption you are using. The following file locations are relative to your SQL Anywhere installation:

Setup	Files required
ECC encryption	♦ Windows: win32\mlcecc10.dll ♦ Unix: lib32/libmlcecc10_r.so
RSA encryption	♦ Windows: win32\mlcrsa10.dll ♦ Unix: lib32/libmlcrsa10_r.so
RSA encryption with FIPS	♦ Windows: win32\mlcrsafips10.dll and win32\sbgse2.dll ♦ Unix: lib32/libmlcrsafips10_r.so and libsbgse2_r.so

4. For Windows, update the M-Business Anywhere web server configuration file *sync.conf* as follows:

   ♦ In the LoadModule section, add the following line:

   ```
   LoadModule iaredirect_module path/bin/mod_iaredirect.dll
   ```

   where *path* is the location of the M-Business Anywhere *bin* directory.

   ♦ In the SyncLoadFile section, add the following line:

   ```
   SyncLoadFile path/bin/mod_iaredirect.dll
   ```

   where *path* is the location of the M-Business Anywhere *bin* directory.

   ♦ Add the following section to the file:

   ```
   <Location /iaredirect/ml>
      SetHandler iaredirect-handler
      iaredirectorConfigFile location/redirector.config
   </Location>
   ```


where */iaredirect/ml* is the path that you will use to invoke the Redirector, and *location* is the directory where *redirector.config* is located.

5. For Solaris and Linux, update the M-Business Anywhere web server configuration file *sync.conf* as follows:

   - In the LoadModule section, add the following line:

     ```
     LoadModule iaredirect_module path/bin/mod_iaredirect.so
     ```

     where *path* is the location of the M-Business Anywhere *bin* directory.

   - Add the following section to the file:

     ```
     <Location /iaredirect/ml>
        SetHandler iaredirect-handler
        iaredirectorConfigFile location/redirector.config
     </Location>
     ```

     where */iaredirect/ml* is the relative URL path that you will use to invoke the Redirector, and *location* is the directory where *redirector.config* is located.

   - You may also want to add the following optional directives to the <Location> section you just created:

     - **MaxSyncUsers *number*** The maximum number of MobiLink users synchronizing through the Redirector. This number is used to allocate necessary resources to the Redirector. This number cannot be less than 60. The default is 1000. Only change this setting if the default number of users is less than the actual number.

     - **ShmemDiagnosis on|off** If set to on, allows debugging of the memory resource. The default is off.

6. To help with debugging, you may want to increase the amount of logging information that the Redirector outputs. To do this, modify the LogLevel directive in *sync.conf* and set it to **LogLevel info**. The log level can be (from most to least verbose): debug, info, notice, warn, error, crit, alert, and emerg.

7. Restart your M-Business Sync Server for the changes to take effect.

Example

Following are examples of the sections of *sync.conf* that configure the M-Business Anywhere web server to route requests to the MobiLink server.

This example works on Windows:

```
LoadModule iaredirect_module "c:\program files\M-Business Anywhere/bin/
mod_iaredirect.dll"
...
SyncLoadFile "c:\program files\M-Business Anywhere/bin/mod_iaredirect.dll"
...
<Location \iaredirect\ml>
    SetHandler iaredirect-handler
    iaredirectorConfigFile "c:\AvantGoServer\conf/redirector.config"
</Location>
```

The following example works on UNIX and Linux:


```
LoadModule iaredirect_module modules/mod_iaredirect.so
...
<Location /iaredirect/ml>
    SetHandler iaredirect-handler
    iaredirectorConfigFile "/redirector.config"
</Location>
```

♦ To test your configuration

1. Call the Redirector using the following syntax:

   `http://`*host*`:`*port*`/`*iaredirect*`/ml/`

   where *iaredirect* is the path you specified in the <Location> tag of *sync.conf*.

2. Check the log files *sync_access.log* and *sync_error.log* to verify that the Redirector logged a request.

   Note: This test does not make a connection to the MobiLink server.

CHAPTER 8

MobiLink File-Based Download

Contents

Introduction to file-based download

File-based download is an alternative way to download data to SQL Anywhere remote databases: downloads can be distributed as files, enabling offline distribution of synchronization changes. This allows you to create a file once and distribute it to many remote databases.

With file-based download, you can put download synchronization changes in a file and transfer it to SQL Anywhere remote databases in any way a file can be transferred. For example, you can:

- broadcast the data by satellite multicast
- apply the update using Sybase Afaria
- email or FTP the file to users

You choose the users you want to receive the file. Full synchronization integrity is preserved in file-based download, including conflict detection and resolution. You can ensure that the file is secure by applying third-party encryption on the file.

When to use

File-based downloads are useful when a large amount of data changes on the consolidated database, but the remote database does not update the data frequently or does not do any updates at all. For example, price lists, product lists, and code tables.

File-based downloads are not useful when the downloaded data is updated frequently on the remote database or when you are running frequent upload-only synchronizations. In these situations, the remote sites may be unable to apply download files because of integrity checks that are performed when download files are applied.

File-based downloads currently can be used only with SQL Anywhere remote databases.

Download-only publications

In most cases, you should use a download-only publication for your file-based download. Use a regular publication only when you need to perform uploads with the same publication as you perform file-based downloads.

See "Download-only publications" [*MobiLink - Client Administration*].

If you use a regular publication, file-based downloads cannot be used as the sole means of updating remote databases. In that case you still need to regularly perform full synchronizations or upload-only synchronizations. Full or upload-only synchronizations are required to advance log offsets and to maintain the log file, which otherwise will grow large and slow down synchronization. A full synchronization may also be required to recover from errors.

Setting up file-based download

The following steps provide an overview of the tasks required to set up file-based download, assuming that you already have MobiLink synchronization set up.

In most cases, you should use a download-only publication with your file-based download. Use a regular publication only when you need to do uploads with the same publication as you do file-based downloads.

♦ Overview of setting up file-based download

1. Create a file-definition database.

   See "Creating the file-definition database" on page 195.

2. At the consolidated database, create scripts with a new script version.

   See "Changes at the consolidated database" on page 196.

3. Create a download file.

   See "Creating the download file" on page 196.

4. Apply the download file.

   See "Synchronizing new remotes" on page 196.

Other resources for getting started

- "File-based download examples" on page 201

Creating the file-definition database

To set up file-based download, you create a **file-definition database**. This is a SQL Anywhere database that has the same synchronization tables and publications as your remote databases. It can be located anywhere. This database contains no data or state information. It does not have to be backed up or maintained; in fact, you can delete it and recreate it as needed.

The file-definition database must include the following:

- the same publications as the remote databases, as well as the tables and columns used in the publication, the foreign key relationships and constraints of those tables and columns, and the tables required by those foreign key relationships.
- a MobiLink user name that identifies the group of remote databases that are to apply the download file. You will use this group MobiLink user name in your synchronization scripts to identify the group of remote databases.

Changes at the consolidated database

On the consolidated database, create a new script version for your file-based download, and implement any scripts required by your existing synchronization system into it. Upload scripts are not required. This script version will be used only for file-based download. For this script version, all scripts that take MobiLink user names as parameters will instead take a MobiLink user name that refers to a group of remote databases. This is the user name that is defined in the file-definition database.

For each script version that you have defined, implement a begin_publication script.

For timestamp-based downloads, implement a modify_last_download_timestamp script for each script version. How you implement this script depends on how much data you intend to send in each download file. For example, one approach is to use the earliest time that any user from the group last downloaded successfully. Remember that the ml_username parameter passed to this script is actually the group name.

See also

- "Script versions" on page 225
- "begin_publication connection event" on page 278
- "modify_last_download_timestamp connection event" on page 357

Creating the download file

The download file contains the data to be synchronized. To create the download file, set up your file-definition database and consolidated database as described above. Run dbmlsync with the -bc option and supply a file name with the extension *.df*. For example,

```
dbmlsync -c "uid=DBA;pwd=sql;eng=fbdl_eng;dbf=fdef.db" -v+
   -e "sv=filebased" -bc file1.df
```

You can also choose to specify options when you create the download file:

- **-be option** Use -be to add a string to the download file that can be accessed at the remote database using the sp_hook_dbmlsync_validate_download_file stored procedure.

  See "-be option" [*MobiLink - Client Administration*] and "sp_hook_dbmlsync_validate_download_file" [*MobiLink - Client Administration*].

- **-bg option** Use the -bg option to create a download file that can be used by remotes that have never synchronized.

Synchronizing new remotes

If you want to apply a download file to a remote database that has never synchronized using MobiLink, then before you apply the download file you need to either perform a normal synchronization on the remote database or use the dbmlsync -bg option when creating the download file.

For timestamp-based synchronization, doing either of these two things causes the download of an initial snapshot of the data. For both timestamp and snapshot based synchronization, this step sets the generation number to the value that is generated by the begin_publication script on the consolidated database.

Perform a normal synchronization

You can prepare a remote database to receive download files by performing a synchronization that does not use a download file.

Use the -bg option

Alternatively, you can create a download file with the -bg option to use with remotes that haven't yet synchronized. You apply this initial download file to prepare the remote database for file-based synchronization.

- **Snapshot downloads** If you are performing snapshot downloads, then the initial download file just needs to set the generation number. You may choose to include an initial snapshot of the data in this file, but since each snapshot download contains all the data and does not depend on previous downloads, this is not required.

  For snapshot downloads, using the -bg option is straightforward. Just specify -bg in the dbmlsync command line when you create the download file. You can use the same script version to create the initial download file as you use for subsequent download files.

- **Timestamp-based downloads** If you are performing timestamp-based downloads, then the initial download must set the generation number on the remote database and include a snapshot of the data. With timestamp-based downloads, each download builds on previous ones. Each download file contains a last download timestamp. All rows changed on the consolidated after the file's last download timestamp are included in the file. To apply a file, a remote database must already have received all the changes that occurred before the file's last download timestamp. This is confirmed by checking that the file's last download timestamp is greater than or equal to the remote database's last download timestamp (the time up to which the remote database has received all changes from the consolidated database).

  Before a remote can apply its first normal download file, it must receive all data changed before that file's last download timestamp and after January 1, 1900. The initial download file created with the -bg option must contain this data. The easiest way to select this data is to create a separate script version that uses the same download_cursor's as your normal file-based synchronization script version but does not have a modify_last_download_timestamp script. If no modify_last_download_timestamp script is defined, then the last download timestamp for a file-based download will default to January 1, 1900.

If you apply download files built with the -bg option to remote databases that have already synchronized, the -bg option causes the generation numbers on the remote database to be updated with the value on the consolidated database at the time the download file was created. This defeats the purpose of generation numbers, which is to prevent you from applying further file-based downloads until an upload has been performed in situations such as when recovering a consolidated database that is lost or corrupted.

See “MobiLink generation numbers” on page 199.

Validation checks

Before applying a download file to a remote database, dbmlsync does a number of things to ensure that the synchronization is valid.

- dbmlsync checks the download file to ensure that the file-definition database that was used to create it has:
  - the same publication as the remote database
  - the same tables and columns used in the publication
  - the same foreign key relationships and constraints as those tables and columns
- dbmlsync checks to see if there is any data in the publication that has not been uploaded from the remote. If there is, the download file is not applied, because applying the download file could cause pending upload data to be lost.
- dbmlsync checks the last download timestamp, next last download timestamp, and creation time of the download file to ensure that:
  - newer data on the remote database will not be overwritten by older data contained in the download file.
  - a download file will not be applied if applying it means that the remote database would miss some changes that have occurred on the consolidated database. This situation might occur if the remote did not apply previous file-based downloads.

    See "Automatic validation" on page 198.
- Optionally, dbmlsync checks the generation number in the remote database to ensure it matches the generation number in the download file.

  See "MobiLink generation numbers" on page 199.
- Optionally, you can create custom validation logic with the sp_hook_dbmlsync_validate_download_file stored procedure.

  For more information, see "Custom validation" on page 200.

Automatic validation

Before applying a download file, dbmlsync performs special checks on the last download timestamp, next last download timestamp, download file creation time, and transaction log.

Last download timestamp and next last download timestamp

Each download file contains all changes to be downloaded that occurred on the consolidated database between the file's last download timestamp, and its next last download timestamp. Both times are expressed in terms of the time at the consolidated database. By default the file's last download time is Jan 1, 1900 12:00 AM and the file's next last download timestamp is the time the download file was created. These defaults

can be overridden by implementing the modify_last_download_timestamp and modify_next_last_download_timestamp scripts on the consolidated database.

A remote site can apply a download file only if the file's last download timestamp is less than or equal to the remote's last download timestamp. This ensures that a remote never misses operations that occur on the consolidated database. Usually when a file-based download fails based on this check, the remote has missed one or more download files. The situation can be corrected by applying the missing download files or by performing a full or download-only synchronization.

In addition, a remote site can apply a download file only if the file's next last download timestamp is greater than the remote's last download timestamp. The remote's last download timestamp is the time (at the consolidated database) up to which the remote has received all changes that are to be downloaded. The remote database's last download time is updated each time the remote successfully applies a download (normal or file-based). This check ensures that a download file will not be applied if more recent data has already been downloaded. A common case where this could happen occurs when download files are applied out of order. For example, suppose a download file *F1.df* is created, and another file *F2.df* is created later. This check ensures that *F1.df* cannot be applied after *F2.df*, because that could allow newer data in *F2.df* to be overwritten with older data in *F1.df*.

When a file-based download fails based on the next last download timestamp, no additional action is required other than to delete the file. Synchronization will succeed once a new file is received.

Creation time

The download file's creation time indicates the time at the consolidated database when creation of the file began. A download file can only be applied if the file's creation time is greater than the remote database's last upload time. The remote's last upload time is the time at the consolidated database when the remote's last successful upload was committed. This check ensures that data that has been uploaded after the creation of the download (and hence is newer than the download) will not be overwritten by older data in the download file.

When a download file is rejected based on this check, no action is required. The remote site should be able to apply the next download file.

When an upload fails because dbmlsync sent an upload to the MobiLink server but got no acknowledgement, the remote database's last upload time may be incorrect. In this case, the creation time check cannot be performed and the remote is unable to apply download files until it completes a normal synchronization.

Transaction log

Before applying a download file, dbmlsync scans the remote database's transaction log and builds up a list of all changes that must be uploaded. Dbmlsync will only apply a download file if it does not contain any operations that affect rows with changes that must be uploaded.

MobiLink generation numbers

Generation numbers provide a mechanism for forcing remote databases to upload data before applying any more download files. This is especially useful when a problem on the consolidated database has resulted in data loss and you must recover lost data from the remote databases.

On the remote database, a separate generation number is automatically maintained for each subscription. On the consolidated database, the generation number for each subscription is determined by the begin_publication script. Each time a remote performs a successful upload, it updates the remote generation number with the value set by the begin_publication script in the consolidated database.

Each time a download file is created, the generation number set by the begin_publication script is stored in the download file. A remote site will only apply a download file if the generation number in the file is equal to the generation number stored in the remote database.

Note
Whenever the generation number generated by the begin_publication script for a file-based download changes, the remote databases must perform a successful upload before they can apply any new download files.

The sp_hook_dbmlsync_validate_download_file stored procedure can be used to override the default checking of the generation number.

For more information about managing MobiLink generation numbers, see:

- "begin_publication connection event" on page 278
- "end_publication connection event" on page 318
- "sp_hook_dbmlsync_validate_download_file" [*MobiLink - Client Administration*]

Custom validation

You can create custom validation logic to determine if a download file should be applied to a remote database. You do this with the sp_hook_dbmlsync_validate_download_file stored procedure. With this stored procedure, you can both reject a download file and override the default checking of the generation number.

You can use the dbmlsync -be option to embed a string in the file. You use the -be option against the file-definition database when you create the download file This string is passed to the sp_hook_dbmlsync_validate_download_file through the #hook_dict table, and can be used in your validation logic.

For more information, see "sp_hook_dbmlsync_validate_download_file" [*MobiLink - Client Administration*].

File-based download examples

This section contains two examples. Each sets up a file-based download synchronization using a consolidated database with only one table. The first is a simple snapshot example and the second is a slightly more involved timestamp-based example.

Snapshot example

This example implements file-based download for snapshot synchronization. It sets up the three databases that are required by the file-based download, and then demonstrates how to download data. This example is presented in such a way that you can either just read through it, or you can cut and paste the text to run the sample.

Create databases for the sample

The following commands create the three databases used in the example: a consolidated database, a remote database, and a file-definition database.

```
dbinit scons.db
dbinit sremote.db
dbinit sfdef.db
```

The following commands start the three databases, create a data source name for MobiLink to use to connect to the consolidated database, and start the MobiLink server.

```
dbeng10 -n sfdef_eng sfdef.db
dbeng10 -n scons_eng scons.db
dbeng10 -n sremote_eng sremote.db
dbdsn -y -w fbd_demo -c "eng=scons_eng;dbf=scons.db;uid=DBA;
   pwd=sql;astart=off;astop=off"
start mlsrv10 -v+ -c "dsn=fbd_demo"
   -zu+ -ot scons.txt
```

Set up the snapshot example consolidated database

In this example, the consolidated database has one table, called T1. After connecting to the consolidated database, you can run the following SQL to create table T1:

```
CREATE TABLE T1 (
    pk    INTEGER PRIMARY KEY,
    c1    INTEGER
);
```

The following code creates a script version called filebased and creates a download script for that script version.

```
CALL ml_add_table_script( 'filebased',
  'T1', 'download_cursor',
     'SELECT pk, c1 FROM T1' );
```

The following code creates a script version called normal and creates upload and download scripts for that script version.


```
CALL ml_add_table_script ( 'normal', 'T1',
  'upload_insert',
  'INSERT INTO T1 VALUES ({ml r.pk}, {ml r.c1})');

CALL ml_add_table_script( 'normal', 'T1',
  'upload_update',
  'UPDATE T1 SET c1 = {ml r.c1} WHERE pk = {ml r.pk} ' );

CALL ml_add_table_script( 'normal', 'T1',
 'upload_delete',
    'DELETE FROM T1 WHERE pk = {ml r.pk}' );

CALL ml_add_table_script( 'normal', 'T1',
  'download_cursor',
     'SELECT pk, c1 FROM T1' );

COMMIT;
```

The following command creates the stored procedure begin_pub and specifies that begin_pub is the begin_publication script for both the "normal" and "filebased" script versions:

```
CREATE PROCEDURE begin_pub (
       INOUT generation_num integer,
             IN    username       varchar(128),
             IN    pubname        varchar(128) )
BEGIN
   SET gnum=1;
END;

CALL ml_add_connection_script(
  'filebased',
  'begin_publication',
        '{ call begin_pub(
      {ml s.generation_number},
      {ml s.username},
      {ml s.publication_name} ) }' );

CALL ml_add_connection_script( 'normal',
  'begin_publication',
        '{ call begin_pub(
      {ml s.generation_number},
      {ml s.username},
      {ml s.publication_name} ) }' );
```

Create the snapshot example remote database

In this example, the remote database also contains one table, called T1. Connect to the remote database and run the following SQL to create the table T1, a publication called P1, and a user called U1. The SQL also creates a subscription for U1 to P1.

```
CREATE TABLE T1 (
    pk    INTEGER PRIMARY KEY,
    c1    INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);


CREATE SYNCHRONIZATION USER U1;
```



```
CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR U1;
```

The following code creates an sp_hook_dbmlsync_validate_download_file hook to implement user-defined validation logic in the remote database:

```
CREATE PROCEDURE sp_hook_dbmlsync_validate_download_file()
BEGIN
    DECLARE udata    varchar(256);
SELECT value
    INTO udata
    FROM #hook_dict
    WHERE name = 'user data';


    IF udata <> 'ok' THEN
    UPDATE #hook_dict
       SET value = 'FALSE'
       WHERE name = 'apply file';
    END IF;
END
```

Create the snapshot example file-definition database

A file-definition database is required in MobiLink systems that use file-based download. This database has the same schema as the remote databases being updated by file-based download, and it contains no data or state information. The file-definition database is used solely to define the structure of the data that is to be included in the download file. One file-definition database can be used for many groups of remote databases, each defined by its own MobiLink group user name.

The following code defines the file-definition database for this sample. It creates a schema that is identical to the remote database, and also creates:

- a publication called P1 that publishes all rows of the T1 table. The same publication name must be used in the file-definition database and the remote databases.
- a MobiLink user called G1. This user represents all the remotes that are to be updated in the file-based download.
- a subscription to the publication.

You must connect to sfdef.db before running this code.

```
CREATE TABLE T1 (
    pk    INTEGER PRIMARY KEY,
    c1    INTEGER
);


CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER G1;
```



```
CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR G1;
```

Prepare for initial synchronization

To prepare your new remote database so that you can apply a download file, you need to either perform a normal synchronization or create the download file with the dbmlsync -bg option. This example shows you how to initialize your new remote database by performing a normal synchronization.

You can perform an initial synchronization of the remote database with the script version called normal that was created earlier:

```
dbmlsync -c "uid=DBA;pwd=sql;eng=sremote_eng;
  dbf=sremote.db" -v+ -e "sv=normal"
```

Demonstrate the snapshot example file-based download

Connect to the consolidated database and insert some data that will be synchronized by file-based download, such as the following:

```
INSERT INTO T1 VALUES( 1, 1 );
INSERT INTO T1 VALUES( 2, 4 );
INSERT INTO T1 VALUES( 3, 9 );
COMMIT;
```

The following command must be run on the computer that holds the file-definition database. It does the following:

- The dbmlsync -bc option creates the download file, and names it *file1.df*.
- The -be option includes the string "OK" in the download file that will be accessible to the sp_dbmlsync_validate_download_file hook.

```
dbmlsync -c
   "uid=DBA;pwd=sql;eng=sfdef_eng;dbf=sfdef.db"
   -v+ -e "sv=filebased" -bc file1.df -be ok -ot fdef.txt
```

To apply the download file, run dbmlsync with the -ba option on the remote database, supplying the name of the download file you want to apply:

```
dbmlsync -c "uid=DBA;pwd=sql;eng=sremote_eng;
   dbf=sremote.db" -v+ -ba file1.df -ot remote.txt
```

The changes are now applied to the remote database. Open Interactive SQL, connect to the remote database, and run the following SQL command to verify that the remote has the data:

```
SELECT * FROM T1
```

Clean up the snapshot example

The following commands stop all three database engines and erase the files.

```
del file1.df
mlstop -h -w
dbstop -y -c eng=sfdef_eng
```



```
dbstop -y -c eng=scons_eng
dbstop -y -c eng=sremote_eng
dberase -y sfdef.db
dberase -y scons.db
dberase -y sremote.db
```

Timestamp-based example

This example implements file-based download for timestamp-based synchronization. It sets up the three databases and then demonstrates how to download data by file. This example is presented in such a way that you can either just read through it, or you can cut and paste the text to run the sample.

Create databases for the sample

The following commands create the three databases used in the example: a consolidated database, a remote database, and a file-definition database.

```
dbinit tcons.db
dbinit tremote.db
dbinit tfdef.db
```

The following commands start the three databases, create a data source name for MobiLink to use to connect to the consolidated database, and start the MobiLink server.

```
dbeng10 -n tfdef_eng tfdef.db
dbeng10 -n tcons_eng tcons.db
dbeng10 -n tremote_eng tremote.db
dbdsn -y -w tfbd_demo -c "eng=tcons_eng;dbf=tcons.db;uid=DBA;
   pwd=sql;astart=off;astop=off"
start mlsrv10 -v+ -c "dsn=tfbd_demo" -zu+ -ot tcons.txt
```

Set up the timestamp example consolidated database

In this example, the consolidated database has one table, called T1. After connecting to the consolidated database, you can run the following code to create T1:

```
CREATE TABLE T1 (
    pk        INTEGER PRIMARY KEY,
    c1        INTEGER,
    last_modified    TIMESTAMP DEFAULT TIMESTAMP
);
```

The following code defines a script version called normal with a minimal number of scripts. This script version is used for synchronizations that do *not* use file-based download.

```
CALL ml_add_table_script( 'normal', 'T1',
   'upload_insert',
      'INSERT INTO T1( pk, c1) VALUES( {ml r.pk}, {ml r.c1} )' );

CALL ml_add_table_script( 'normal', 'T1',
   'upload_update',
   'UPDATE T1 SET c1 = {ml r.c1} WHERE pk = {ml r.pk} ' );

CALL ml_add_table_script( 'normal', 'T1',
   'upload_delete',
      'DELETE FROM T1 WHERE pk = {ml r.pk}' );

CALL ml_add_table_script( 'normal', 'T1',
```



```
   'download_cursor',
      'SELECT pk, c1 FROM T1
      WHERE last_modified >= {ml s.last_table_download}' );
```

The following code sets the generation number for all subscriptions to 1. It is good practice to use generation numbers in case your consolidated database ever becomes lost or corrupted and you need to force an upload.

```
CREATE PROCEDURE begin_pub (
       INOUT generation_num integer,
             IN    username       varchar(128),
             IN    pubname        varchar(128) )
BEGIN
    SET generation_num = 1;
END;

CALL ml_add_connection_script( 'normal',
    'begin_publication',
                '{ call begin_pub(
        {ml s.generation_number},
        {ml s.username},
        {ml s.publication_name},
        {ml s.last_publication_upload},
        {ml s.last_publication_download} ) }' );

COMMIT;
```

The following code defines the script version called filebased. This script version is used to create file-based download.

```
CALL ml_add_connection_script( 'filebased',
    'begin_publication',
                '{ call begin_pub(
        {ml s.generation_number},
        {ml s.username},
        {ml s.publication_name} ) }' );

CALL ml_add_table_script( 'filebased', 'T1',
    'download_cursor',
       'SELECT pk, c1 FROM T1
        WHERE last_modified >= {ml s.last_table_download}' );
```

The following code sets the last download time so that all changes that occurred within the last five days will be included in download files. Any remote that has missed all the download files created in the last five days will have to perform a normal synchronization before being able to apply any more file-based downloads.

```
CREATE PROCEDURE ModifyLastDownloadTimestamp(
                INOUT last_download_timestamp TIMESTAMP,
                IN    ml_username             VARCHAR(128) )
BEGIN
    SELECT dateadd( day, -5, CURRENT TIMESTAMP )
    INTO last_download_timestamp;
END;

CALL ml_add_connection_script( 'filebased',
   'modify_last_download_timestamp',
            'CALL ModifyLastDownloadTimestamp(
      {ml s.last_download}, {ml s.username} )' );

COMMIT;
```


Create the timestamp example remote database

In this example, the remote database also contains one table, called T1. After connecting to the remote database, run the following code to create table T1, a publication called P1, and a user called U1. The code also creates a subscription for U1 to P1.

```
CREATE TABLE T1 (
    pk    INTEGER PRIMARY KEY,
    c1    INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER U1;

CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR U1;
```

The following code defines an sp_hook_dbmlsync_validate_download_file stored procedure. This stored procedure prevents the application of download files that do not have the string "ok" embedded in them.

```
CREATE PROCEDURE sp_hook_dbmlsync_validate_download_file()
BEGIN
    DECLARE udata    varchar(256);

    SELECT value
    INTO udata
    FROM #hook_dict
    WHERE name = 'user data';

    IF udata <> 'ok' THEN
      UPDATE #hook_dict
         SET value = 'FALSE'
         WHERE name = 'apply file';
    END IF;
END
```

Create the timestamp example file-definition database

The following code defines the file-definition database for the timestamp example. It creates a table, a publication, a user, and a subscription for the user to the publication.

```
CREATE TABLE T1 (
    pk    INTEGER PRIMARY KEY,
    c1    INTEGER
);

CREATE PUBLICATION P1 (
    TABLE T1
);

CREATE SYNCHRONIZATION USER G1;

CREATE SYNCHRONIZATION SUBSCRIPTION
TO P1
FOR G1;
```


Prepare for initial synchronization

To prepare your new remote database so that you can apply a download file, you need to either perform a normal synchronization or create the download file with the dbmlsync -bg option. This example shows you how to use -bg.

The following code defines a script version called filebased_init for the consolidated database. This script version has a single begin_publication script.

```
CALL ml_add_table_script(
  'filebased_init', 'T1', 'download_cursor',
     'SELECT pk, c1 FROM T1' );

CALL ml_add_connection_script(
  'filebased_init',
  'begin_publication',
        '{ call begin_pub(
      {ml s.generation_number},
      {ml s.username},
      {ml s.publication_name} ) }' );

COMMIT;
```

The following two command lines create and apply an initial download file using the script version called filebased_init and the -bg option.

```
dbmlsync -c "uid=DBA;pwd=sql;eng=tfdef_eng;dbf=tfdef.db"
  -v+ -e "sv=filebased_init" -bc tfile1.df -be ok -bg
  -ot tfdef1.txt

dbmlsync -c "uid=DBA;pwd=sql;eng=tremote_eng;dbf=tremote.db"
  -v+ -ba tfile1.df -ot tremote.txt
```

Demonstrate the timestamp example file-based download

Connect to the consolidated database and insert some data that will be synchronized by file-based download, such as the following:

```
INSERT INTO T1(pk, c1) VALUES( 1, 1 );
INSERT INTO T1(pk, c1) VALUES( 2, 4 );
INSERT INTO T1(pk, c1) VALUES( 3, 9 );
commit;
```

The following command line creates a download file containing the new data.

```
dbmlsync -c
   "uid=DBA;pwd=sql;eng=tfdef_eng;dbf=tfdef.db"
   -v+ -e "sv=filebased" -bc tfile2.df -be ok -ot tfdef2.txt
```

The following command line applies the download file to the remote database.

```
dbmlsync -c "uid=DBA;pwd=sql;eng=tremote_eng;dbf=tremote.db"
   -v+ -ba tfile2.df -ot tfdef3.txt
```

The changes are now applied to the remote database. Open Interactive SQL, connect to the remote database, and run the following SQL command to verify that the remote has the data:

```
SELECT * FROM T1
```


Clean up the timestamp example

The following commands stop all three database engines and then erase the files.

```
del file1.df
mlstop -h -w
dbstop -y -c eng=tfdef_eng
dbstop -y -c eng=tcons_eng
dbstop -y -c eng=tremote_eng
dberase -y tfdef.db
dberase -y tcons.db
dberase -y tremote.db
```

Part II. MobiLink Events

This part describes how to write scripts for MobiLink events.

CHAPTER 9

Writing Synchronization Scripts

Contents

Introduction to synchronization scripts

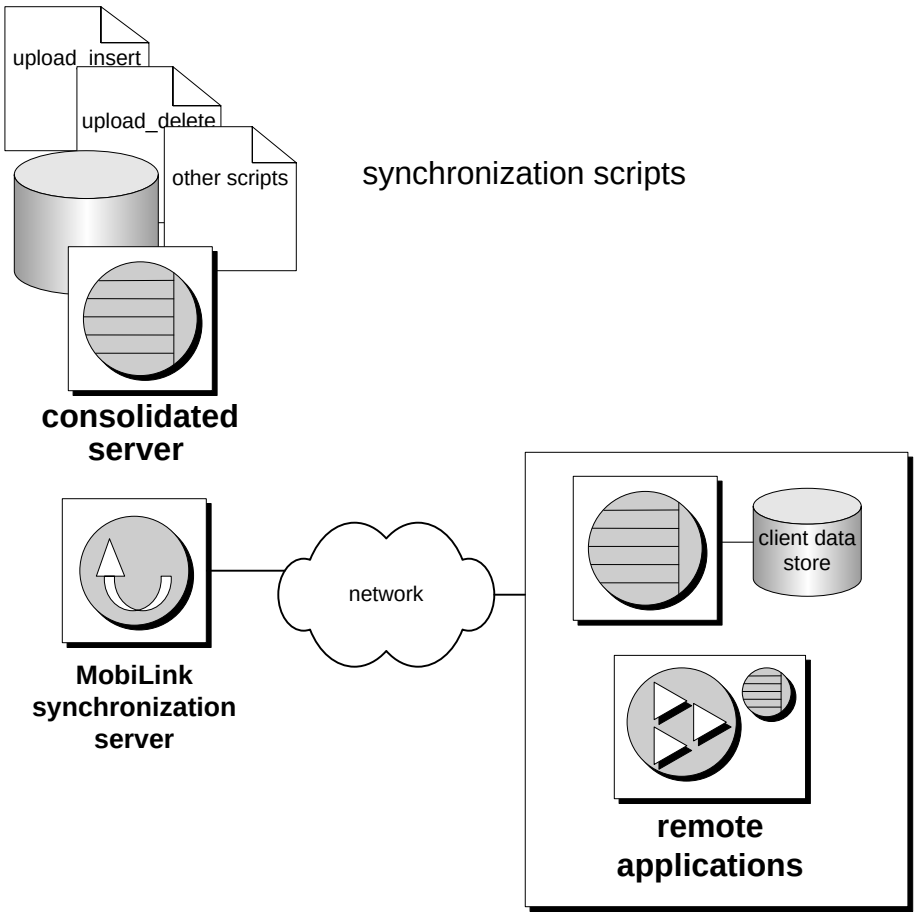
You control the synchronization process by writing synchronization scripts and storing or referencing them in MobiLink system tables in the consolidated database. You can write scripts in SQL, Java, or .NET.

MobiLink synchronization logic is specified with synchronization scripts. Scripts define:

- how data that is uploaded from the remote database should be applied to the consolidated database
- what data should be downloaded from the consolidated database

Scripts can be individual statements or stored procedure calls. They are stored or referenced in your consolidated database. To add scripts to the consolidated database, you can use Sybase Central or you can use system procedures.

During synchronization, the MobiLink server reads the scripts and executes them against the consolidated database.



The synchronization process has multiple steps. A unique event identifies each step. You control the synchronization process by writing scripts associated with some of these events. You write a script only when some particular action must occur at a particular event. The MobiLink server executes each script when its associated event occurs. If you do not define a script for a particular event, the MobiLink server simply proceeds to the next step.

For example, one event is begin_upload_rows. You can write a script and associate it with this event. The MobiLink server reads this script when it is first needed, and executes it during the upload phase of synchronization. If you write no script, the MobiLink server proceeds immediately to the next step, which is processing the uploaded rows.

Some scripts, called table scripts, are associated not only with an event, but also with a particular table in the remote database. The MobiLink server performs some tasks on a table-by-table basis; for example, downloading rows. You can have many scripts associated with the same event, but each with different application tables. Alternatively, you can define many scripts for some application tables, but none for others.

For an overview of events, see "The synchronization process" [*MobiLink - Getting Started*].

For a description of every script you can write, see "Synchronization Events" on page 239.

You can write scripts in SQL, Java, or .NET. This chapter applies to all kinds of scripts, but focuses on how to write synchronization scripts in SQL.

For a description and comparison of SQL, Java, and .NET, see "Options for writing synchronization logic" [*MobiLink - Getting Started*].

For information about writing scripts in .NET, see "Writing Synchronization Scripts in .NET" on page 461.

For information about writing scripts in Java, see "Writing Synchronization Scripts in Java" on page 415.

For information about how to implement synchronization scripts, see "Synchronization Techniques" on page 95.

Simple synchronization script

MobiLink provides many events that you can exploit, but it is not mandatory to provide scripts for each event. In a simple synchronization model, you may need only a few scripts.

Downloading all the rows from the table to each remote database synchronizes the ULProduct table in the CustDB sample application. In this case, no additions are permitted at the remote databases. You can implement this simple form of synchronization with a single script; in this case only one event has a script associated with it.

The MobiLink event that controls the rows to be downloaded during each synchronization is named the download_cursor event. Cursor scripts must contain SELECT statements. The MobiLink server uses these queries to define a cursor. In the case of a download_cursor script, the cursor selects the rows to be downloaded to one particular table in the remote database.

In the CustDB sample application, there is a single download_cursor script for the ULProduct table in the sample application, which consists of the following query:


```
SELECT prod_id, price, prod_name
FROM ULProduct
```

This query generates a result set. The rows that make up this result set are downloaded to the client. In this case, all the rows of the table are downloaded.

The MobiLink server knows to send the rows to the ULProduct application table because this script is associated with both the download_cursor event and ULProduct table by the way it is stored in the consolidated database. Sybase Central allows you to make these associations.

In this example, the query selects data from a consolidated table also named ULProduct. The names need not match. You could, instead, download data to the ULProduct application table from any table, or any combination of tables, in the consolidated database by rewriting the query.

You can write more complicated synchronization scripts. For example, you could write a script that downloads only recently modified rows, or one that provides different information to each remote database.

Scripts and the synchronization process

Each script corresponds to a particular event in the synchronization process. You write a script only when some action must occur. All unnecessary events can be left undefined.

The two principal parts of the process are the processing of uploaded information and the preparation of rows for downloading.

The MobiLink server reads and prepares each script once, when it is first needed. The script is then executed whenever the event is invoked.

The sequence of events

For information about the full sequence of MobiLink events, see "Overview of MobiLink events" on page 241.

For the details of upload processing, see "Writing scripts to upload rows" on page 230.

For the details of download processing, see "Writing scripts to download rows" on page 232.

Notes

- MobiLink technology allows multiple clients to synchronize concurrently. In this case, each client uses a separate connection to the consolidated database.
- The begin_connection and end_connection events are independent of any one synchronization as one connection can handle many synchronization requests. These scripts have no parameters. These are examples of connection-level scripts.
- Some events are invoked only once for each synchronization and have a single parameter. This parameter is the user name, which uniquely identifies the MobiLink client that is synchronizing. These are also examples of connection-level scripts.
- Some events are invoked once for each table being synchronized. Scripts associated with these events are called table-level scripts. They provide two parameters. The first is the user name supplied in the call to the synchronization function, and the second is the name of the table in the remote database being synchronized.

  While each table can have its own table scripts, you can also write table-level scripts that are shared by several tables.
- Some events, such as begin_synchronization, occur at both the connection level and the table level. You can supply both connection and table scripts for these events.
- The COMMIT statements illustrate how the synchronization process is broken up into distinct transactions.
- Errors are a separate event that can occur at any point within the synchronization process. Errors are handled using the following script.

  ```
  handle_error( error_code, error_message, user_name, table_name )
  ```


For reference material, including detailed information about each script and its parameters, see "Synchronization Events" on page 239.

Script types

Synchronization scripts can apply to the entire connection or to specific tables.

- **connection-level scripts** These scripts perform actions that are connection-specific or synchronization-specific and that are independent of any one remote table. These scripts are used in conjunction with other scripts when implementing more complex synchronization schemes.

  See "Connection scripts" on page 219.

- **table-level scripts** These scripts perform actions specific to one synchronization and one particular remote table. These scripts are used in conjunction with other scripts when implementing more complex synchronization schemes such as conflict resolution.

  See "Table scripts" on page 219.

Connection scripts

Connection-level scripts control high level events that are not associated with a particular table. Use these events to perform global tasks that are required during every synchronization.

Connection scripts control actions centered on connecting and disconnecting, as well as actions at synchronization-level events such as beginning and ending the upload or download process. Some connection scripts have related table scripts. These connection scripts are always invoked regardless of the tables being synchronized.

You only need to write a connection-level script when some action must occur at a particular event. You may need to create scripts for only a few events. The default action at any event is for the MobiLink server to carry out no actions. Some simple synchronization schemes need no connection scripts.

ml_global script version

To save you from defining the same scripts multiple times, you can define connection-level scripts once and then re-use them. You do this by defining a script version called ml_global.

See "ml_global script version" on page 226.

Table scripts

Table scripts allow actions at specific events relating to the synchronization of a specific table, such as the start or end of uploading rows, resolving conflicts, or selecting rows to download.

The synchronization scripts for a given table can refer to any table (or a combination of tables) in the consolidated database. You can use this feature to fill a particular remote table with data stored in one or more consolidated tables, or to store data uploaded from a single remote table into multiple tables in the consolidated database.

Table names need not match

The names of tables in the remote databases need not match the names of the tables in the consolidated database. The MobiLink server determines which scripts are associated with a table by looking up the remote table name in the ml_table system table.

Script parameters

Most synchronization scripts can receive parameters from the MobiLink server. For details about the parameters you can use in each script, see "Synchronization Events" on page 239.

You can specify parameters in your SQL scripts in one of two ways:

- question marks
- named script parameters

Script parameters represented by question marks

Representing parameters with question marks is an ODBC convention. To use question marks in your MobiLink SQL scripts, place a single question mark in your script for each parameter. The MobiLink server replaces each question mark with the value of a parameter. It substitutes values in the order the parameters appear in the script definition.

The parameters must be in the order specified in the *Synchronization Events* chapter. Some parameters are optional. A parameter is optional only if no subsequent parameters are specified. For example, you must use parameter 1 if you want to use parameter 2.

See "Synchronization Events" on page 239.

Named script parameters

MobiLink provides named parameters that you can use instead of question marks in your scripts. Named parameters have the following advantages:

- Named parameters allow you to specify any subset of the available parameters in any order.
- With the exception of in/out parameters, you can specify the same named parameter more than once within a script.
- When you use named parameters, you can specify the remote ID in your scripts. This is the only way to specify the remote ID in scripts.
- You can create your own named parameters. See "User-defined named parameters" on page 222.

You cannot mix named parameters and question marks in a single script.

There are four types of MobiLink named parameters. To specify a named parameter, you must prefix it with its type, as follows:

Type of named parameter	Prefix	Examples
System parameters.	s.	{ml s.remote_id}
Row parameters. (The column name. If the column contains spaces, enclose it in double quotes or square brackets.)	r.	{ml r.cust_id} {ml r."Column name"}

Type of named parameter	Prefix	Examples
Old row parameters. (Only used in upload_update scripts. If the column name contains spaces, enclose it in double quotes or square brackets.)	o.	{ml o.cust_name} {ml o."Column name"}
Authentication parameters. See "Authentication parameters" on page 223.	a.	{ml a.1}
User-defined parameters. See "User-defined named parameters" on page 222.	u.	{ml u.varname}

To reference a script parameter by name, enclose the parameter in curly braces and prefix it with ml, as in **{ml** *parameter* **}**. For example, **{ml s.action_code}**. The curly brace notation is an ODBC convention.

For convenience, you can enclose a larger section of code in the curly braces, as long as the section of code does not contain any schema names with the same name as a MobiLink script parameter. For example, both of the following upload_insert scripts are valid and equivalent:

```
INSERT INTO t ( id, c0 ) VALUES( {ml r.id}, {ml r.c0} )
```

and

```
{ml  INSERT INTO t ( id, c0 ) VALUES( r.id, r.c0 ) }
```

Note
To use named row parameters when the columns in your remote database were not generated by the MobiLink Create Synchronization Model wizard, you need to use the ml_add_column system procedure to store column information in the consolidated database. See "ml_add_column" on page 532.

User-defined named parameters

You can also define your own parameters. These are especially useful for RDBMSs that don't allow user-defined variables.

User-defined parameters are defined (and set to NULL) when first referenced. They must start with the letter u and a period (**u.**). A user-defined parameter lasts for one synchronization—it is set to NULL at the start of every synchronization. User-defined parameters are in/out.

A typical use of user-defined parameters is to access state information without having to store it in a table (requiring potentially complex joins).

Example

For example, assume you create a stored procedure called MyBSProc that sets a variable called var1 to bs_value:


```
CREATE PROCEDURE MyBSProc(
  IN username (VARCHAR 128), INOUT var1 (VARCHAR 128)
)
begin
    SET var1 = 'bs_value';
end
```

The following begin_connection script defines the user-defined parameter var1 and sets the value to bs_value:

```
CALL ml_add_connection_script (
  'version1',
  'begin_synchronization',
  '{call MyBSProc( {ml s.username}, {ml u.var1} )}' );
```

The following begin_upload script references var1, whose value is bs_value:

```
CALL ml_add_connection_script (
  'version1',
  'begin_upload',
  'update SomeTable set some_column = 123 where some_other_column = {ml
u.var1}' );
```

Assume you have another stored procedure called MyPFDProc that defines its first parameter to in/out. The following prepare_for_download script changes the value of var1 to pfd_value:

```
CALL ml_add_connection_script (
  'version1',
  'prepare_for_download',
  '{call MyPFDProc( {ml u.var1} )}' );
```

The following begin_download script references var1, whose value is now pfd_value:

```
CALL ml_add_connection_script (
  'version1',
  'begin_download',
  'insert into SomeTable values( {ml s.username}, {ml u.var1} )' );
```

Authentication parameters

In MobiLink scripts, authentication parameters are named parameters that are prefaced with the letter a, such as {ml a.1}. The parameters must be numbers starting at 1, with a limit of 255. The values are sent up from MobiLink clients.

When used in the authenticate_* scripts, authentication parameters pass authentication information.

Authentication parameters can be used in all other events (except begin_connection and end_connection) to pass information from MobiLink clients. This technique is a convenient way to do something that you could otherwise do by creating and populating a table.

On SQL Anywhere remotes, you pass the information with the dbmlsync -ap option. On UltraLite remotes, you pass the information with auth_parms and num_auth_parms.

See also

- "Script parameters" on page 221

- dbmlsync: "-ap option" [*MobiLink - Client Administration*]
- UltraLite: "Authentication Parameters synchronization parameter" [*MobiLink - Client Administration*] and "Number of Authentication Parameters parameter" [*MobiLink - Client Administration*]

Example

For UltraLite remote databases, pass the parameters using the num_auth_parms and auth_parms fields in the ul_synch_info struct. num_auth_parms is a count of the number of parameters, from 0 to 255. auth_parms is a pointer to an array of strings. To prevent the strings from being viewed as plain text, the strings are sent in the same way as passwords. If num_auth_parms is 0, set auth_parms to NULL. Following is an example of passing parameters in UltraLite:

```
ul_char * Params[ 3 ] = { UL_TEXT( "param1" ),
   UL_TEXT( "param2" ), UL_TEXT( "param3" ) };

...
info.num_auth_parms = 3;
info.auth_parms = Params;
```

For SQL Anywhere remote databases, you pass authentication parameters using the dbmlsync -ap option, in a comma-separated list. For example, the following command line passes three parameters:

```
dbmlsync -ap "param1,param2,param3"
```

On the server, you reference the scripts in the order in which they were sent up. In this example, the authenticate_parameters script could be:

```
{CALL my_auth_parm( {
   s.auth.status,
   s.remote_id,
   s.username,
   a.1,
   a.2,
   a.3 } ) }
```


Script versions

Scripts are organized into groups called **script versions**. By specifying a particular version, MobiLink clients can select which set of synchronization scripts will be used to process the upload and prepare the download.

For information about how to add a script version to the consolidated database, see "Adding a script version" on page 226.

Application of script versions

Script versions allow you to organize your scripts into sets, which are run under different circumstances. This ability provides flexibility and is especially useful in the following circumstances:

- **Customizing applications** Using a different set of scripts to process information from different types of remote users. For example, you could write a different set of scripts for use when managers synchronize their databases than would be used for other people in the organization. Although you could achieve the same functionality with one set of scripts, these scripts would be more complicated.
- **Upgrading applications** When you want to upgrade a database application, new scripts may be needed because the new version of your application may handle data differently. New scripts are almost always necessary when the remote database changes. It is usually impossible to upgrade all users simultaneously. MobiLink clients can request that a new set of scripts be used during synchronization. Since both old and new scripts can coexist on the server, all users can synchronize no matter which version of your application they are using.
- **Maintaining multiple applications** A single MobiLink server may need to synchronize two entirely different applications. For example, some employees may use a sales application, whereas others require an application designed for inventory control. When two applications require different sets of data, you can create two versions of the synchronization scripts, one version for each application.
- **Setting properties for the script version** You can set properties for your script version that can be referenced from classes in .NET or Java synchronization logic. See "ml_add_property" on page 539.

Assigning version names

A script version name is a string. You specify this name when you add a script to the consolidated database. For example, if you add your scripts with the ml_add_connection_script and the ml_add_table_script stored procedures, the script version name is the first parameter. Alternatively, if you add your scripts using Sybase Central, you are prompted for the version name.

You cannot use the following names for script versions: **ml_sis_1** or **ml_qa_1**. These names are used internally by MobiLink.

> **Caution**
> It is strongly recommended that your script version names do not start with **ml_**. Script versions starting with ml_ are reserved for internal use.

Specifying a version for a synchronization

If no script version is specified at the remote site when synchronization is initiated, the synchronization fails.

ml_global script version

You can create a script version called **ml_global** that is used differently from other script versions. If you create a script version called ml_global, you define it once and then the connection scripts associated with are automatically used in all synchronizations. You never explicitly specify ml_global as a script version.

If you define a script in the ml_global script version and then you define a script for the same event in the script version that you specify for the synchronization, the specified script version is used. Scripts in the ml_global script version are only used if they are not defined in the primary script version that is being synchronized.

The ml_global script version can only contain connection-level scripts. It is not required, and may not be useful if you are using only one script version.

Adding a script version

All scripts are associated with a script version. When working in Sybase Central Admin mode, you must add a version name to your consolidated database before you can add any connection scripts. When adding scripts with system procedures, a new version name is automatically added with the script. In Sybase Central Model mode, only one script version is allowed and it is by default given the same name as the model.

See "Script versions" on page 225.

♦ To add a script version to a database (Sybase Central Admin mode)

1. In Sybase Central, choose Connections ► Connect with MobiLink 10 and connect to the consolidated database.
2. Open the Versions folder.
3. Choose File ► New ► Version.

   The Create Script Version wizard appears.
4. Follow the instructions in the wizard.

♦ To remove a script version from a database (Sybase Central Admin mode)

1. In Sybase Central, choose Connections ► Connect with MobiLink 10 and connect to the consolidated database.
2. Open the Versions folder.
3. Right-click the version name and select Delete.
4. The Confirm Delete dialog appears. Click Yes.

♦ To add a script version to a database (system procedures)

- You can add a script version in the same operation as adding a connection script or table script.

  For more information, see "System procedures to add or delete scripts" on page 532.

Required scripts

When you run the MobiLink server, certain scripts are required. Which scripts are required is determined by whether you are doing a bi-directional, upload-only, or download-only synchronization.

For bi-directional or upload-only synchronization, MobiLink requires at least one of the following table scripts:

- new_row_cursor
- old_row_cursor
- upload_delete
- upload_insert
- upload_new_row_insert
- upload_old_row_insert
- upload_update
- Or, if you are processing the upload by direct row handling, MobiLink requires a script for the handle_UploadData connection event.

For bi-directional or download-only synchronization, MobiLink expects every table in the synchronization to have a download table script (download_cursor or download_delete_cursor). Or, if you are processing the download by direct row handling, MobiLink requires that you specify a handle_DownloadData connection script. Note that this script can be empty and you can process the download in any other event.

By default, if a required script is missing the synchronization aborts. You can override this behavior using the MobiLink server -fr option.

See "-fr option" on page 51.

Adding and deleting scripts

When you use the Create Synchronization Model wizard, scripts are automatically added to the consolidated database when you deploy the model.

When you create synchronization scripts outside of Sybase Central Model mode, you must add them to MobiLink system tables in the consolidated database. In the case of SQL scripts, the entire script is saved in the MobiLink system table. In the case of Java or .NET scripts, the method name is registered in the system table. The method for storing scripts and method names is similar.

See "MobiLink Server System Tables" on page 551.

If you are using Sybase Central, you must add a synchronization version to the database before you can add individual scripts. See "Adding a script version" on page 226.

♦ To add or delete a connection script (Sybase Central Admin mode)

1. In Sybase Central, choose Connections ► Connect with MobiLink 10 and connect to the consolidated database.
2. Open the Connection Scripts folder.
3. To add a connection script, choose File ► New ► Connection Script.

   The Create Connection Script wizard appears. Follow the instructions in the wizard.
4. To delete a connection script, right-click the script name and select Delete.

   The Confirm Delete dialog appears. Click Yes.

♦ To add or delete a table script (Sybase Central Admin mode)

1. In Sybase Central, choose Connections ► Connect with MobiLink 10 and connect to the consolidated database.
2. Open the Synchronized Tables folder.
3. Double-click the table for which you want to add a script.
4. To add a table script, choose File ► New ► Table Script and follow the instructions in the wizard.

   or

   To delete a table script, right-click the script name and select Delete. The Confirm Delete dialog appears. Click Yes.

♦ To add or delete all types of scripts (system procedures)

- You can add scripts to a consolidated database or delete scripts from a consolidated database using stored procedures that are installed when you set up your consolidated database.

  For a description of the stored procedures that you can use to add or delete scripts, see:

- "ml_add_connection_script" on page 533
- "ml_add_table_script" on page 542
- "ml_add_dnet_connection_script" on page 534
- "ml_add_dnet_table_script" on page 535
- "ml_add_java_connection_script" on page 536
- "ml_add_java_table_script" on page 537

Direct inserts of scripts

In most cases, it is recommended that you use stored procedures or Sybase Central to insert scripts into the system tables. However, in some rare cases you may need to use an INSERT statement to directly insert the scripts. For example, older versions of some RDBMSs may have length limitations that make it difficult to use stored procedures.

For a complete description of the MobiLink system tables, see "MobiLink Server System Tables" on page 551.

The format of the INSERT statements that are required to directly insert scripts can be found in the source code for the ml_add_connection_script and ml_add_table_script stored procedures. The source code for these stored procedures is located in the MobiLink setup scripts. There is a different setup script for each supported RDBMS. The setup scripts are all located in *install-dir\MobiLink\setup* and are called:

Consolidated database	Setup file
SQL Anywhere	*syncsa.sql*
Oracle	*syncora.sql*
IBM DB2 UDB	*syncdb2long.sql*
Microsoft SQL Server	*syncmss.sql*
Adaptive Server Enterprise	*syncase.sql*

Writing scripts to upload rows

To upload information contained in your remote database to your consolidated database, you define upload scripts. You write separate scripts to handle rows that are updated, inserted, or deleted at the remote database. A simple implementation would carry out corresponding actions (update, insert, delete) at the consolidated database.

The MobiLink server uploads data in a single transaction. For a description of the upload process, see "Events during upload" on page 246.

For techniques for uploading rows in .NET synchronization logic, see "Uploading or downloading rows" on page 472.

Notes

- The begin_upload and end_upload scripts for each remote table hold logic that is independent of the individual rows being updated.
- The upload consists of single row inserts, updates, and deletes. These actions are typically performed using upload_insert, upload_update and upload_delete scripts.
- To prepare the upload for SQL Anywhere clients, the dbmlsync utility requires access to all transaction logs written since the last successful synchronization. See "Transaction log files" [*MobiLink - Client Administration*].

Writing upload_insert scripts

The MobiLink server uses this event during processing of the upload to handle rows inserted into the remote database. The following INSERT statement shows how you use the upload_insert statement.

```
INSERT INTO emp (emp_id, emp_name)
VALUES ({ml r.emp_id}, {ml r.emp_name})
```

See "upload_insert table event" on page 393.

Writing upload_update scripts

The MobiLink server uses this event during processing of the upload to handle rows updated at the remote database. The following UPDATE statement illustrates use of the upload_update statement.

```
UPDATE emp
SET emp_name = {ml r.emp_name}
WHERE emp_id = {ml o.emp_id}
```

See "upload_update table event" on page 410.

Writing upload_delete scripts

The MobiLink server uses this event during processing of the upload to handle rows deleted from the remote database. The following statement shows how to use the upload_delete statement.

```
DELETE FROM emp
WHERE emp_id = {ml r.emp_id}
```

See "upload_delete table event" on page 388.

Writing upload_fetch scripts

The upload_fetch script is a SELECT statement that defines a cursor in the consolidated database table. This cursor is used to compare the old values of updated rows, as received from the remote database, against the value in the consolidated database. In this way, the upload_fetch script identifies conflicts when updates are being processed.

Given a synchronized table defined as:

```
CREATE TABLE uf_example (
    pk1 integer NOT NULL,
    pk2     integer NOT NULL,
    val     varchar(200),
    PRIMARY KEY( pk1, pk2 ))
```

Then one possible upload_fetch script for this table is:

```
SELECT pk1, pk2, val
FROM uf_example
WHERE pk1 = {ml r.pk1} and pk2 = {ml r.pk2}
```

See "upload_fetch table event" on page 390.

The MobiLink server requires the WHERE clause of the query in the upload_fetch script to identify exactly one row in the consolidated database to be checked for conflicts.

Writing scripts to download rows

There are two scripts that can be used for processing each table during the download transaction. These are the download_cursor script, which carries out inserts and updates, and the download_delete_cursor script, which carries out deletes.

These scripts are either SELECT statements or calls to procedures that return result sets. The MobiLink server downloads the result set of the script to the remote database. The MobiLink client automatically inserts or updates rows based on the download_cursor script result set, and deletes rows based on the download_delete_cursor event.

For more information about using stored procedures, see "Downloading a result set from a stored procedure call" on page 128.

The MobiLink server downloads data in a single transaction. For a description of the download process, see "Events during download" on page 249.

Notes

- Like the upload, the download starts and ends with connection events. Other events are table-level events.
- If you change the SendDownloadAck setting to ON, if no confirmation of the download is received from the client, the entire download transaction is rolled back in the consolidated database. (By default, SendDownloadAck is set to OFF.)

  See "SendDownloadACK (sa) extended option" [*MobiLink - Client Administration*] and "Send Download Acknowledgment synchronization parameter" [*MobiLink - Client Administration*].
- The begin_download and end_download scripts for each remote table hold logic that is independent of the individual rows being updated.
- For timestamp-based downloads, you specify the last_download_timestamp parameter to ensure that only changes since the last synchronization are downloaded. For example, the download_cursor or download_delete_cursor SQL script could include the line:

  ```
  WHERE Customer.last_modified >= {ml s.last_table_download}
  ```

  See "Using last download times in scripts" on page 98.
- The download does not distinguish between inserts and updates. The script associated with the download_cursor event is a SELECT statement that defines the rows to be downloaded. The client detects whether the row exists or not and carries out the appropriate insert or update operation.
- At the end of the download processing, the client automatically deletes rows if necessary to avoid referential integrity violations.

  See "Referential integrity and synchronization" [*MobiLink - Getting Started*].

Writing download_cursor scripts

You write download_cursor scripts to download information from the consolidated database to your remote database. You must write one of these scripts for each table in the remote database for which you want to download changes. You can use other scripts to customize the download process, but no others are necessary.

- Each download_cursor script must contain a SELECT statement or a call to a procedure that contains a SELECT statement. The MobiLink server uses this statement to define a cursor in the consolidated database.
- The script must select all columns that correspond to the columns in the corresponding table in the remote database. The columns in the consolidated database can have different names than the corresponding columns in the remote database, but they must be of compatible types.
- The columns must be selected in the order that the corresponding columns are defined in the remote database. This order is identical to the order of the columns in the consolidated database.

Example

The following script could serve as a download_cursor script for a remote table that holds employee information. The MobiLink server would use this SQL statement to define the download cursor. This script downloads information about all the employees.

```
SELECT emp_id, emp_fname, emp_lname
FROM employee
```

The MobiLink server passes specific parameters to some scripts. To use these parameters, you can use named parameters, or you can include a question mark in your SQL statement. In the latter case, the MobiLink server substitutes the value of the parameter before executing the statement against the consolidated database. The following script shows how you can use named parameters:

```
CALL ml_add_table_script(
   'Lab',
   'ULOrder',
   'download_cursor',
   'SELECT o.order_id, o.cust_id, o.prod_id, o.emp_id, o.disc, o.quant,
o.notes, o.status
         FROM ULOrder o
         WHERE o.last_modified >= {ml s.last_table_download}
               AND o.emp_name = {ml s.username}' )
```

Notes

- Row values can be selected from a single table or from a join of multiple tables.
- The script itself need not include the name of the remote table. The remote table need not have the same name as the table in the consolidated database. The name of the remote table is identified by an entry in the ml_table MobiLink system table. In Sybase Central, the remote tables are listed together with their scripts.
- The rows in the remote table must contain the values of emp_id, emp_fname, and emp_lname. The remote columns must be in that order, although they can have different names. The columns in the remote database are in the same order as those in the reference database.

- All cursor scripts must select the columns in the same order as the columns are defined in the remote database. Where column names or table structure is different in the consolidated database, columns should be selected in the correct order for the remote database, or equivalently, the reference database. Columns are assigned to columns in the remote database based on their order in the SELECT statement.
- When you build an UltraLite application, the UltraLite generator creates a sample download script for each table in your UltraLite application. It inserts these sample scripts into your reference database. The example scripts assume that the consolidated database contains the same tables as your application. You must modify the sample scripts if your consolidated database differs in design, but these scripts provide a starting point.

See also

- "download_cursor table event" on page 294
- "Partitioning rows among remote databases" on page 102
- "Writing download_delete_cursor scripts" on page 234

Writing download_delete_cursor scripts

You write download_delete_cursor scripts to delete rows from your remote database. You must write one of these scripts for each table in the remote database from which you want to delete rows during synchronization.

You cannot just delete rows from the consolidated database and have them disappear from remote databases. You need to keep track of the primary keys for deleted rows, so that you can select those primary keys with your download_delete_cursor. There are two common techniques for achieving this:

- **Logical deletes** Do not physically delete the row in the consolidated database. Instead, have a status column that keeps track of whether rows are valid. This simplifies the download_delete_cursor. However, the download_cursor and other applications may need to be modified to recognize and use the status column. If you have a last modified column that holds the time of deletion, and if you also keep track of the last download time for each remote, then you can physically delete the row once all remote download times are newer than the time of deletion.
- **Shadow table** For each table for which you want to track deletes, create a shadow table with two columns, one holding the primary key for the table, and the other holding a timestamp. Create a trigger that inserts the primary key and timestamp into the shadow table whenever a row is deleted. Your download_delete_cursor can then select from this shadow table. As with logical deletes, you can delete the row from the shadow table once all remote databases have downloaded the corresponding data.

The MobiLink server deletes rows in the remote database by selecting primary key values from the consolidated database and passing those values to the remote database. If the values match those of a primary key in the remote database, then that row is deleted.

- Each download_delete_cursor script must contain a SELECT statement or a call to a stored procedure that returns a result set. The MobiLink server uses this statement to define a cursor in the consolidated database.

- This statement must select all the columns that correspond to the primary key columns in the table in the remote database. The columns in the consolidated database can have different names than the corresponding columns in the remote database, but they must be of compatible types.
- The values must be selected in the same order as the corresponding columns are defined in the remote database. That order is the order of the columns in the CREATE TABLE statement used to make the table, not the order they appear in the statement that defines the primary key.
- If you delete a parent record, the child records are automatically deleted as well.

  For more information about deleting child records, see "Referential integrity and synchronization" [*MobiLink - Getting Started*].

While each download_delete_cursor script must select all the column values present in the primary key of the corresponding remote table, it may also select all the other columns. This feature is present only for compatibility with older clients. Selecting the additional columns is less efficient, as the database engine must retrieve more data. Unless the client is of an old design, the MobiLink server discards the extra values immediately. The extra values are downloaded only to older clients.

Deleting all the rows in a table

When MobiLink detects a download_delete_cursor with a row that contains all NULLs, it deletes all the data in the remote table. The number of NULLs in the download_delete_cursor can be the number of primary key columns or the total number of columns in the table.

For example, the following download_delete_cursor SQL script deletes every row in a table in which there are two primary key columns. This example works for SQL Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server databases.

```
SELECT NULL, NULL
```

In IBM DB2 UDB and Oracle consolidated databases, you must specify a dummy table to select NULL. For IBM DB2 UDB 7.1, you can use the following syntax:

```
SELECT NULL, NULL FROM SYSIBM.SYSDUMMY1
```

For Oracle consolidated databases, you can use the following syntax:

```
SELECT NULL, NULL FROM DUAL
```

Examples

The following example is a download_delete_cursor script for a remote table that holds employee information. The MobiLink server uses this SQL statement to define the delete cursor. This script deletes information about all employees who are both in the consolidated and remote databases at the time the script is executed.

```
SELECT emp_id
FROM employee
```

The download_delete_cursor accepts the parameters last_download and ml_username. The following script shows how you can use each parameter to narrow your selection.

```
SELECT order_id
FROM ULOrder
```



```
WHERE last_modified >= {ml s.last_table_download}
    AND status = 'Approved'
    AND user_name = {ml s.username}
```

Note:
For some consolidated databases, you may need to cast to the appropriate data type. See "CAST function [Data type conversion]" [*SQL Anywhere Server - SQL Reference*].

These examples could be inefficient in an organization with many employees. You can make the delete process more efficient by selecting only rows that could be present in the remote database. For example, you could limit the number of rows by selecting only those people who have recently been assigned a new manager. Another strategy is to allow the client application to delete the rows itself. This method is possible only when a rule identifies the unneeded rows. For example, rows might contain a timestamp that indicates an expiry date. Before you delete the rows, use the STOP SYNCHRONIZATION DELETE statement to stop these deletes being uploaded during the next synchronization. Be sure to execute START SYNCHRONIZATION DELETE immediately afterwards if you want other deletes to be synchronized in the normal fashion.

See also

You can use the referential integrity checking built into all MobiLink clients to delete rows in an efficient manner. See "Referential integrity and synchronization" [*MobiLink - Getting Started*].

For more information about using download_delete_cursor scripts, see:

- "download_cursor table event" on page 294
- "download_delete_cursor table event" on page 297
- "Handling deletes" on page 123
- "Temporarily stopping the synchronization of deletes" on page 123
- "STOP SYNCHRONIZATION DELETE statement [MobiLink]" [*SQL Anywhere Server - SQL Reference*]
- "Partitioning rows among remote databases" on page 102
- "Snapshot synchronization" on page 100

Writing scripts to handle errors

An error in a synchronization script occurs when an operation in the script fails while the MobiLink server is executing it. The DBMS returns a SQLCODE to the MobiLink server indicating the nature of the error. Each consolidated database DBMS has its own set of SQLCODE values.

When an error occurs, the MobiLink server invokes the handle_error event. You should provide a connection script associated with this event to handle errors. The MobiLink server passes several parameters to this script to provide information about the nature and context of the error, and requires an output value to tell it how to respond to the error.

Error handling actions

Some actions you may want to take in an error-handling script are:

- Log the error in a separate table.
- Instruct the MobiLink server whether to ignore the error and continue, or rollback the synchronization, or rollback the synchronization and shut down the MobiLink server.
- Send an email message.

For more information, see "handle_error connection event" on page 339.

Reporting errors

Since errors can disrupt the natural progression of the synchronization process, it can be difficult to create a log of errors and their resolutions. The report_error script provides a means of accomplishing this task. The MobiLink server executes this script whenever an error occurs. If the handle_error script is defined, it is executed immediately prior to the reporting script.

The parameters to the report_error script are identical to those of the handle_error script, except that the report_error script can not modify the action code. Since the value of the action code is that returned by the handle_error script, this script can be used to debug error-handling problems.

This script typically consists of an insert statement, which records the values, perhaps with other data, such as the time or date, in a table for later reference. To ensure that this data is not lost, the MobiLink server always runs this script in a separate transaction and automatically commits the changes as soon as this script completes.

See "report_error connection event" on page 367.

Example

The following report_error script, which consists of a single insert statement, adds the script parameters into a table, along with the current date and time. The script does not commit this change because the MobiLink server always does so automatically.

```
INSERT INTO errors
VALUES(
  CURRENT DATE,
  {ml s.action_code},
```



```
        {ml s.error_code},
        {ml s.error_message},
        {ml s.username},
        {ml s.table} );
```

Handling multiple errors in a single SQL statement

ODBC allows multiple errors per SQL statement, and some RDBMSs make use of this feature. Microsoft SQL Server, for example, can have two errors for a single statement. The first is the actual error, and the second is usually an informational message telling you why the current statement has been terminated.

When a single SQL statement causes multiple errors, the handle_error script is invoked once per error. The MobiLink server uses the most severe action code (that is, the numerically greatest) to determine the action to take. The same applies to the handle_error script.

If the handle_error script itself causes a SQL error, then the default action code (3000) is assumed.

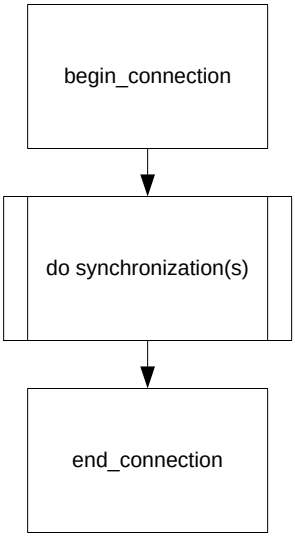
CHAPTER 10

Synchronization Events

Contents

Overview of MobiLink events

When a synchronization request occurs and the MobiLink server decides that a new connection must be created, the begin_connection event is fired and synchronization starts.



Following the synchronization, the connection is placed in a connection pool, and MobiLink again waits for a synchronization request. Before a connection is eventually dropped from the connection pool, the end_connection event is fired. But if another synchronization request for the same version is received, then MobiLink handles the next synchronization request on the same connection. There are a number of events that affect the current synchronization.

Within each synchronization, the following transactions may occur. Each transaction is optional.

- authentication
- begin synchronization
- upload

  You can specify multiple upload transactions with the dbmlsync -tu option.
- prepare for download
- download
- end synchronization

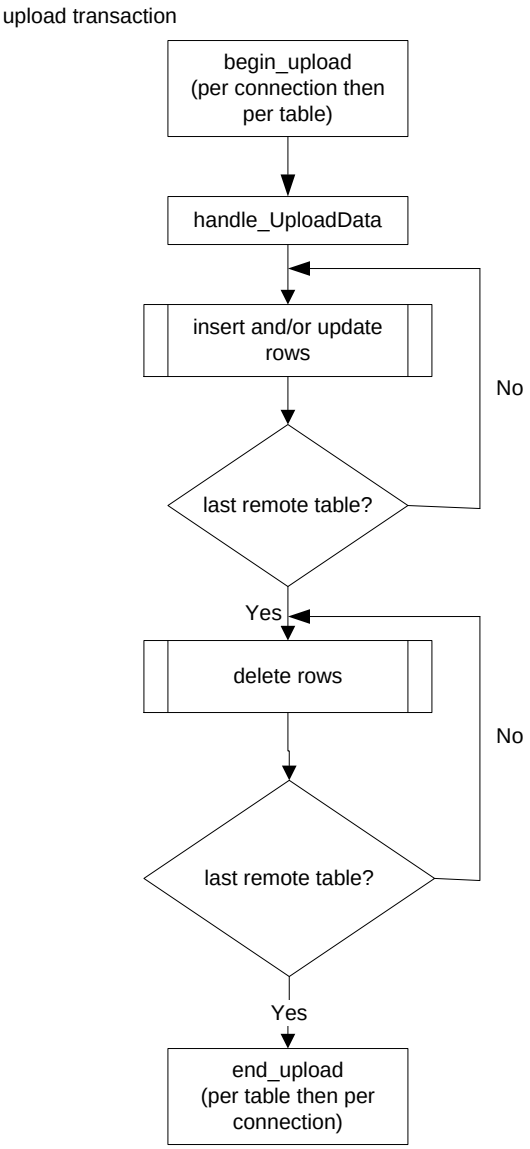
In addition, you can have two connection transactions. A begin connection transaction occurs right after a connection is made, and an end connection transaction occurs when the connection is closed.

The primary phases of a synchronization are the upload and download transactions. The events contained in the upload and download transactions are outlined below.

The upload transaction

The upload transaction applies changes uploaded from a remote database.

The begin_upload event marks the beginning of the upload transaction. The upload transaction is a two-part process. First, inserts and updates are uploaded for all remote tables, and second, deletes are uploaded for all remote tables.



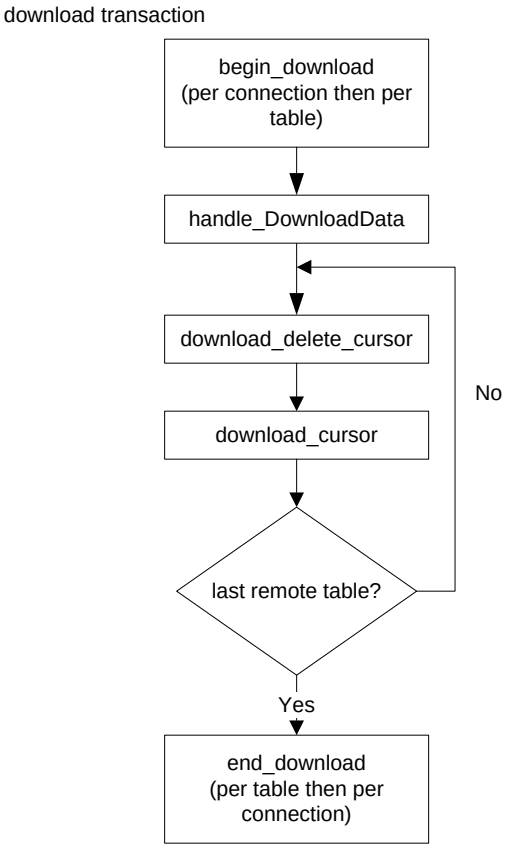
The end_upload event marks the end of the upload transaction.

See “Writing scripts to upload rows” on page 230.

The download transaction

The download transaction fetches rows from the consolidated database. It begins with the begin_download event.

The download transaction is a two-part process. For each table, first deletes are downloaded, and then update/insert rows (upserts) are downloaded. The end_download event ends the download transaction.



See "Writing scripts to download rows" on page 232.

Event overview in pseudocode

The following pseudocode provides an overview of the sequence in which events, and hence the scripts of the same names, are invoked. This representation of the MobiLink event model assumes a full synchronization (not upload-only or download-only) with no errors.

Notes

- In most cases, if you have not defined a script for a given event, the default action is to do nothing.
- The begin_connection and end_connection events are **connection-level events**. They are independent of any single synchronization and have no parameters.

- Some events are invoked once per synchronization for each table being synchronized. Scripts associated with these events are called **table-level scripts**.

  While each table can have its own table scripts, you can also write table-level scripts that are shared by several tables.
- Some events, such as begin_synchronization, occur at both the connection level and the table level. You can supply both connection and table scripts for these events.
- The COMMIT statements illustrate how the synchronization process is broken up into distinct transactions.
- A database error can occur at any point within the synchronization process. Database errors are handled using the handle_error or handle_odbc_error scripts.

Warning
There should be no implicit or explicit commit or rollback in your synchronization scripts or the procedures or triggers that are called from your synchronization scripts. COMMIT or ROLLBACK statements within scripts alter the transactional nature of the synchronization steps. If you use them, you cannot guarantee the integrity of your data in the event of a failure.

MobiLink complete event model

```
-------------------------------------------------------
MobiLink complete event model.

Legend:
- // This is a comment.
- <name>
    The pseudo code for <name> is listed separately
    in a later section, under a banner:
        ------------------------
        name
        ------------------------
- VariableName <- value
    Assign the given value to the given variable name.
    Variable names are in mixed case.
- event_name
    If you have defined a script for the given event name,
    it will be invoked.
-------------------------------------------------------


CONNECT to consolidated database
begin_connection_autocommit
begin_connection
COMMIT
for each synchronization request with
     the same script version {
  <synchronize>
}
end_connection
COMMIT
DISCONNECT from consolidated database
```



```
-----------------------------------------------------
synchronize
-----------------------------------------------------

<authenticate>
<begin_synchronization>
<upload>
<prepare_for_download>
<download>
<end_synchronization>


-----------------------------------------------------
authenticate
-----------------------------------------------------

Status <- 1000
UseDefaultAuthentication <- TRUE
if( authenticate_user script is defined ) {
  UseDefaultAuthentication <- FALSE
  TempStatus <- authenticate_user
  if( TempStatus > Status ) {
    Status <- TempStatus
  }
}

if( authenticate_user_hashed script is defined ) {
  UseDefaultAuthentication <- FALSE
  TempStatus <- authenticate_user_hashed
  if( TempStatus > Status ) {
    Status <- TempStatus
  }
}
  if( authenticate_parameters script is defined )
 {
    TempStatus <- authenticate_parameters
    if( TempStatus > Status ) {
      Status <- TempStatus
  }

if( UseDefaultAuthentication ) {
  if( the user exists in the ml_user table ) {
    if( ml_user.hashed_password column is not NULL ) {
      if( password matches ml_user.hashed_password ) {
        Status <- 1000
      } else {
        Status <- 4000
      }
    } else {
      Status <- 1000
    }
  } else if( -zu+ was on the command line ) {
    Status <- 1000
  } else {
    Status <- 4000
  }
}

if( Status >= 3000 ) {
  // Abort the synchronization.
} else {
  // UserName defaults to MobiLink user name
  // sent from the remote.
  if( modify_user script is defined ) {
```

```
    UserName <- modify_user
    // The new value of UserName is later passed to
    // all scripts that expect the MobiLink user name.
  }
}
COMMIT

-----------------------------------------------------
begin_synchronization
-----------------------------------------------------

begin_synchronization   // Connection event.
for each table being synchronized {
    begin_synchronization    // Call the table level script.
}
for each publication being synchronized {
  begin_publication
}
COMMIT


-----------------------------------------------------
end_synchronization
-----------------------------------------------------

for each publication being synchronized {
  if( begin_publication script was called ) {
    end_publication
  }
}
for each table being synchronized {
  if( begin_synchronization table script was called ) {
    end_synchronization // Table event.
  }
}
if( begin_synchronization table script was called ) {
  end_synchronization     // Connection event.
}
for each table being synchronized {
synchronization_statistics // Table event.
}
synchronization_statistics // Connection event.
for each table being synchronized {
  time_statistics // Table event.
}
time_statistics // Connection event.

COMMIT
```

For the details of upload processing, see “Events during upload” on page 246.

For the details of download processing, see “Events during download” on page 249.

Events during upload

The following pseudocode illustrates how upload events and upload scripts are invoked.

These events take place at the upload location in the complete event model. See “Overview of MobiLink events” on page 241.

Overview of the upload

```
-------------------------------------------------------
upload
-------------------------------------------------------

begin_upload // Connection event
for each table being synchronized {
  begin_upload // Table event
}
  handle_UploadData
  for each table being synchronized {
    begin_upload_rows
    for each uploaded INSERT or UPDATE for this table {
      if( INSERT ) {
        <upload_inserted_row>
      }
      if( UPDATE ) {
        <upload_updated_row>
      }
    }
    end_upload_rows
  }
  for each table being synchronized IN REVERSE ORDER {
    begin_upload_deletes
    for each uploaded DELETE for this table {
      <upload_deleted_row>
    }
    end_upload_deletes
  }

For each table being synchronized {
  if( begin_upload table script is called ) {
    end_upload // Table event
  }
}
if( begin_upload connection script was called ) {
  end_upload // Connection event

  for each table being synchronized {
    upload_statistics  // Table event.
  }
    upload_statistics  // Connection event.

  COMMIT
```

Upload inserts

```
-------------------------------------------------------
<upload_inserted_row>
-------------------------------------------------------
// NOTES:
// - Only table scripts for the current table are involved.

  ConflictsAreExpected <- (
       upload_new_row_insert script is defined
    or upload_old_row_insert script is defined
    or resolve_conflict script is defined )
  if( upload_insert script is defined ) {
    upload_insert
  } else if( ConflictsAreExpected
      and upload_update script is not defined
      and upload_insert script is not defined
      and upload_delete script is not defined ) {
```

```
      // Forced conflict.
      upload_new_row_insert
      resolve_conflict
  } else {
      // Ignore the insert.
  }
```

Upload updates

```
-----------------------------------------------------
upload_updated_row
-----------------------------------------------------
// NOTES:
// - Only table scripts for the current table are involved.
// - Both the old (original) and new rows are uploaded for
//   each update.

  ConflictsAreExpected <- (
       upload_new_row_insert script is defined
    or upload_old_row_insert script is defined
    or resolve_conflict script is defined )
  Conflicted <- FALSE
  if( upload_update script is defined ) {
    if( ConflictsAreExpected
      and upload_fetch script is defined ) {
      FETCH using upload_fetch INTO current_row
      if( current_row <> old row ) {
        Conflicted <- TRUE
      }
    }
    if( not Conflicted ) {
      upload_update
    }
  } else if( upload_update script is not defined
      and upload_insert script is not defined
      and upload_delete script is not defined ) {
      // Forced conflict.
      Conflicted <- TRUE
  }
  if( ConflictsAreExpected and Conflicted ) {
    upload_old_row_insert
    upload_new_row_insert
    resolve_conflict
  }
```

Upload deletes

```
-----------------------------------------------------
upload_deleted_row
-----------------------------------------------------
// NOTES:
// - Only table scripts for the current table are involved.

  ConflictsAreExpected <- (
       upload_new_row_insert script is defined
    or upload_old_row_insert script is defined
    or resolve_conflict script is defined )
  if( upload_delete is defined ) {
    upload_delete
  } else if( ConflictsAreExpected
    and upload_update script is not defined
    and upload_insert script is not defined
    and upload_delete script is not defined ) {
```



```
    // Forced conflict.
    upload_old_row_insert
    resolve_conflict
  } else {
    // Ignore this delete.
  }
```

Events during download

The following pseudocode provides an overview of the sequence in which download events, and hence the script of the same name, are invoked.

These events take place at the download location in the complete event model provided in "Overview of MobiLink events" on page 241.

```
-------------------------------------------------------
prepare_for_download
-------------------------------------------------------

modify_last_download_timestamp
prepare_for_download
if( modify_last_download_timestamp script is defined
    or prepare_for_download script is defined ) {
    COMMIT
}

-------------------------------------------------------
download
-------------------------------------------------------

begin_download // Connection event.
for each table being synchronized {
   begin_download // Table event.
}
   handle_DownloadData
   for each table being synchronized {
     begin_download_deletes
     for each row in download_delete_cursor {
       if( all primary key columns are NULL ) {
         send TRUNCATE to remote
       } else {
         send DELETE to remote
       }
     }
     end_download_deletes
     begin_download_rows
     for each row in download_cursor {
       send INSERT ON EXISTING UPDATE to remote
     }
     end_download_rows
   }
   modify_next_last_download_timestamp
   for each table being synchronized {
     if( begin_download table script is called ) {
        end_download // Table event
     }
}
if( begin_download connect script is called ) {
   end_download // Connection event
}
```



```
    for each table being synchronized {
      download_statistics   // Table event.
    }
      download_statistics   // Connection event.

COMMIT
```

Notes

- If an acknowledgement is expected, and if no confirmation of the downloads is received from the client, the entire download transaction is rolled back in the consolidated database.

  For SQL Anywhere remotes, see "SendDownloadACK (sa) extended option" [*MobiLink - Client Administration*]. For UltraLite remotes, see "Send Download Acknowledgment synchronization parameter" [*MobiLink - Client Administration*].

- The download stream does not distinguish between inserts and updates. The script associated with the download_cursor event is a SELECT statement that defines the rows to be downloaded. The client detects whether the row exists and carries out the appropriate insert or update operation.

- At the end of the download processing, the client automatically deletes rows that violate referential integrity.

  See "Referential integrity and synchronization" [*MobiLink - Getting Started*].

authenticate_file_transfer connection event

Implements custom authentication for file transfers using the mlfiletransfer utility or the MLFileTransfer method.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.file_authentication_code	INTEGER. Required. This is an INOUT parameter. It indicates the overall success of the authentication. If this value is 1000-1999, file transfer is allowed. If this value is 2000-2999, file transfer is not allowed.	1
s.filename	VARCHAR(128). This optional parameter is the name of the file that is being transferred that is to be authenticated. Do not include a path. The file must be located in the root transfer directory that you specified with the mlsrv10 -ftr option, or in one of the subdirectories that are automatically created.	2
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	3

Remarks

The MobiLink server executes this event before allowing any file transfer using the mlfiletransfer utility or MLFileTransfer method. It is executed after the user has authenticated using regular authentication. If this script is not defined, the file transfer is allowed.

The MLFileTransfer method can only be used by UltraLite clients.

See also

- "Adding and deleting scripts" on page 228
- "-ftr option" on page 52
- "MobiLink file transfer utility [mlfiletransfer]" [*MobiLink - Client Administration*]

- UltraLite: "Using MobiLink file transfers" [*MobiLink - Client Administration*]
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

authenticate_parameters connection event

Receives values from the remote that can be used to authenticate beyond a user ID and password. The values can also be used to arbitrarily customize each synchronization.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.authentication_status	INTEGER. This is an INOUT parameter.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
a.*N* (one or more)	VARCHAR(128). For example, named parameters could be `a.1 a.2`.	3...

Parameter Description

- **authentication_status** The authentication_status parameter is required. It indicates the overall success of the authentication, and can be set to one of the following values:

Returned Value	authentication_status	Description
V <= 1999	1000	Authentication succeeded.
1999 < V <= 2999	2000	Authentication succeeded, but password expiring soon.
2999 < V <= 3999	3000	Authentication failed as password has expired.
3999 < V <= 4999	4000	Authentication failed.
4999 < V <= 5999	5000	Authentication failed as user is already synchronizing.
5999 < V	4000	If the returned value is greater than 5999, MobiLink interprets it as a returned value of 4000 (authentication failed).

- **username** This parameter is the MobiLink user name. VARCHAR(128).
- **remote_ID** The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.

  See "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*].
- **remote_parameters** The number of remote parameters must match the number expected or an error results. An error also occurs if parameters are sent from the client and there is no script for this event.

Remarks

You can send strings (or parameters in the form of strings) from both SQL Anywhere and UltraLite clients. This allows you to have authentication beyond a user ID and password. It also means that you can customize your synchronization based on the value of parameters, and do this in a pre-synchronization phase, during authentication.

The MobiLink server executes this event upon starting each synchronization. It is executed in the same transaction as the authenticate_user event.

You can use this event to replace the built-in MobiLink authentication mechanism with a custom mechanism. You may want to call into the authentication mechanism of your DBMS, or you may want to implement features not present in the MobiLink built-in mechanism.

If the authenticate_user or authenticate_user_hashed scripts are invoked and return an error, this event is not called.

SQL scripts for the authenticate_parameters event must be implemented as stored procedures.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Authentication parameters" on page 223
- "MobiLink Users" [*MobiLink - Client Administration*]
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Custom user authentication" [*MobiLink - Client Administration*]
- "authenticate_user connection event" on page 256
- "authenticate_user_hashed connection event" on page 261
- "begin_synchronization connection event" on page 281
- dbmlsync: "-ap option" [*MobiLink - Client Administration*]
- UltraLite: "Authentication Parameters synchronization parameter" [*MobiLink - Client Administration*] and "Number of Authentication Parameters parameter" [*MobiLink - Client Administration*]

Examples

For UltraLite remote databases, pass the parameters using the num_auth_parms and auth_parms fields in the ul_synch_info struct. num_auth_parms is a count of the number of parameters, from 0 to 255. auth_parms is a pointer to an array of strings. To prevent the strings from being viewed as plain text, the strings are sent in the same way as passwords. If num_auth_parms is 0, set auth_parms to NULL. Following is an example of passing parameters in UltraLite:

```
ul_char * Params[ 3 ] = { UL_TEXT( "param1" ),
   UL_TEXT( "param2" ), UL_TEXT( "param3" ) };
```



```
...
info.num_auth_parms = 3;
info.auth_parms = Params;
```

For SQL Anywhere remote databases, you pass parameters using the dbmlsync -ap option, in a comma-separated list. For example, the following command line passes three parameters:

```
dbmlsync -ap "param1,param2,param3"
```

In this example, the authenticate_parameters script could be:

```
{CALL my_auth_parm( {
   s.auth.status,
   s.remote_id,
   s.username,
   a.1,
   a.2,
   a.3 } ) }
```


authenticate_user connection event

Implements custom user authentication.

Parameters

In the following table, the description indicates the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.authentication_status	INTEGER. This is an INOUT parameter.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.password	VARCHAR(128).	3
s.new_password	VARCHAR(128).	4

Default action

Use MobiLink built-in user authentication mechanism.

Remarks

The MobiLink server executes this event upon starting each synchronization. It is executed in a transaction before the begin_synchronization transaction.

You can use this event to replace the built-in MobiLink authentication mechanism with a custom mechanism. You may want to call into the authentication mechanism of your DBMS, or you may want to implement features not present in the MobiLink built-in mechanism, such as password expiry or a minimum password length.

The parameters used in an authenticate_user event are as follows:

- **authentication_status** The authentication_status parameter is required. It indicates the overall success of the authentication, and can be set to one of the following values:

Returned Value	authentication_status	Description
V <= 1999	1000	Authentication succeeded.

Returned Value	authentication_status	Description
1999 < V <= 2999	2000	Authentication succeeded: password expiring soon.
2999 < V <= 3999	3000	Authentication failed: password expired.
3999 < V <= 4999	4000	Authentication failed.
4999 < V <= 5999	5000	Authentication failed as user is already synchronizing.
5999 < V	4000	If the returned value is greater than 5999, MobiLink interprets it as a returned value of 4000.

- **username** This optional parameter is the MobiLink user name.

  See "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*].

- **remote_id** The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.

- **password** This optional parameter indicates the password for authentication purposes. If the user does not supply a password, this is NULL.

- **new_password** This optional parameter indicates a new password. If the user does not change their password, this is NULL.

SQL scripts for the authenticate_user event must be implemented as stored procedures.

When the two authentication scripts are both defined, and both scripts return different authentication_status codes, the higher value is used.

The authenticate_user script is executed in a transaction along with all authentication scripts. This transaction always commits.

There are predefined scripts that you can use for the authenticate_user event to simplify authentication using LDAP, IMAP and POP3 servers.

See "Authenticating to external servers" [*MobiLink - Client Administration*].

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "MobiLink Users" [*MobiLink - Client Administration*]
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Custom user authentication" [*MobiLink - Client Administration*]
- "Authenticating to external servers" [*MobiLink - Client Administration*]
- "authenticate_user_hashed connection event" on page 261
- "authenticate_parameters connection event" on page 253
- "modify_user connection event" on page 363

- "begin_synchronization connection event" on page 281

SQL example

A typical authenticate_user script is a call to a stored procedure. The order of the parameters in the call must match the order above. The following example uses ml_add_connection_script to assign the event to a stored procedure called my_auth.

```
CALL ml_add_connection_script(
   'ver1', 'authenticate_user', 'call my_auth ( {ml s.username} )'
)
```

The following SQL Anywhere stored procedure uses only the user name to authenticate—it has no password check. The procedure ensures only that the supplied user name is one of the employee IDs listed in the ULEmployee table.

```
CREATE PROCEDURE my_auth( in @user_name varchar(128) )
BEGIN
  IF EXISTS
  ( SELECT * FROM ulemployee
    WHERE emp_id = @user_name )
  THEN
    MESSAGE 'OK' type info to client;
    RETURN 1000;
  ELSE
    MESSAGE 'Not OK' type info to client;
    RETURN 4000;
  END IF
END
```

Java example

The following call to a MobiLink system procedure registers a Java method called authenticateUser as the script for the authenticate_user event when synchronizing the script version ver1. This syntax is for SQL Anywhere consolidated databases.

```
CALL ml_add_java_connection_script(
   'ver1', 'authenticate_user',
   'ExamplePackage.ExampleClass.authenticateUser'
)
```

Following is the sample Java method authenticateUser. It calls Java methods that check and, if needed, change the user's password.

```
public String authenticateUser(
  ianywhere.ml.script.InOutInteger authStatus,
  String user,
  String pwd,
  String newPwd )
  throws java.sql.sqlException {
  // A real authenticate_user handler would
  // handle more authentication code states.
  _curUser = user;

  if( checkPwd( user, pwd ) ) {
    // Authentication successful.
    if( newPwd != null ) {
      // Password is being changed.
      if( changePwd( user, pwd, newPwd ) ) {
        // Authentication OK and password change OK.
```



```
        // Use custom code.
        authStatus.setValue( 1001 );

      } else {
        // Authentication OK but password
        // change failed. Use custom code.
        java.lang.System.err.println( "user: "
         + user + " pwd change failed!" );
        authStatus.setValue( 1002 );
      }
    } else {
      authStatus.setValue( 1000 );
    }
  } else {
    // Authentication failed.
    authStatus.setValue( 4000 );
  }
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called AuthUser as the script for the authenticate_user connection event when synchronizing the script version ver1. This syntax is for SQL Anywhere consolidated databases.

```
CALL ml_add_dnet_connection_script(
   'ver1', 'authenticate_user',
   'TestScripts.Test.AuthUser'
)
```

Following is the sample .NET method AuthUser. It calls .NET methods that check and, if needed, change the user's password.

```
public string AuthUser(
  ref int authStatus,
  string user,
  string pwd,
  string newPwd ) {
  // A real authenticate_user handler would
  // handle more authentication code states.
  _curUser = user;
  if( CheckPwd( user, pwd ) ) {
    // Authentication successful.
    if( newPwd != null ) {
      // Password is being changed.
      if( ChangePwd( user, pwd, newPwd ) ) {
        // Authentication OK and password change OK.
        // Use custom code.
        authStatus = 1001;
      } else {
        // Authentication OK but password
        // change failed. Use custom code.
        System.Console.WriteLine( "user: "
         + user + " pwd change failed!" );
        authStatus = 1002;
      }
    } else {
      authStatus = 1000 ;
    }
  } else {
    // Authentication failed.
    authStatus = 4000;
```



```
    }
    return ( null );
  }
```

For a more detailed example of an authenticate_user script written in C# in .NET, see ".NET synchronization example" on page 476.

authenticate_user_hashed connection event

Implements a custom user authentication mechanism.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.authentication_status	INTEGER. This is an INOUT parameter.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.hashed_password	BINARY(20). If the user does not supply a password, this is value NULL.	3
s.hashed_new_password	BINARY(20). If this event is not being used to change the user's password, this value is NULL.	4

Default action

Use MobiLink built-in user authentication mechanism.

Remarks

This event is identical to authenticate_user except for the passwords, which are in the same hashed form as those stored in the ml_user.hashed_password column. Passing the passwords in hashed form provides increased security.

A one-way hash is used. A one-way hash takes a password and converts it to a byte sequence that is (essentially) unique to each possible password. The one-way hash lets password authentication take place without having to store the actual password in the consolidated database.

This script can be called multiple times during an authentication sequence for a user.

When authenticate_user and authenticate_user_hashed are both defined, and both scripts return different authentication_status codes, the higher value is used.

See also

- "Script parameters" on page 221

- "Adding and deleting scripts" on page 228
- "MobiLink Users" [*MobiLink - Client Administration*]
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Custom user authentication" [*MobiLink - Client Administration*]
- "authenticate_user connection event" on page 256
- "authenticate_parameters connection event" on page 253

SQL example

A typical authenticate_user_hashed script is a call to a stored procedure. The order of the parameters in the call must match the order above. The following example calls ml_add_connection_script to assign the event to a stored procedure called my_auth.

```
CALL ml_add_connection_script(
   'ver1', 'authenticate_user_hashed',
   'call my_auth (
      {ml s.authentication_status},
      {ml s.username},
      {ml s.hashed_password})'
)
```

The following SQL Anywhere stored procedure uses both the user name and password to authenticate. The procedure ensures only that the supplied user name is one of the employee IDs listed in the ULEmployee table. The procedure assumes that the Employee table has a binary(20) column called hashed_pwd.

```
CREATE PROCEDURE my_auth(
  inout @authentication_status integer,
  in @user_name varchar(128),
  in @hpwd binary(20) )
BEGIN
  IF EXISTS
  ( SELECT * FROM ulemployee
    WHERE emp_id = @user_name
      and hashed_pwd = @hpwd )
  THEN
    message 'OK' type info to client;
    RETURN 1000;
  ELSE
    message 'Not OK' type info to client;
    RETURN 4000;
  END IF
END
```

Java example

The following call to a MobiLink system procedure registers a Java method called authUserHashed as the script for the authenticate_user_hashed event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1', 'authenticate_user_hashed',
   'ExamplePackage.ExampleClass.authUserHashed')
```

Following is the sample Java method authUserHashed. It calls Java methods that check and, if needed, change the user's password.

```
public String authUserHashed(
  ianywhere.ml.script.InOutInteger authStatus,
  String user,
```



```
  byte pwd[],
  byte newPwd[] )
  throws java.sql.SQLException {
  // A real authenticate_user_hashed handler
  // would handle more auth code states.
  _curUser = user;
  if( checkPwdHashed( user, pwd ) ) {
    // Authorization successful.
    if( newPwd != null ) {
      // Password is being changed.

      if( changePwdHashed( user, pwd, newPwd ) ) {
        // Authorization OK and password change OK.
        // Use custom code.
        authStatus.setValue( 1001 );
      } else {
        // Auth OK but password change failed.
        // Use custom code
        java.lang.System.err.println( "user: " + user
          + " pwd change failed!" );
        authStatus.setValue( 1002 );
      }

      } else {
        authStatus.setValue( 1000 );
      }
    } else {
      // Authorization failed.
      authStatus.setValue( 4000 );
    }
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called AuthUserHashed as the script for the authenticate_user_hashed connection event when synchronizing the script version ver1. This syntax is for SQL Anywhere consolidated databases.

```
CALL ml_add_dnet_connection_script(
   'ver1',
   'authenticate_user_hashed',
   'TestScripts.Test.AuthUserHashed'
)
```

Following is the sample .NET method AuthUserHashed.

```
public string AuthUserHashed(
  ref int authStatus,
  string user,
  byte[] pwd,
  byte[] newPwd ) {
  // A real authenticate_user_hashed handler
  // would handle more auth code states.
  _curUser = user;
  if( CheckPwdHashed( user, pwd ) ) {
    // Authorization successful.
    if( newPwd != null ) {
      // Password is being changed.

      if( ChangePwdHashed( user, pwd, newPwd ) ) {
      // Authorization OK and password change OK.
      // Use custom code.
```



```
            authStatus = 1001;
        } else {
            // Auth OK but password change failed.
            // Use custom code
            System.Console.WriteLine( "user: " + user
              + " pwd change failed!" );
            authStatus = 1002;
        }

            } else {
              authStatus = 1000;
            }
        } else {
            // Authorization failed.
            authStatus = 4000;
        }
        return ( null );
    }
```

begin_connection connection event

Invoked at the time the MobiLink server connects to the consolidated database server.

Parameters

None.

Default action

None.

Remarks

The MobiLink synchronization opens connections on demand as synchronization requests come in. When an application forms or reforms a connection with the MobiLink server, the MobiLink server temporarily allocates one connection with the database server for the duration of that synchronization. This event may not be called if the MobiLink server is using a connection from the pool.

Note
This script is not generally used in Java or .NET, because instead of database variables you would use member variables in this class instance, and prepare the members in the constructor.

See also

- “Adding and deleting scripts” on page 228
- “end_connection connection event” on page 306
- “-cn option” on page 39
- “-w option” on page 75

SQL example

The following SQL script works with a SQL Anywhere consolidated database. Two variables are created, one for the last_download timestamp, and one for employee ID.

```
CALL ml_add_connection_script(
    'custdb',
    'begin_connection',
    'create variable @LastDownload timestamp;
    create variable @EmployeeID integer;')
```


begin_connection_autocommit connection event

Turns on autocommit.

Parameters

None.

Default action

Autocommit is off.

Remarks

When the MobiLink server connects to the consolidated database, it turns off autocommit so that it can roll back the upload and download if an error occurs.

However, if you are using an Adaptive Server Enterprise consolidated database, you cannot perform DDL functions such as creating temporary tables unless autocommit is on. If you are using an Adaptive Server Enterprise consolidated database, run your DDL commands in the begin_connection_autocommit event. When the event is finished, autocommit is turned off.

Begin_connection_autocommit scripts must be written so that they are repeatable. This is because if an error or deadlock occurs, the MobiLink server needs to be able to retry the script (since it can't roll it back).

See also

- "Adding and deleting scripts" on page 228

begin_download connection event

Processes any statements just before the MobiLink server commences preparing the download.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_download	TIMESTAMP. The last download time of any synchronized table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2

Default action

None.

Remarks

The MobiLink server executes this event as the first step in the processing of downloaded information. Download information is processed in a single transaction. The execution of this event is the first action in this transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_download connection event" on page 308
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

The following example calls ml_add_connection_script to assign the event to a stored procedure called SetDownloadParameters.

```
CALL ml_add_connection_script (
   'Lab',
   'begin_download',
```



```
    'CALL SetDownloadParameters( {ml s.last_table_download}, {ml
s.username} )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginDownloadConnection as the script for the begin_download connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'example_ver',
   'begin_download',
   'ExamplePackage.ExampleClass.beginDownloadConnection' )
```

Following is the sample Java method beginDownloadConnection. It calls a Java method (prepDeleteTables) that will prepare the delete tables using a JDBC synchronization that was set earlier.

```
public String beginDownloadConnection(
  Timestamp ts,
  String user )
  throws java.sql.SQLException {
  prepDeleteTables ( _syncConn, ts, user );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginDownload as the script for the begin_download connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'begin_download',
  'TestScripts.Test.BeginDownload'
)
```

Following is the sample .NET method BeginDownload. It calls a .NET method (prepDeleteTables) that will prepare the delete tables using a JDBC synchronization that was set earlier.

```
public string BeginDownload(
  DateTime timestamp,
  string user ) {
  prepDeleteTables ( _syncConn, ts, user );
  return ( null );
}
```


begin_download table event

Processes statements related to a specific table just before preparing the download inserts, updates, and deletions.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applica-ble
s.username	VARCHAR(128). The MobiLink user name.	2
s.table	VARCHAR (128). The table name.	3

Default action

None.

Remarks

The MobiLink server executes this event as the first step in preparing download information for a specific table. The download information is prepared in its own transaction. The execution of this event is the first table-specific action in the transaction.

You can have one begin_download script for each table in the remote database. The script is only invoked when that table is synchronized.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_download table event" on page 310
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

The following call to the MobiLink system procedure ml_add_table_script calls the BeginTableDownload procedure. This syntax is for a SQL Anywhere 10 consolidated database.

```
CALL ml_add_table_script(
   'version1',
   'Leads',
   'begin_download',
   'CALL BeginTableDownLoad(
     {ml s.last_table_download},
     {ml s.username},
     {ml s.table} )' );
```

The following SQL statements create the BeginTableDownload procedure.

```
CREATE PROCEDURE BeginTableDownload(
   LastDownload timestamp,
   MLUser varchar(128),
   TableName varchar(128) )
BEGIN
  EXECUTE IMMEDIATE 'update ' || TableName ||
' set last_download_check = CURRENT TIMESTAMP
  WHERE Owner = ' ||MLUser;
END
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginDownloadTable as the script for the begin_download table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'begin_download',
   'ExamplePackage.ExampleClass.beginDownloadTable' )
```

Following is the sample Java method beginDownloadTable. It saves the name of the current table for use in a later method call.

```
public String beginDownloadTable(
  Timestamp ts,
  String user,
  String table ) {
  _curTable = table;
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginTableDownload as the script for the begin_download table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_download',
  'TestScripts.Test.BeginTableDownload'
)
```

Following is the sample .NET method BeginTableDownload. It saves the name of the current table for use in a later method call.


```
public string BeginTableDownload(
  DateTime timestamp,
  string user,
  string table ) {
  _curTable = table;
  return ( null );
}
```


begin_download_deletes table event

Processes statements related to a specific table just before fetching a list of rows to be deleted from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR (128). The MobiLink user name.	2
s.table	VARCHAR (128). The table name.	3

Default action

None.

Remarks

This event is executed immediately before fetching a list of rows to be deleted from the named table in the remote database.

You can have one begin_download_deletes script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_download_rows table event" on page 275
- "end_download_rows table event" on page 315
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

To minimize the amount of data on remotes, you can use this event to flag data that will be deleted when the download_delete_cursor is executed. The following example flags for deletion sales leads from the remote device that are over 10 weeks old. The example can be used on a SQL Anywhere 10 database.

The following call to a MobiLink system procedure assigns the BeginDownloadDeletes stored procedure to the begin_download_deletes event when synchronizing the script version ver1.

```
CALL ml_add_table_script (
  'ver1',
  'Leads',
  'begin_download_deletes',
  'CALL BeginDownloadDeletes (
    {ml s.last_table_download},
    {ml s.username},
    {ml s.table})' );
```

The following SQL statement creates the BeginDownloadDeletes stored procedure.

```
CREATE PROCEDURE BeginDownloadDeletes(
  LastDownload timestamp,
  MLUser varchar(128),
  TableName varchar(128) )
BEGIN
  execute immediate 'update ' || TableName ||
  ' set delete_flag = 1 where
  days(creation_time, CURRENT DATE) > 70 and Owner = '
  || MLUser;
END;
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginDownloadDeletes as the script for the begin_download_deletes table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'begin_download_deletes',
   'ExamplePackage.ExampleClass.beginDownloadDeletes' )
```

The sample Java method beginDownloadDeletes saves the name of the current table for use in a later method call.

```
public String beginDownloadDeletes (
  Timestamp ts,
  String user,
  String table ) {
  _curTable = table;
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginDownloadDeletes as the script for the begin_download_deletes table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script (
  'ver1', 'table1', 'begin_download_deletes',
```



```
    'TestScripts.Test.BeginDownloadDeletes'
  )
```

The sample .NET method BeginDownloadDeletes saves the name of the current table for use in a later method call.

```
public string BeginDownloadDeletes(
  DateTime timestamp,
  string user,
  string table ) {
  _curTable = table;
  return ( null );
}
```

begin_download_rows table event

Processes statements related to a specific table just before fetching a list of rows to be inserted or updated in the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR (128). The MobiLink user name.	2
s.table	VARCHAR (128). The table name.	3

Default action

None.

Remarks

This event is executed immediately before fetching the stream of rows to be inserted or updated in the named table in the remote database.

You can have one begin_download_rows script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_download_deletes table event" on page 272
- "end_download_deletes table event" on page 312
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

You can use the begin_download_rows table event to flag rows that you no longer want to download for this table. The following example archives sales leads that are over seven days old.

The following call to a MobiLink system procedure registers the BeginDownloadRows stored procedure for the begin_download_rows event.

```
CALL ml_add_table_script(
  'version1',
  'Leads',
  'begin_download_rows',
  'CALL BeginDownloadRows (
    {ml s.last_table_download},
    {ml s.username},
    {ml s.table})' ); )
```

The following SQL statement creates the BeginDowloadRows stored procedure.

```
CREATE PROCEDURE BeginDownloadRows (
  LastDownload timestamp, MLUser varchar(128),
  TableName varchar(128) )
BEGIN
  execute immediate 'update ' || TableName ||
  ' set download_flag = 0 where
  days(creation_time, CURRENT DATE) > 7 and Owner = '
  || MLUser;
END;
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginDownloadRows as the script for the begin_download_rows table event when synchronizing the script version ver1. This syntax is for SQL Anywhere consolidated databases.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'begin_download_rows',
   'ExamplePackage.ExampleClass.beginDownloadRows' )
```

Following is the sample Java method beginDownloadRows. It generates an UPDATE statement using the table and user for MobiLink to execute.

```
public String beginDownloadRows(
  Timestamp ts,
  String user,
  String table ) {
  return( "update " + table + " set download_flag = 0 "
   + " where days(creation_time, CURRENT DATE) > 7 " +
   " and Owner = '" + user + "'" );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginDownloadRows as the script for the begin_download_rows table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1', 'table1', 'begin_download_rows',
  'TestScripts.Test.BeginDownloadRows'
)
```

Following is the sample .NET method BeginDownloadRows. It generates an UPDATE statement using the table and user for MobiLink to execute.


```
public string BeginDownloadRows(
  DateTime timestamp,
  string user,
  string table ) {
  return( "update " + table + " set download_flag = 0 "
   + " where days(creation_time, CURRENT DATE) > 7 " +
   " and Owner = '" + user + "'" );
}
```


begin_publication connection event

Provides useful information about the publication(s) being synchronized. This script may also be used to manage generation numbers for file-based downloads.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

Parameter name for SQL scripts	Description	Order
s.generation_number	INTEGER. This is an INOUT parameter. If your deployment does not use file-based downloads, this parameter can be ignored. The default is 1.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.publication_name	VARCHAR(128)	3
s.last_publication_upload	TIMESTAMP. The time of the last successful upload of this publication.	4
s.last_publication_download	TIMESTAMP. The last download time for the publication.	5
s.subscription_id	VARCHAR(128)	6

Default action

The default generation number is 1. If no script is defined for this event, the generation number sent to the remote will always be 1.

Remarks

This event lets you design synchronization logic based on the publications currently being synchronized. This event is invoked in the same transaction as the begin_synchronization event, and is invoked after the begin_synchronization event. It is invoked once per publication being synchronized.

One potential use for this event is to affect what is downloaded based on the publication used. For example, consider a table that is part of both a priority publication (PriorityPub) and a publication for all tables (AllTablesPub). A script for the begin_publication event could store the publication names in a Java class or a SQL variable or package. Download scripts could then behave differently based on whether the publication being synchronized is PriorityPub or AllTablesPub.

If an UltraLite remote is synchronizing with UL_SYNC_ALL, this event is invoked once with the name 'unknown'.

Generation number

The generation_number parameter is specifically for file-based downloads. The output value of the generation number is passed from the begin_synchronization script to the end_synchronization script. The meaning of the generation_number depends on whether the current synchronization is being used to create a download file, or whether the current synchronization has an upload.

In file-based downloads, generation numbers are used to force an upload before the download. The number is stored in the download file. During a synchronization that has an upload, one generation number is output for every subscription to a publication. They are sent to the remote database in the upload acknowledgement, and stored in SYSSYNC.generation_number.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_publication connection event" on page 318
- "MobiLink File-Based Download" on page 193
- "MobiLink generation numbers" on page 199
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

You may want to record the information for each publication being synchronized. The following example calls ml_add_connection_script to assign the event to a stored procedure called RecordPubSync.

```
CALL ml_add_connection_script(
 'version1',
 'begin_publication',
 '{CALL RecordPubSync(
    {ml s.generation_number},
    {ml s.username},
    {ml s.publication_name},
    {ml s.last_publication_upload},
    {ml s.last_publication_download},
    {ml s.subscription_id} )}' );
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginPublication as the script for the begin_publication connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
 'ver1',
 'begin_publication',
 'ExamplePackage.ExampleClass.beginPublication' )
```

Following is the sample Java method beginPublication. It saves the name of each publication for later use.

```
public String beginPublication(
  ianywhere.ml.script.InOutInteger generation_number,
  String user,
  String pub_name,
  Timestamp last_publication_upload,
  Timestamp last_download ) {
  _publicationNames[ _numPublications++ ] = pub_name;
  return ( null );
}
```


.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginPub as the script for the begin_publication connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
 'ver1',
 'begin_publication',
 'TestScripts.Test.BeginPub'
)
```

Following is the sample .NET method BeginPub. It saves the name of each publication for later use.

```
public string BeginPub(
  ref int generation_number,
  string user,
  string pub_name,
  DateTime last_publication_upload,
  DateTime last_download ) {
  _publicationNames[ _numPublications++ ] = pub_name;
  return ( null );
}
```


begin_synchronization connection event

Processes any statements at the time an application connects to the MobiLink server in preparation for the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1

Default action

None.

Remarks

The MobiLink server executes this event immediately after an application preparing to synchronize has formed a connection with the MobiLink server. It is executed within a separate transaction before the upload transaction.

The begin_synchronization script is useful for maintaining statistics. This is because the end_synchronization script is invoked even if there is an error or conflict, so while the upload transaction is rolled back, things like statistics are maintained.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_synchronization connection event" on page 321
- "begin_synchronization table event" on page 283
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

You may want to store the username value in a temporary table or variable if you will be referencing that value many times in subsequent scripts.

```
CALL ml_add_connection_script (
  'version1',
  'begin_synchronization',
  'set @EmployeeID = {ml s.username}' );
```


Java example

The following call to a MobiLink system procedure registers a Java method called beginSynchronizationConnection as the script for the begin_synchronization connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
 'ver1',
 'begin_synchronization',
 'ExamplePackage.ExampleClass.beginSynchronizationConnection'
)
```

Following is the sample Java method beginSynchronizationConnection. It saves the name of the synchronizing user for later use.

```
public String beginSynchronizationConnection(
  String user ) {
  _curUser = user;
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginSync as the script for the begin_synchronization connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script( 'ver1',
  'begin_synchronization',
  'TestScripts.Test.BeginSync'
)
```

Following is the sample .NET method BeginSync. It saves the name of the synchronizing user for later use.

```
public string BeginSync(
  string user ) {
  _curUser = user;
  return ( null );
}
```


begin_synchronization table event

Processes statements related to a specific table at the time an application connects to the MobiLink server in preparation for the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR (128). The MobiLink user name.	1
s.table	VARCHAR (128). The table name.	2

Default action

None.

Remarks

The MobiLink server executes this event after an application that is preparing to synchronize has formed a connection with the MobiLink server, and after the begin_synchronization connection-level event.

You can have one begin_synchronization script for each table in the remote database. The event is only invoked when the table is synchronized.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_synchronization table event" on page 323
- "begin_synchronization connection event" on page 281
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The begin_synchronization table event is used to set up the synchronization of a particular table. The following SQL script registers a script that creates a temporary table for storing rows during synchronization. This syntax is for a SQL Anywhere consolidated database.


```
CALL ml_add_table_script(
 'ver1',
 'sales_order',
 'begin_synchronization',
 'CREATE TABLE #sales_order (
    id          integer NOT NULL default autoincrement,
    cust_id     integer NOT NULL,
    order_date      date NOT NULL,
    fin_code_id char(2) NULL,
    region      char(7) NULL,
    sales_rep       integer NOT NULL,
    PRIMARY KEY (id),
)' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginSynchronizationTable as the script for the begin_synchronization table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
 'ver1',
 'table1',
 'begin_synchronization',
 'ExamplePackage.ExampleClass.beginSynchronizationTable' )
```

Following is the sample Java method beginSynchronizationTable. It adds the current table name to a list of table names contained in this instance.

```
public String beginSynchronizationTable(
  String user,
  String table ) {
  _tableList.add( table );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginTableSync as the script for the begin_synchronization table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script (
  'ver1',
  'table1',
  'begin_synchronization',
  'TestScripts.Test.BeginTableSync' )
```

Following is the sample .NET method BeginTableSync. It adds the current table name to a list of table names contained in this instance.

```
public string BeginTableSync(
  string user,
  string table ) {
  _tableList.Add( table );
  return ( null );
}
```


begin_upload connection event

Processes any statements just before the MobiLink server commences processing the stream of uploaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR (128). The MobiLink user name.	1

Default action

None.

Remarks

The MobiLink server executes this event as the first step in the processing of uploaded information. Upload information is processed in a single transaction. The execution of this event is the first action in this transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_upload connection event" on page 325
- "begin_upload table event" on page 287
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The begin_upload connection event is used to perform whatever steps you need performed prior to uploading rows. The following SQL script creates a temporary table for storing old and new row values for conflict processing of the sales_order table. This example works with a SQL Anywhere consolidated database.

```
CALL ml_add_connection_script(
 'version1',
 'begin_upload',
 'CREATE TABLE #sales_order_conflicts (
    id          integer NOT NULL default autoincrement,
    cust_id     integer NOT NULL,
    order_date      date NOT NULL,
    fin_code_id char(2) NULL,
    region      char(7) NULL,
    sales_rep       integer NOT NULL,
```



```
    new_value       char(1) NOT NULL,
    PRIMARY KEY (id) )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginUploadConnection as the script for the begin_upload connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
 'ver1',
 'begin_upload',
 'ExamplePackage.ExampleClass.beginUploadConnection ' )
```

Following is the sample Java method beginUploadConnection. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String beginUploadConnection( String user ) {
  java.lang.System.out.println(
    "Starting upload for user: " + user );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginUpload as the script for the begin_upload connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'begin_upload',
  'TestScripts.Test.BeginUpload'
)
```

The following C# example saves the current user name for use in a later event.

```
public string BeginUpload( string curUser ) {
  user = curUser;
  return ( null );
}
```


begin_upload table event

Processes statements related to a specific table just before the MobiLink server commences processing the stream of uploaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

The MobiLink server executes this event as the first step in the processing of uploaded information. Upload information is processed in a separate transaction. The execution of this event is the first table-specific action in this transaction.

You can have one begin_upload script for each table in the remote database. The script is only invoked when the table is actually synchronized.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_upload table event" on page 327
- "begin_upload connection event" on page 285
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

When uploading rows from a remote you may want to place the changes in an intermediate table and manually process changes yourself. You can populate a global temporary table in this event.


```
CALL ml_add_table_script(
 'version1',
 'Leads',
 'begin_upload',
 'INSERT INTO T_Leads
  SELECT * FROM Leads
  WHERE Owner = @EmployeeID' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginUploadTable as the script for the begin_upload table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'begin_upload',
  'ExamplePackage.ExampleClass.beginUploadTable'
 )
```

Following is the sample Java method beginUploadTable. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String beginUploadTable(
  String user,
  String table ) {
  java.lang.System.out.println("Beginning to process upload for: " + table);
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginTableUpload as the script for the begin_upload table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'begin_upload',
  'TestScripts.Test.BeginTableUpload'
)
```

Following is the sample .NET method BeginTableUpload. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string BeginTableUpload(
  string user,
  string table ) {
  System.Console.WriteLine("Beginning to process upload for: " + table);
  return ( null );
}
```


begin_upload_deletes table event

Processes statements related to a specific table just before uploading deleted rows from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

This event occurs immediately before applying the changes that result from rows deleted in the client table named in the second parameter.

You can have one begin_upload_deletes script for each table in the remote database. The script is only invoked when the table is actually synchronized.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_upload_deletes table event" on page 330
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The begin_upload_deletes connection event is used to perform whatever steps you need performed after uploading inserts and updates for a particular table, but before deletes are uploaded for that table. The following SQL script creates a temporary table for storing deletes temporarily during upload. This syntax is for a SQL Anywhere consolidated database.


```
CALL ml_add_table_script(
 'ver1',
 'sales_order',
 'begin_upload_deletes',
 'CREATE TABLE #sales_order_deletes (
    id          integer NOT NULL default autoincrement,
    cust_id     integer NOT NULL,
    order_date  date NOT NULL,
    fin_code_id char(2) NULL,
    region      char(7) NULL,
    sales_rep   integer NOT NULL,
    PRIMARY KEY (id) )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginUploadDeletes as the script for the begin_upload_deletes table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'begin_upload_deletes',
  'ExamplePackage.ExampleClass.beginUploadDeletes' )
```

Following is the sample Java method beginUploadDeletes. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String beginUploadDeletes(
  String user,
  String table )
  throws java.sql.SQLException {
  java.lang.System.out.println( "Starting upload
    deletes for table: " + table );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginUploadDeletes as the script for the begin_upload_deletes table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'begin_upload_deletes',
  'TestScripts.Test.BeginUploadDeletes'
)
```

Following is the sample .NET method BeginUploadDeletes. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string BeginUploadDeletes(
  string user,
  string table ) {
  System.Console.WriteLine(
   "Starting upload deletes for table: " + table );
```



```
    return ( null );
}
```


begin_upload_rows table event

Processes statements related to a specific table just before uploading inserts and updates from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

This event occurs immediately prior to applying the changes that result from inserts and deletes to the client table named in the second parameter.

You can have one begin_upload_rows script for each table in the remote database. The script is only invoked when the table is actually synchronized.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_upload_rows table event" on page 333
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The begin_upload_rows connection event is used to perform whatever steps you need performed before uploading inserts and updates for a particular table. The following script calls a stored procedure that prepares the consolidated database for inserts and updates into the Inventory table:


```
CALL ml_add_table_script(
 'MyCorp 1.0',
 'Inventory',
 'begin_upload_rows',
 'CALL PrepareForUpserts()' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called beginUploadRows as the script for the begin_upload_rows table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'begin_upload_rows',
  'ExamplePackage.ExampleClass.beginUploadRows' )
```

Following is the sample Java method beginUploadRows. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String beginUploadRows(
  String user,
  String table )
  throws java.sql.SQLException {
  java.lang.System.out.println(
   "Starting upload rows for table: " +
   table + " and user: " + user );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called BeginUploadRows as the script for the begin_upload_rows table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'begin_upload_rows',
  'TestScripts.Test.BeginUploadRows'
)
```

Following is the sample .NET method BeginUploadRows. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string BeginUploadRows(
  string user,
  string table ) {
  System.Console.WriteLine(
   "Starting upload rows for table: " +
   table + " and user: " + user );
  return ( null );
}
```


download_cursor table event

Defines a cursor to select rows to download and insert or update in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2

Default action

None.

Remarks

The MobiLink server opens a read-only cursor with which to fetch a list of rows to download to the remote database. This script should contain a suitable SELECT statement.

You can have one download_cursor script for each table in the remote database.

To optimize performance of the download stage of synchronization to UltraLite clients, when the range of primary key values is outside the current rows on the device, you should order the rows in the download cursor by primary key. Downloads of large reference tables, for example, can benefit from this optimization.

Each download_cursor script must contain a SELECT statement or a call to a procedure that contains a SELECT statement. The MobiLink server uses this statement to define a cursor in the consolidated database.

The script must select all columns that correspond to the columns in the corresponding table in the remote database. The columns in the consolidated database can have different names than the corresponding columns in the remote database, but they must be of compatible types.

The columns must be selected in the order that the corresponding columns are defined in the remote database.

Note that download_cursor allows for cascading deletes. Thus, you can delete records from a database.

To avoid downloading unnecessary rows, you should include the following line in the WHERE clause of your download_cursor script:

```
AND last_table_download > '1900/1/1'
```

For Java and .NET applications, this script must return valid SQL.

If you are considering using READPAST table hints in download_cursor scripts because you are doing lots of updates that affect download performance, consider using snapshot isolation for downloads instead. The READPAST table hint can cause problems if used in download_cursor scripts. When using timestamp-based downloads, the READPAST hint can cause rows to be missed, and can cause a row to never be downloaded to a remote database. For example, a row is added to the consolidated database and committed; the row has a last_modified column with a time of yesterday. The same row is updated but not committed. A remote database with a last_download time of last week synchronizes. A download_cursor script attempts to select the row using READPAST, and skips the row. The transaction that updated the row is rolled back. The next last download time for the remote is advanced to today. From this point on, the row will never be downloaded unless it is updated. A possible workaround is to implement a modify_next_last_download_timestamp script and set the last download time to be the start time of the oldest open transaction.

See also

- “Script parameters” on page 221
- “Adding and deleting scripts” on page 228
- “Writing scripts to download rows” on page 232
- “Writing download_cursor scripts” on page 233
- “Partitioning rows among remote databases” on page 102
- “download_delete_cursor table event” on page 297
- “Using remote IDs and MobiLink user names in scripts” [*MobiLink - Client Administration*]
- “Using last download times in scripts” on page 98

SQL example

The following example comes from an Oracle installation, although the statement is valid against all supported databases. This example downloads all rows that have been changed since the last time the user downloaded data, and that match the user name in the emp_name column.

```
CALL ml_add_table_script(
  'Lab',
  'ULOrder',
  'download_cursor',
  'SELECT order_id,
    cust_id,
    prod_id,
    emp_id,
    disc,
    quant,
    notes,
    status
   FROM ULOrder
   WHERE last_modified >= {ml s.last_table_download}
    AND emp_name = {ml s.username}' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called downloadCursor as the script for the download_cursor table event when synchronizing the script version ver1.


```
CALL ml_add_java_table_script(
   'ver1',
   'ULCustomer',
   'download_cursor',
   'ExamplePackage.ExampleClass.downloadCursor ' )
```

Following is the sample Java method downloadCursor. It returns a SQL statement to download rows where the last_modified column is greater than or equal to the last download time.

```
public String downloadCursor(
  java.sql.Timestamp ts,
  String user )  {
  return( "SELECT cust_id, cust_name FROM ULCustomer
             WHERE last_modified >= ' "
   + ts + " ' ");
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called DownloadCursor as the script for the download_cursor table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
 'ver1',
 'table1',
 'download_cursor',
 'TestScripts.Test.DownloadCursor'
)
```

Following is the sample .NET method DownloadCursor. It populates a temporary table with the contents of a file called *rows.txt*. It then returns a cursor that causes MobiLink to send the rows in the temporary table to the remote database. This syntax is valid for SQL Anywhere consolidated databases.

```
public string DownloadCursor(
  DateTime ts,
  string user ) {
  DBCommand stmt   = curConn.CreateCommand();
  StreamReader input  = new StreamReader( "rows.txt" );
  string sql    = input.ReadLine();
  stmt.CommandText = "DELETE FROM dnet_dl_temp";
  stmt.ExecuteNonQuery();
  while( sql != null ){
    stmt.CommandText = "INSERT INTO dnet_dl_temp VALUES " + sql;
    stmt.ExecuteNonQuery();
    sql = input.ReadLine();
  }
  return( "SELECT * FROM dnet_dl_temp" );
}
```


download_delete_cursor table event

Defines a cursor to select rows that are to be deleted in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2

Default action

None.

Remarks

The MobiLink server opens a read-only cursor with which to fetch a list of rows to download, and then insert or update in the remote database. This script must contain a SELECT statement that returns the primary key values of the rows to be deleted from the table in the remote database.

You can have one download_delete_cursor script for each table in the remote database.

If the download_delete_cursor has nulls for the primary key columns for one or more rows in a table, then MobiLink tells the remote to delete all the data in the table. See "Deleting all the rows in a table" on page 235.

Note that rows deleted from the consolidated database will not appear in a result set defined by a download_delete_cursor event, and so are not automatically deleted from the remote database. One technique for identifying rows to be deleted from remote databases is to add a column to the consolidated database table identifying a row as inactive.

To avoid downloading unnecessary rows, you should include the following line in the WHERE clause of your download_delete_cursor script:

```
AND last_modified > '1900/1/1'
```


For Java and .NET applications, this script must return valid SQL.

It can be problematic using READPAST table hints in a download_delete_cursor. For details, see the download_cursor event.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "download_cursor table event" on page 294
- "Writing scripts to download rows" on page 232
- "Partitioning rows among remote databases" on page 102
- "Writing download_delete_cursor scripts" on page 234
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

This example is taken from the Contact sample and can be found in *Samples\MobiLink\Contact \build_consol.sql*. It deletes from the remote database any customers who have been changed since the last time this user downloaded data (`Customer.last_modified >= {ml s.last_table_download}`), and either

- do not belong to the synchronizing user (`SalesRep.username != {ml s.username}`), or
- are marked as inactive in the consolidated database (`Customer.active = 0`).

```
CALL ml_add_table_script(
 'ver1',
 'table1',
 'download_delete_cursor',
 'SELECT cust_id FROM Customer key join SalesRep
  WHERE Customer.last_modified >= {ml s.last_table_download} AND
   ( SalesRep.username != {ml s.username} OR Customer.active = 0 )')
```

Java example

The following call to a MobiLink system procedure registers a Java method called downloadDeleteCursor as the script for the download_delete_cursor event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'download_delete_cursor',
   'ExamplePackage.ExampleClass.downloadDeleteCursor' )
```

Following is the sample Java method downloadDeleteCursor. It calls a Java method that generates the SQL for the download delete cursor.

```
public String downloadDeleteCursor(
  Timestamp ts,
  String user ) {
  return( getDownloadCursor( _curUser, _curTable ) );
}
```


.NET example

The following call to a MobiLink system procedure registers a .NET method called DownloadDeleteCursor as the script for the download_delete_cursor table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'download_delete_cursor',
  'TestScripts.Test.DownloadDeleteCursor'
)
```

Following is the sample .NET method DownloadDeleteCursor. It calls a .NET method that generates the SQL for the download delete cursor.

```
public string DownloadDeleteCursor(
  DateTime timestamp,
  string user ) {
  return( GetDownloadCursor( _curUser, _curTable ) );
}
```


download_statistics connection event

Tracks synchronization statistics for download operations.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name as specified in your SYNCHRONIZATION USER definition.	1
s.warnings	INTEGER. The number of warnings issued.	2
s.errors	INTEGER. The number of errors, including handled errors, that occurred.	3
s.fetched_rows	INTEGER. The number of rows fetched by the download_cursor script.	4
s.deleted_rows	INTEGER. The number of rows fetched by the download_delete_cursor script.	5
s.filtered_rows	INTEGER. The number of rows from the deleted_rows parameter actually sent to the remote. This reflects download filtering of uploaded values.	6
s.bytes	INTEGER. The number of bytes sent to the remote as the download.	7

Default action

None.

Remarks

The download_statistics event allows you to gather, for any user, statistics on downloads. The download_statistics connection script is called just prior to the commit at the end of the download transaction.

Note:
Depending on the command line, not all warnings or errors are logged, so the warnings and errors counts may be more than the number of warnings or errors logged.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "download_statistics table event" on page 303
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405
- "synchronization_statistics connection event" on page 376
- "synchronization_statistics table event" on page 379
- "time_statistics connection event" on page 382
- "time_statistics table event" on page 385
- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts synchronization statistics into a table called download_audit.

```
CALL ml_add_connection_script(
 'ver1',
 'download_statistics',
 'INSERT INTO download_audit(
   user_name,
   warnings,
   errors,
   deleted_rows,
   fetched_rows,
   download_rows,
   bytes )
  VALUES (
   {ml s.username},
   {ml s.warnings},
   {ml s.errors},
   {ml s.fetched_rows},
   {ml s.deleted_rows},
   {ml s.filtered_rows},
   {ml s.bytes})')
```

Once vital statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following call to a MobiLink system procedure registers a Java method called downloadStatisticsConnection as the script for the download_statistics event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'download_statistics',
  'ExamplePackage.ExampleClass.downloadStatisticsConnection' )
```


Following is the sample Java method downloadStatisticsConnection. It prints the number of fetched rows to the MobiLink output log. (Note that printing the number of fetched rows to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String downloadStatisticsConnection(
  String user,
  int warnings,
  int errors,
  int fetchedRows,
  int deletedRows,
  int bytes ) {
  java.lang.System.out.println(
   "download connection stats fetchedRows: "
   + fetchedRows );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called DownloadStats as the script for the download_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'download_statistics',
  'TestScripts.Test.DownloadStats'
)
```

Following is the sample .NET method DownloadStats. It prints the number of fetched rows to the MobiLink output log. (Note that printing the number of fetched rows to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string DownloadStats(
  string user,
  int warnings,
  int errors,
  int deletedRows,
  int fetchedRows,
  int downloadRows,
  int bytes ) {
  System.Console.WriteLine(
   "download connection stats fetchedRows: "
   + fetchedRows );
  return ( null );
}
```


download_statistics table event

Tracks synchronization statistics for download operations by table.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name as specified in your SYNCHRONIZATION USER definition.	1
s.table	VARCHAR(128). The table name.	2
s.warnings	INTEGER. The number of warnings issued.	3
s.errors	INTEGER. The number of errors, including handled errors, that occurred.	4
s.fetched_rows	INTEGER. The number of rows fetched by the download_cursor script.	5
s.deleted_rows	INTEGER. The number of rows fetched by the download_delete_cursor script.	6
s.filtered_rows	INTEGER. The number of rows from (6) actually sent to the remote. This reflects download filtering of uploaded values.	7
s.bytes	INTEGER. The number of bytes sent to the remote as the download.	8

Default action

None.

Remarks

The download_statistics event allows you to gather, for any user and table, statistics on downloads as they apply to that table. The download_statistics table script is called just prior to the commit at the end of the download transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "download_statistics connection event" on page 300
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405
- "synchronization_statistics connection event" on page 376
- "synchronization_statistics table event" on page 379
- "time_statistics connection event" on page 382
- "time_statistics table event" on page 385
- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts synchronization statistics into a table called download_audit. Once vital statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

```
CALL ml_add_table_script(
 'ver1',
 'table1',
 'download_statistics',
 'INSERT INTO download_audit (
   user_name,
   table, warnings,
   errors,
   deleted_rows,
   fetched_rows,
   download_rows,
   bytes)
  VALUES (
   {ml s.username},
   {ml s.table},
   {ml s.warnings},
   {ml s.errors},
   {ml s.fetched_rows},
   {ml s.deleted_rows},
   {ml s.filtered_rows},
   {ml s.bytes})')
```

Java example

The following call to a MobiLink system procedure registers a Java method called downloadStatisticsTable as the script for the download_statistics table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'download_statistics',
  'ExamplePackage.ExampleClass.downloadStatisticsTable' )
```

Following is the sample Java method downloadStatisticsTable. It prints some statistics for this table to the MobiLink output log. (Note that printing statistics for a table to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String downloadStatisticsTable(
  String user,
```



```
  String table,
  int warnings,
  int errors,
  int fetchedRows,
  int deletedRows,
  int bytes ) {
  java.lang.System.out.println( "download table stats "
   + "table: " + table + "bytes: " + bytes );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called DownloadTableStats as the script for the download_statistics table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
 'ver1',
 'table1',
 'download_statistics',
 'TestScripts.Test.DownloadTableStats'
)
```

Following is the sample .NET method DownloadTableStats. It prints some statistics for this table to the MobiLink output log. (Note that printing statistics for a table to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string DownloadTableStats(
  string user,
  string table,
  int warnings,
  int errors,
  int deletedRows,
  int fetchedRows,
  int downloadRows,
  int bytes ) {
  System.Console.WriteLine( "download table stats "
   + "table: " + table + "bytes: " + bytes );
  return ( null );
}
```


end_connection connection event

Processes any statements just before the MobiLink server closes a connection with the consolidated database server, either in preparation to shut down or when a connection is removed from the connection pool.

Parameters

None.

Default action

None.

Remarks

You can use the end_connection script to perform an action of your choice just prior to closing of a connection between the MobiLink server and the consolidated database server.

This script is normally used to complete any actions started by the begin_connection script and free any resources acquired by it.

See also

- "begin_connection connection event" on page 265
- "Adding and deleting scripts" on page 228

SQL example

The following SQL script drops a temporary table that was created by the begin_connection script. This syntax is for a SQL Anywhere consolidated database. Strictly speaking, this table doesn't need to be dropped explicitly, since SQL Anywhere will do this automatically when the connection is destroyed. Whether or not a temporary table needs to be dropped explicitly depends on your consolidated database type.

```
CALL ml_add_connection_script(
 'version 1.0',
 'end_connection',
 'DROP TABLE #sync_info' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called endConnection as the script for the end_connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'end_connection',
   'ExamplePackage.ExampleClass.endConnection' )
```

Following is the sample Java method endConnection. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String endConnection() {
  java.lang.System.out.println( "Ending connection." );
  return ( null );
}
```


.NET example

The following call to a MobiLink system procedure registers a .NET method called EndConnection as the script for the end_connection connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'end_connection',
  'TestScripts.Test.EndConnection'
)
```

Following is the sample .NET method EndConnection. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string EndConnection() {
  System.Console.WriteLine( "Ending connection." );
  return ( null );
}
```


end_download connection event

Processes any statements just after the MobiLink server concludes preparation of the download data.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_download	TIMESTAMP. The last download times of any synchronized table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2

Default action

None.

Remarks

The MobiLink server executes this script after all rows have been downloaded and, if expecting a download acknowledgement, confirmation of receipt has been received. Download information is processed in a single transaction. The execution of this script is the last non statistical action in this transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_download connection event" on page 267
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

The following example shows one possible use of an end_download connection script. The ULEmpCust table has an action column. The following script uses the value in this column to delete records from the remote database.


```
CALL ml_add_connection_script(
 'ver1',
 'end_download',
 'DELETE FROM ULEmpCust ec
   ec.emp_id = {ml s.username} AND action = ''D''')
```

Java example

The following call to a MobiLink system procedure registers a Java method called endDownloadConnection as the script for the end_download connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'end_download',
  'ExamplePackage.ExampleClass.endDownloadConnection' )
```

Following is the sample Java method endDownloadConnection. The ULEmpCust table has an action column. The following script uses the value in this column to delete records from the remote database. It also uses the current MobiLink connection (saved earlier) to perform an update before the download ends. The SQL syntax is for SQL Anywhere consolidated databases.

```
public String endDownloadConnection(
  Timestamp ts,
  String user )
  throws java.sql.SQLException {
  String del_sql =    "DELETE FROM ULEmpCust ec " +
   "WHERE ec.emp_id = '" + user + "' " +
   "AND action = 'D' ";
   execUpdate( _syncConn, del_sql );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndDownload as the script for the end_download connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
 'ver1',
 'end_download',
 'TestScripts.Test.EndDownload' )
```

Following is the sample .NET method EndDownload. The ULEmpCust table has an action column. The following script uses the value in this column to delete records from the remote database. It also uses the current MobiLink connection (saved earlier) to perform an update before the download ends. The SQL syntax is for SQL Anywhere consolidated databases.

```
public string EndDownload(
  DateTime timestamp,
  string user ) {
  string del_sql = "DELETE FROM ULEmpCust ec " +
   "WHERE ec.emp_id = '" + user + "' " +
   "AND action = 'D' ";
   execUpdate( _syncConn, del_sql );
  return ( null );
}
```


end_download table event

Processes statements related to a specific table just after the MobiLink server concludes preparing the stream of downloaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.table	VARCHAR(128). The table name.	3

Default action

None.

Remarks

The MobiLink server executes this script after all rows have been downloaded and confirmation of receipt has been received. The download information is prepared in a separate transaction. The execution of this script is the last table-specific, non-statistical action in this transaction.

You can have one end_download script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_download table event" on page 269
- "end_download connection event" on page 308
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

The end_download table event is used to perform whatever steps you need performed after downloading a particular table. The following SQL Anywhere SQL script drops a temporary table created by a prepare_for_download script to hold download rows for the sales_summary table.

```
CALL ml_add_table_script(
 'MyCorp 1.0',
 'sales_summary',
 'end_download',
 'DROP TABLE #sales_summary_download' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called endDownloadTable as the script for the end_download table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script (
  'ver1',
  'table1',
  'end_download',
  'ExamplePackage.ExampleClass.endDownloadTable' )
```

Following is the sample Java method endDownloadTable. It resets the current table member variable.

```
public String endDownloadTable(
  Timestamp ts,
  String user,
  String table ) {
  _curTable = null;
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndTableDownload as the script for the end_download table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'end_download',
  'TestScripts.Test.EndTableDownload'
)
```

Following is the sample .NET method EndTableDownload. It resets the current table member variable.

```
public string EndTableDownload
  DateTime timestamp,
  string user,
  string table ) {
  _curTable = null;
  return ( null );
}
```


end_download_deletes table event

Processes statements related to a specific table just after preparing a list of rows to be deleted from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
table	VARCHAR(128). The table name.	3

Default action

None.

Remarks

This script is executed immediately after preparing a list of rows to be deleted from the named table in the remote database.

You can have one end_download_deletes script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_download_deletes table event" on page 272
- "end_download connection event" on page 308
- "begin_download_rows table event" on page 275
- "end_download_rows table event" on page 315
- "download_delete_cursor table event" on page 297
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

You may want to mark a row as deleted on the remote database. The following script updates a column in the consolidated database called OnRemote. *Note:* The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
CALL ml_add_table_script(
 'version1',
 'Leads',
 'end_download_deletes',
 'UPDATE Leads SET OnRemote = 0
  WHERE LastModified >= {ml s.last_table_download}
   AND Owner = {ml s.username} AND DeleteFlag=1');
```

Java example

The following call to a MobiLink system procedure registers a Java method called endDownloadDeletes as the script for the end_download_deletes table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'end_download_deletes',
  'ExamplePackage.ExampleClass.endDownloadDeletes' )
```

You may want to mark a row as deleted on the remote database. Following is the sample Java method endDownloadDeletes. It updates a column in the consolidated database called OnRemote to indicate the record no longer resides on the remote database.

Note: The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
public String endDownloadDeletes(
  Timestamp ts,
  String user,
  String table ) {
  return( "UPDATE Leads SET OnRemote = 0
     WHERE LastModified >= {ml s.last_table_download}
     AND Owner = {ml s.username} AND DeleteFlag=1" );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndDownloadDeletes as the script for the end_download_deletes table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'end_download_deletes',
  'TestScripts.Test.EndDownloadDeletes'
)
```

You may want to mark a row as deleted on the remote database. Following is the sample .NET method EndDownloadDeletes. It updates a consolidated database column called OnRemote to indicate that the record no longer resides on the remote database. The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.


```
public string EndDownloadDeletes(
  DateTime timestamp,
  string user,
  string table) {
  return( "UPDATE Leads SET OnRemote = 0
     WHERE LastModified >= {ml s.last_table_download}
     AND Owner = {ml s.username} AND DeleteFlag=1" );
}
```

end_download_rows table event

Processes statements related to a specific table just after preparing a list of rows to be inserted or updated in the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_table_download	TIMESTAMP. The last download time for the table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.table	VARCHAR(128). The table name.	3

Default action

None.

Remarks

This script is executed immediately after preparing the stream of rows to be inserted or updated in the named table in the remote database.

You can have one end_download_rows script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_download_rows table event" on page 275
- "end_download connection event" on page 308
- "end_download_deletes table event" on page 312
- "begin_download_deletes table event" on page 272
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

You may want to mark a row as successfully downloaded to the remote database. The following script updates a column in the consolidated database called OnRemote. *Note:* The WHERE clause on the UPDATE matches the WHERE clause used for your download_delete_cursor event script.

```
CALL ml_add_table_script(
   'version1',
   'Leads',
   'end_download_rows',
   'UPDATE Leads SET OnRemote = 1
    WHERE LastModified >= {ml s.last_table_download}
     AND Owner = {ml s.username}
     AND DownloadFlag=1' );
```

Java example

The following call to a MobiLink system procedure registers a Java method called endDownloadRows as the script for the end_download_rows table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'end_download_rows',
   'ExamplePackage.ExampleClass.endDownloadRows' )
```

Following is the sample Java method endDownloadRows. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String endDownloadRows(
  Timestamp ts,
  String user,
  String table ) {
  java.lang.System.out.println(
   "Done downloading inserts and updates for table "
   + table );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndDownloadRows as the script for the end_download_rows table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
 'ver1',
 'table1',
 'end_download_rows',
 'TestScripts.Test.EndDownloadRows'
)
```

Following is the sample .NET method EndDownloadRows. It prints a message to the MobiLink output log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string EndDownloadRows(
  DateTime timestamp,
  string user,
```



```
  string table ) {
  System.Console.WriteLine(
   "Done downloading inserts and updates for table "
   + table );
  return ( null );
}
```

end_publication connection event

Provides useful information about the publication(s) being synchronized.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.generation_number	INTEGER. If your deployment does not use file-based downloads, this parameter can be ignored. The default value is 1.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.publication_name	VARCHAR(128)	3
s.last_publication_upload	TIMESTAMP. Last successful upload time of this publication.	4
s.last_publication_download	TIMESTAMP. The last download time of this publication.	5
s.subscription_id	VARCHAR(128)	6

Default action

None.

Remarks

This event lets you design synchronization logic based on the publications currently being synchronized. This event is invoked in the same transaction as the end_synchronization event, and is invoked before the end_synchronization event. It is invoked once per publication being synchronized.

If the current synchronization successfully applied an upload, the last_upload parameter will contain the time this latest upload was applied. If the current synchronization has a successful download acknowledgement, the last_download time will contain the time this latest download was generated. This is the same value that was passed to the download scripts as the last download time.

If an UltraLite remote is synchronizing with UL_SYNC_ALL, this event is invoked once with the name 'unknown'.

Generation number

The generation_number parameter is specifically for file-based downloads.

The output value of the generation number is passed from the begin_publication script to the end_publication script. The meaning of the generation_number depends on whether the current synchronization is being used to create a download file, or whether the current synchronization has an upload.

In file-based downloads, generation numbers are used to force an upload before the download. The number is stored in the download file.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_publication connection event" on page 278
- "MobiLink File-Based Download" on page 193
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

SQL example

You may want to record the information for each publication being synchronized. The following example calls ml_add_connection_script to assign the event to a stored procedure called RecordPubEndSync.

```
CALL ml_add_connection_script(
   'version1',
   'end_publication',
   'CALL RecordPubEndSync(
    {ml s.generation_number},
    {ml s.username},
    {ml s.publication_name},
    {ml s.last_publication_upload},
    {ml s.last_publication_download} )' );
```

Java example

The following call to a MobiLink system procedure registers a Java method called endPublication as the script for the end_publication connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'end_publication',
   'ExamplePackage.ExampleClass.endPublication' )
```

Following is the sample Java method endPublication. It outputs a message to the MobiLink log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String endPublication(
  ianywhere.ml.script.InOutInteger generation_number,
  String user,
  String pub_name,
  Timestamp last_publication_upload,
  Timestamp last_publication_download ) {
```



```
    java.lang.System.out.println(
     "Finished synchronizing publication " + pub_name );
    return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndPub as the script for the end_publication connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'end_publication',
  'TestScripts.Test.EndPub'
)
```

Following is the sample .NET method endPub. It outputs a message to the MobiLink log. (Note that printing a message to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string EndPub(
  ref int generation_number,
  string user,
  string pub_name,
  DateTime last_publication_upload,
  DateTime last_publication_download ) {
  System.Console.Write(
   "Finished synchronizing publication " + pub_name );
  return ( null );
}
```


end_synchronization connection event

Processes any statements at the time an application disconnects from the MobiLink server upon completion of the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.sync_ok	INTEGER. This value is 1 for a successful synchronization and 0 for an unsuccessful synchronization.	2

Default action

None.

Remarks

The MobiLink server executes this script after synchronization is complete and, if expecting a download acknowledgement, the MobiLink client has returned confirmation of receipt of the download. It is executed within a separate transaction after the download transaction.

The end_synchronization script is useful for maintaining statistics. This is because if the begin_synchronization script is called, the end_synchronization script is invoked even if there is an error or conflict, so while the upload transaction is rolled back, statistics are maintained.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_synchronization connection event" on page 281
- "begin_synchronization table event" on page 283
- "end_synchronization table event" on page 323
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following SQL script calls a system procedure that records the end time of the synchronization attempt along with its success or failure status. This syntax is for SQL Anywhere consolidated databases.


```
CALL ml_add_connection_script(
 'ver1',
 'end_synchronization',
 'CALL RecordEndOfSyncAttempt(
   {ml s.username},
   {ml s.sync_ok} )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called endSynchronizationConnection as the script for the end_synchronization event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'end_synchronization',
  'ExamplePackage.ExampleClass.endSynchronizationConnection'
)
```

Following is the sample Java method endSynchronizationConnection. It uses a JDBC connection to execute an update. This syntax is for SQL Anywhere consolidated databases.

```
public String endSynchronizationConnection(
  String user )
  throws java.sql.SQLException {
  execUpdate( _syncConn,
   "UPDATE sync_count set count = count + 1 where user_id = '"
   + user + "' " );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndSync as the script for the end_synchronization connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'end_synchronization',
  'TestScripts.Test.EndSync'
)
```

Following is the sample .NET method EndSync. It updates the table sync_count. This syntax is for SQL Anywhere consolidated databases.

```
public string EndSync(
  string user ) {
  return(
   "UPDATE sync_count set count = count + 1 where user_id = '"
   + user + "' ");
  return ( null );
}
```


end_synchronization table event

Processes statements related to a specific table at the time an application disconnects from the MobiLink server upon completion of the synchronization process.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2
s.sync_ok	INTEGER. This value is 1 for a successful synchronization and 0 for an unsuccessful synchronization.	3

Default action

None.

Remarks

The MobiLink server executes this script after an application has synchronized and is about to disconnect from the MobiLink server, and before the connection level script of the same name.

You can have one end_synchronization script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_synchronization table event" on page 283
- "end_synchronization connection event" on page 321
- "end_synchronization table event" on page 323
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following SQL Anywhere SQL script drops a temporary table created by the begin_synchronization script.

```
CALL ml_add_table_script(
 'ver1',
 'sales_order',
 'end_synchronization',
 'DROP TABLE #sales_order' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called endSynchronizationTable as the script for the end_synchronization table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'end_synchronization',
   'ExamplePackage.ExampleClass.endSynchronizationTable' )
```

Following is the sample Java method endSynchronizationTable. It returns a SQL statement that drops a temporary table created by the begin_synchronization script.

```
public String endSynchronizationTable(
  String user,
  String table ) {
  return( "DROP TABLE #sales_order" );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndTableSync as the script for the end_synchronization table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
 'ver1',
 'table1',
 'end_synchronization',
 'TestScripts.Test.EndTableSync'
)
```

Following is the sample .NET method EndTableSync. It returns a SQL to statement that drops a temporary table created by the begin_synchronization script.

```
public string EndTableSync(
  string user,
  string table ) {
  return( "DROP TABLE #sales_order" );
}
```


end_upload connection event

Processes any statements just after the MobiLink server concludes processing uploaded inserts, updates, and deletes.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1

Default action

None.

Remarks

The MobiLink server executes this script as the last step in the processing of uploaded information. Upload information is processed in a single transaction. The execution of this script is the last action in this transaction before statistical scripts.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_upload connection event" on page 285
- "end_upload table event" on page 327
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following SQL Anywhere SQL script calls the EndUpload stored procedure.

```
CALL ml_add_connection_script(
 'ver1',
 'sales_order',
 'end_upload',
 'CALL EndUpload({ml username});' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called endUploadConnection as the script for the end_upload connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
```



```
    'end_upload',
    'ExamplePackage.ExampleClass.endUploadConnection' )
```

Following is the sample Java method endUploadConnection. It calls a method to perform operations on the database.

```
public String endUploadConnection( String user ) {
  // Clean up new and old tables.
  Iterator    two_iter = _tables_with_ops.iterator();
  while( two_iter.hasNext() ) {
    TableInfo cur_table = (TableInfo)two_iter.next();
    dumpTableOps( _sync_conn, cur_table );
  }
  _tables_with_ops.clear();
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndUpload as the script for the end_upload connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'end_upload',
  'TestScripts.Test.EndUpload'
)
```

Following is the sample .NET method EndUpload. It returns a SQL statement that calls the EndUpload stored procedure.

```
public string EndUpload( string user ) {
  return ( "CALL EndUpload({ml username});" );
}
```


end_upload table event

Processes statements related to a specific table just after the MobiLink server concludes processing the stream of uploaded inserts, updates, and deletions.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

The MobiLink server executes this script as the last step in the processing of uploaded information. Upload information is processed in a separate transaction. The execution of this script is the last table-specific action in this transaction.

You can have one end_upload script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_upload table event" on page 287
- "end_upload connection event" on page 325
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following call to a MobiLink system procedure assigns the end_upload event to a stored procedure called ULCustomerIDPool_maintain.


```
CALL ml_add_table_script(
   'custdb',
   'ULCustomerIDPool',
   'end_upload',
   'CALL ULCustomerIDPool_maintain( username );' )
```

The following SQL statements create the ULCustomerIDPool_maintain stored procedure.

```
CREATE OR REPLACE PROCEDURE ULCustomerIDPool_maintain(
  SyncUserID IN integer )
AS
  pool_count INTEGER;
  pool_max   INTEGER;
BEGIN
  -- Determine how many ids to add to the pool


SELECT COUNT(*)
    INTO pool_count
    FROM ULCustomerIDPool
    WHERE pool_emp_id = SyncUserID;
  -- Determine the current Customer id max


SELECT MAX(pool_cust_id)
    INTO pool_max
    FROM ULCustomerIDPool;
  -- Top up the pool with new ids


 WHILE pool_count < 20 LOOP
    pool_max := pool_max + 1;
      INSERT INTO ULCustomerIDPool(
          pool_cust_id, pool_emp_id )
            VALUES ( pool_max, SyncUserID );
      pool_count := pool_count + 1;
  END LOOP;
END;
```

Java example

The following call to a MobiLink system procedure registers a Java method called endUploadTable as the script for the end_upload table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1'
   'end_upload',
   'ExamplePackage.ExampleClass.endUploadTable' )
```

Following is the sample Java method endUploadTable. It generates a delete for a table with a name related to the passing-in table name. This syntax is for SQL Anywhere consolidated databases.

```
public String endUploadTable(
  String user,
  String table ) {
  return( "DELETE from '" + table + "_temp'" );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndTableUpload as the script for the end_upload table event when synchronizing the script version ver1 and the table table1.


```
CALL ml_add_dnet_table_script(
   'ver1',
   'table1',
   'end_upload',
   'TestScripts.Test.EndUpload'
)
```

The following .NET example moves rows inserted into a temporary table into the table passed into the script.

```
public string EndUpload( string user, string table ) {
  DBCommand stmt = curConn.CreateCommand();
  // Move the uploaded rows to the destination table.
  stmt.CommandText = "INSERT INTO "
    + table
    + " SELECT * FROM dnet_ul_temp";
  stmt.ExecuteNonQuery();
  stmt.Close();
  return ( null );
}
```


end_upload_deletes table event

Processes statements related to a specific table just after applying deletes uploaded from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

This script is run immediately after applying the changes that result from rows deleted in the remote table named in the second parameter.

You can have one end_upload_deletes script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "begin_upload_deletes table event" on page 289
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

You can use this event to process rows deleted during the upload on an intermediate table. You can compare the rows in the base table with rows in the intermediate table and decide what to do with the deleted row.

The following call to a MobiLink system procedure assigns the EndUploadDeletesLeads stored procedure to the end_upload_deletes event.


```
CALL ml_add_table_script(
  'version1',
  'Leads',
  'end_upload_deletes',
  'call EndUploadDeletesLeads()');
```

The following SQL statement creates the EndUploadDeletes stored procedure.

```
CREATE PROCEDURE EndUploadDeletesLeads ( )
Begin
  FOR names AS curs CURSOR FOR
  SELECT LeadID
    FROM Leads
    WHERE LeadID NOT IN (SELECT LeadID FROM T_Leads);
  DO
   CALL decide_what_to_do( LeadID );
  END FOR;
end
```

Java example

The following call to a MobiLink system procedure registers a Java method called endUploadDeletes as the script for the end_upload_deletes table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'end_upload_deletes',
   'ExamplePackage.ExampleClass.endUploadDeletes' )
```

Following is the sample Java method a endUploadDeletes. It calls a Java method that manipulates the database.

```
public String endUploadDeletes(
  String user,
  String table )
  throws java.sql.SQLException {
  processUploadedDeletes( _syncConn, table );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndUploadDeletes as the script for the end_upload_deletes table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
   'ver1',
   'table1',
   'end_upload_deletes',
   'TestScripts.Test.EndUploadDeletes'
)
```

Following is the sample .NET method a EndUploadDeletes. It calls a .NET method that manipulates the database.

```
public string EndUploadDeletes(
  string user,
  string table ) {
  processUploadedDeletes( _syncConn, table );
```



```
    return ( null );
}
```


end_upload_rows table event

Processes statements related to a specific table just after applying uploaded inserts and updates from the specified table in the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

Uploaded information can require inserting or updating rows in the consolidated database. This script is run immediately after applying the changes that result from modifications to the remote table named in the second parameter.

You can have one end_upload_rows script for each table in the remote database.

See also

- "Adding and deleting scripts" on page 228
- "begin_upload_rows table event" on page 292
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

Java example

The following call to a MobiLink system procedure registers a Java method called endUploadRows as the script for the end_upload_rows table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
```



```
    'end_upload_rows',
    'ExamplePackage.ExampleClass.endUploadRows' )
```

Following is the sample Java method endUploadRows. It calls a Java method that manipulates the database.

```
public String endUploadRows(
  String user,
  String table )
  throws java.sql.SQLException {
  processUploadedRows( _syncConn, table );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called EndUploadRows as the script for the end_upload_rows table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'end_upload_rows',
  'TestScripts.Test.EndUploadRows'
)
```

Following is the sample .NET method endUploadRows. It calls a .NET method that manipulates the database.

```
public string EndUploadRows(
  string user,
  string table ) {
  processUploadedRows( _syncConn, table );
  return ( null );
}
```


handle_DownloadData connection event

Used by direct row handling to create a set of rows to download.

Parameters

None.

Default action

None.

Remarks

The handle_DownloadData event allows you to determine what operations to download to MobiLink clients using direct row handling.

Direct row handling is used to synchronize to data sources other than MobiLink supported consolidated databases. See "Direct Row Handling" on page 517.

To create the direct download, you can use the DownloadData and DownloadTableData classes in the MobiLink server API for Java or .NET.

For Java, the DBConnectionContext getDownloadData method returns a DownloadData instance for the current synchronization. DownloadData encapsulates all download operations to send to a remote client. You can use the DownloadData getDownloadTables and getDownloadTableByName methods to obtain a DownloadTableData instance. DownloadTableData encapsulates download operations for a particular table. You can use the getUpsertPreparedStatement method to obtain prepared statements for insert and update operations. You can use the DownloadTableData getDeletePreparedStatement method to obtain prepared statements for delete operations.

For .NET, the DBConnectionContext GetDownloadData method returns a DownloadData instance for the current synchronization. DownloadData encapsulates all download operations to send to a remote client. You can use the DownloadData GetDownloadTables and GetDownloadTableByName methods to obtain a DownloadTableData instance. DownloadTableData encapsulates download operations for a particular table. You can use the GetUpsertCommand method to obtain commands for insert and update operations. You can use the DownloadTableData getDeleteCommand method to obtain commands for delete operations.

For Java, see "DBConnectionContext interface" on page 431. For .NET, see "DBConnectionContext interface" on page 481.

You can create the download in handle_DownloadData or another synchronization event. MobiLink provides this flexibility so that you can set the download when data is uploaded or when particular events occur. If you want to create the direct download in an event other then handle_DownloadData, you must create a handle_DownloadData script whose method does nothing. MobiLink requires this script to be defined to enable direct row handling. Except in upload-only synchronization, the MobiLink server requires that at a minimum, a handle_DownloadData script be defined.

If you create the direct download in an event other than handle_DownloadData, the event must not be before the begin_synchronization event and cannot be after the end_download event.

Note:
This event cannot be implemented as SQL.

See also

- "Direct Row Handling" on page 517
- "Setting direct downloads" on page 527
- Java: "DownloadData interface" on page 434
- Java: "DownloadTableData interface" on page 437
- .NET: "DownloadData interface" on page 492
- .NET: "DownloadTableData interface" on page 494
- "handle_UploadData connection event" on page 347
- "Required scripts" on page 227
- "Adding and deleting scripts" on page 228

Java example

The following call to a MobiLink system procedure registers a Java method called handleDownload for the handle_DownloadData connection event when synchronizing the script version ver1. You run this system procedure against your MobiLink consolidated database.

```
CALL ml_add_java_connection_script(
 'ver1',
 'handle_DownloadData',
 'MyPackage.MyClass.handleDownload' )
```

See "ml_add_java_connection_script" on page 536.

The following example shows you how to use the handleDownload method to create a download.

The following code sets up a class level DBConnectionContext instance in the constructor for a class called MobiLinkOrders.

```
import ianywhere.ml.script.*;
import java.io.*;
import java.sql.*;
import java.lang.System;

public class  MobiLinkOrders{

  DBConnectionContext _cc;

  public MobiLinkOrders( DBConnectionContext cc ) {
    _cc = cc;
  }
```

In your HandleDownload method, you use the DBConnectionContext getDownloadData method to return a DownloadData instance for the current synchronization. The DownloadData getDownloadTableByName method returns a DownloadTableData instance for the remoteOrders table. The DownloadTableData getUpsertPreparedStatement method returns a java.sql.PreparedStatement. To add an operation to the download, you set all column values and call the executeUpdate method.

Following is the handleDownload method of the MobiLinkOrders class. It adds two rows to the download for a table called remoteOrders.


```
// Method used for the handle_DownloadData event.
public void handleDownload() throws SQLException {
  // Get DownloadData instance for current synchronization.
  DownloadData my_dd = _cc.getDownloadData();

  // Get a DownloadTableData instance for the remoteOrders table.
  DownloadTableData td = my_dd.getDownloadTableByName("remoteOrders");

  // Get a java.sql.PreparedStatement for upsert (update/insert) operations.
  PreparedStatement upsert_ps = td.getUpsertPreparedStatement();

  // Set values for one row.
  upsert_ps.setInt( 1, 2300 );
  upsert_ps.setInt( 2, 100 );

  // Add the values to the download.
  int update_result = upsert_ps.executeUpdate();

  // Set values for another row.
  upsert_ps.setInt( 1, 2301 );
  upsert_ps.setInt( 2, 50 );
  update_result=upsert_ps.executeUpdate();

  upsert_ps.close();

  // ...
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called HandleDownload as the script for the authenticate_user connection event when synchronizing the script version ver1. This syntax is for SQL Anywhere consolidated databases.

```
CALL ml_add_dnet_connection_script(
   'ver1', 'handle_DownloadData',
   'TestScripts.Test.HandleDownload'
)
```

Following is the sample .NET method HandleDownload:

```
using System;
using System.Data;
using System.IO;
using iAnywhere.MobiLink.Script;
using iAnywhere.MobiLink;

namespace MyScripts
{
    /// <summary>
    /// Tests that scripts are called correctly for most sync events.
    /// </summary>
    public class MobiLinkOrders
    {
 private DBConnectionContext _cc;

 public MobiLinkOrders( DBConnectionContext cc )
 {
     _cc = cc;
 }

 ~MobiLinkOrders()
```



```
    {
    }

    public void handleDownload()
    {
        // Get DownloadData instance for current synchronization.
        DownloadData my_dd = _cc.GetDownloadData();

        // Get a DownloadTableData instance for the remoteOrders table.
        DownloadTableData td = my_dd.GetDownloadTableByName("remoteOrders");

        // Get an IdbCommand for upsert (update/insert) operations.
        IDbCommand upsert_stmt = td.GetUpsertCommand();

        IDataParameterCollection parameters = upsert_stmt.Parameters;

        // Set values for one row.
        parameters[ 0 ] = 2300;
        parameters[ 1 ] = 100;

        // Add the values to the download.
        int update_result = upsert_stmt.ExecuteNonQuery();

        // Set values for another row.
        parameters[ 0 ] = 2301;
        parameters[ 1 ] = 50;
        update_result = upsert_stmt.ExecuteNonQuery();

        // ...
    }

   }
}
```

handle_error connection event

Executed whenever the MobiLink server encounters a SQL error.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.action_code	INTEGER. This is an INOUT parameter.	1
s.error_code	INTEGER	2
s.error_message	TEXT	3
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	4
s.table	VARCHAR(128). If the script is not a table script, the table name is NULL.	5

Default action

When no handle_error script is defined or this script causes an error, the default action code is 3000: roll back the current transaction and cancel the current synchronization.

Remarks

The MobiLink server sends in the current action code. Initially, this is set to 3000 for each set of errors caused by a single SQL operation. Usually, there is only one error per SQL operation, but there may be more. This handle_error script is called once per error in the set. The action code passed into the first error is 3000. Subsequent calls are passed in the action code returned by the previous call. MobiLink will use the highest numerical value returned from multiple calls.

You can modify the action code in the script, and return a value instructing MobiLink how to proceed. The action code tells the MobiLink server what to do next. Before it calls this script, the MobiLink server sets the action code to a default value, which depends upon the severity of the error. Your script may modify this value. Your script must return or set an action code.

The action code parameter takes one of the following values:

- **1000** Skip the current row and continue processing.
- **3000** Roll back the current transaction and cancel the current synchronization. This is the default action code, and is used when no handle_error script is defined or this script causes an error.
- **4000** Roll back the current transaction, cancel the synchronization, and shut down the MobiLink server.

The error codes and message allow you to identify the nature of the error. If the error happened as part of synchronization, the user name is supplied. Otherwise, this value is NULL.

The MobiLink server executes this script if an ODBC error occurs while MobiLink is processing an insert, update, or delete script during the upload transaction or is fetching download rows. If an ODBC error occurs at another time, the MobiLink server calls the report_error or report_ODBC_error script and aborts the synchronization.

If the error happened while manipulating a particular table, the table name is supplied. Otherwise, this value is NULL. The table name is the name of a table in the client application. This name may or may not have a direct counterpart in the consolidated database, depending upon the design of the synchronization system.

SQL scripts for the handle_error event must be implemented as stored procedures.

You can return a value from the handle_error script one of the following ways:

- Pass the action parameter to an OUTPUT parameter of a procedure:

```
CALL my_handle_error( {ml s.action_code}, {ml s.error_code}, {ml
s.error_message}, {ml s.username}, {ml s.table} )
```

- Set the action code via a procedure or function return value:

```
{ml s.action_code} = CALL my_handle_error( {ml s.error_code}, {ml
s.error_message}, {ml s.username}, {ml s.table} )
```

  Most RDBMSs use the RETURN statement to set the return value from a procedure or function.

The CustDB sample application contains error handlers for various database-management systems.

See also

- “Script parameters” on page 221
- “Adding and deleting scripts” on page 228
- “report_error connection event” on page 367
- “report_odbc_error connection event” on page 370
- “handle_odbc_error connection event” on page 343
- “Using remote IDs and MobiLink user names in scripts” [*MobiLink - Client Administration*]

SQL example

The following example works with a SQL Anywhere consolidated database. It allows your application to ignore redundant inserts.

The following call to a MobiLink system procedure assigns the ULHandleError stored procedure to the handle_error event.


```
CALL ml_add_connection_script(
    'ver1',
    'handle_error',
    'CALL ULHandleError(
      {ml s.action_code},
      {ml s.error_code},
      {ml s.error_message},
      {ml s.username},
      {ml s.table} )' )
```

The following SQL statement creates the ULHandleError stored procedure.

```
CREATE PROCEDURE ULHandleError(
   INOUT action integer,
   IN error_code integer,
   IN error_message varchar(1000),
   IN user_name varchar(128),
   IN table_name varchar(128) )
BEGIN
  -- -196 is SQLE_INDEX_NOT_UNIQUE
  -- -194 is SQLE_INVALID_FOREIGN_KEY
  IF error_code = -196 or error_code = -194 then
      -- ignore the error and keep going
      SET action = 1000;
  ELSE
      -- abort the synchronization
      SET action = 3000;
  END IF;
END
```

Java example

The following call to a MobiLink system procedure registers a Java method called handleError as the script for the handle_error connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
    'ver1',
    'handle_error',
    'ExamplePackage.ExampleClass.handleError' )
```

Following is the sample Java method handleError. It processes an error based on the data that is passed in. It also determines the resulting error code.

```
public String handleError(
  ianywhere.ml.script.InOutInteger actionCode,
  int errorCode,
  String errorMessage,
  String user,
  String table ) {
  int new_ac;
  if( user == null ) {
    new_ac = handleNonSyncError( errorCode,
    errorMessage ); }
  else if( table == null ) {

    new_ac = handleConnectionError( errorCode,
    errorMessage, user ); }
  else {
    new_ac = handleTableError( errorCode,
    errorMessage, user, table );
  }
  // Keep the most serious action code.
  if( actionCode.getValue() < new_ac ) {
```



```
      actionCode.setValue( new_ac );
    }
  }
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called HandleError as the script for the handle_error connection event when synchronizing the script version ver1.

```
CALL mll_add_dnet_connection_script(
  'ver1',
  'handle_error',
  'TestScripts.Test.HandleError' )
```

Following is the sample .NET method HandleError.

```
public string HandleError() (
  ref int actionCode,
  int errorCode,
  string errorMessage,
  string user,
  string table ) {
  int new_ac;
  if( user == null ) {
    new_ac = HandleNonSyncError( errorCode,
    errorMessage ); }
  else if( table == null ) {
    new_ac = HandleConnectionError( errorCode,
    errorMessage, user ); }
  else {
    new_ac = HandleTableError( errorCode,
    errorMessage, user, table );
  }
  // Keep the most serious action code.
  if( actionCode < new_ac ) {
    actionCode = new_ac;
  }
}
```


handle_odbc_error connection event

Executed whenever the MobiLink server encounters an error triggered by the ODBC Driver Manager.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.action_code	INTEGER. This is an INOUT parameter.	1
s.ODBC_state	VARCHAR(5)	2
s.error_message	TEXT	3
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	4
s.table	VARCHAR(128)	5

Default action

The MobiLink server selects a default action code. You can modify the action code in the script, and return a value instructing MobiLink how to proceed. The action code parameter takes one of the following values:

- **1000** Skip the current row and continue processing.
- **3000** Rollback the current transaction and cancel the current synchronization. This is the default action code, and is used when no handle_error script is defined or this script causes an error.
- **4000** Rollback the current transaction, cancel the synchronization, and shut down the MobiLink server.

Remarks

The MobiLink server executes this script whenever it encounters an error flagged by the ODBC Driver Manager if the error occurs while MobiLink is processing an insert, update, or delete script during the upload

transaction or is fetching download rows. If an ODBC error occurs at another time, the MobiLink server calls the report_error or report_ODBC_error script and aborts the synchronization.

The error codes allow you to identify the nature of the error.

The action code tells the MobiLink server what to do next. Before it calls this script, the MobiLink server sets the action code to a default value, which depends upon the severity of the error. Your script may modify this value. Your script must return or set an action code.

The handle_odbc_error script is called after the handle_error and report_error scripts, and before the report_odbc_error script.

When only one, but not both, error-handling script is defined, the return value from that script decides error behavior. When both error-handling scripts are defined, the MobiLink server uses the numerically highest action code. If both handle_error and handle_ODBC_error are defined, MobiLink uses the action code with the highest numerical value returned from all calls.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "handle_error connection event" on page 339
- "report_error connection event" on page 367
- "report_odbc_error connection event" on page 370
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example works with a SQL Anywhere consolidated database. It allows your application to ignore ODBC integrity constraint violations.

The following call to a MobiLink system procedure assigns the HandleODBCError stored procedure to the handle_odbc_error event.

```
CALL ml_add_connection_script(
 'ver1',
 'handle_odbc_error',
 'CALL HandleODBCError(
   {ml s.action_code},
   {ml s.ODBC_state},
   {ml s.error_message},
   {ml s.username},
   {ml s.table} )' )
```

The following SQL statement creates the HandleODBCError stored procedure.

```
CREATE PROCEDURE HandleODBCError(
   INOUT action integer,
   IN odbc_state varchar(5),
   IN error_message varchar(1000),
   IN user_name varchar(128),
   IN table_name varchar(128) )
  BEGIN
   IF odbc_state = '23000' then
      -- Ignore the error and keep going.
      SET action = 1000;
   ELSE
      -- Abort the synchronization.
```



```
      SET action = 3000;
   END IF;
END
```

Java example

The following call to a MobiLink system procedure registers a Java method called handleODBCError as the script for the handle_odbc_error event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
 'ver1',
 'handle_odbc_error',
 'ExamplePackage.ExampleClass.handleODBCError'
)
```

Following is the sample Java method handleODBCError. It processes an error based on the data that is passed in. It also determines the resulting error code.

```
public String handleODBCError(
  ianywhere.ml.script.InOutInteger actionCode,
  String ODBCState,
  String errorMessage,
  String user,
  String table ) {
  int new_ac;
  if( user == null ) {
    new_ac = handleNonSyncError( ODBCState,
  errorMessage );
  }

  else if( table == null ) {
    new_ac = handleConnectionError( ODBCState,
     errorMessage, user );
  } else {
    new_ac = handleTableError( ODBCState,
     errorMessage, user, table );
  }
  // Keep the most serious action code.
  if( actionCode.getValue() < new_ac ) {
    actionCode.setValue( new_ac );
  }
  return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called HandleODBCError as the script for the handle_odbc_event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
 'ver1',
 'handle_odbc_error',
 'TestScripts.Test.HandleODBCError' )
```

Following is the sample .NET method HandleODBCError.

```
public string HandleODBCError (
  ref int actionCode,
  string ODBCState,
  string errorMessage,
  string user,
  string table ) {
```



```
    int new_ac;
    if( user == null ) {
      new_ac = HandleNonSyncError( ODBCState,
        errorMessage );
    }
    else if( table == null ) {
      new_ac = HandleConnectionError( ODBCState,
       errorMessage, user );
    } else {
      new_ac = HandleTableError( ODBCState,
       errorMessage, user, table );
    }
    // Keep the most serious action code.
    if( actionCode < new_ac ) {
      actionCode = new_ac;
    }
    return( null );
  }
```

handle_UploadData connection event

Used by direct row handling to process uploaded rows.

Parameters

Parameter name for SQL scripts	Description	Order
UploadData	A .NET or Java class encapsulating table operations uploaded by a MobiLink client. This class is defined in the MobiLink server API for Java and .NET.	1

Default action

None.

Remarks

The handle_UploadData event allows you to process the upload for MobiLink direct row handling. This event fires once for each upload transaction in a synchronization, unless you are using transaction-level uploads, in which case it fires for each transaction.

See "Direct Row Handling" on page 517.

This event takes a single UploadData parameter. Your Java or .NET method can use the UploadData getUploadedTables or getUploadedTableByName methods to obtain UploadedTableData instances. UploadedTableData allows you to access insert, update, and delete operations uploaded by a MobiLink client in the current synchronization.

For more information about the UploadData and UploadedTableData classes, see "Handling direct uploads" on page 521.

If you want to read column name metadata, you must specify the SendColumnNames MobiLink client extended option or property. Otherwise you can refer to columns by index, as defined at the remote database.

See "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*] and "Send Column Names synchronization parameter" [*MobiLink - Client Administration*].

Note:
This event cannot be implemented as SQL.

See also

- "Direct Row Handling" on page 517
- "Handling direct uploads" on page 521
- Java: "UploadData interface" on page 452
- Java: "UploadedTableData interface" on page 454
- .NET: "UploadData interface" on page 509
- .NET: "UploadedTableData interface" on page 511
- dbmlsync: "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*]

- UltraLite: "Send Column Names synchronization parameter" [*MobiLink - Client Administration*]
- "handle_DownloadData connection event" on page 335
- "Required scripts" on page 227
- "Adding and deleting scripts" on page 228

Java examples

The following call to a MobiLink system procedure registers a Java method called handleUpload for the handle_UploadData connection event when synchronizing the script version ver1. You run this system procedure against your MobiLink consolidated database.

```
CALL ml_add_java_connection_script(
  'ver1',
  'handle_UploadData',
  'MyPackage.MyClass.handleUpload' )
```

For more information about ml_add_java_connection_script, see "ml_add_java_connection_script" on page 536.

The following Java method processes the upload for the remoteOrders table. The UploadData.getUploadedTableByName method returns an UploadedTableData instance for the remoteOrders table. The UploadedTableData getInserts method returns a java.sql.ResultSet instance representing new rows.

```
import ianywhere.ml.script.*;
import java.sql.*;
import java.io.*;
// ...


public void handleUpload( UploadData ut )
  throws SQLException, IOException {
  // Get an UploadedTableData instance representing the
  // remoteOrders table.
  UploadedTableData remoteOrdersTable = ut.getUploadedTableByName
("remoteOrders");
  // Get inserts uploaded by the MobiLink client.
  java.sql.ResultSet rs = remoteOrdersTable.getInserts();

  while( rs.next() ) {
    // You can reference column names here because SendColumnNames is on
    // Get the primary key.
    int pk=rs.getInt("pk");

    // Get the uploaded num_ordered value.
    int num_ordered = rs.getInt("num_ordered");

    // The current insert row is now ready to be uploaded to wherever
    // you want it to go (a file, a web service, and so on).

  }

  rs.close();

}
```

The following example outputs insert, update and delete operations uploaded by a MobiLink remote database. The UploadData getUploadedTables method returns UploadedTableData instances representing all tables uploaded by a remote. The order of the tables in this array is the order in which they where uploaded

by the remote. The UploadedTableData getInserts, getUpdates, and getDeletes methods return standard JDBC result sets. You can use the println method or output data to a text file or another location.

```
import ianywhere.ml.script.*;
import java.sql.*;
import java.io.*;
// ...


public void handleUpload( UploadData ud )
  throws SQLException, IOException {
  int i;
  UploadedTableData tables[] = ud.getUploadedTables();
  for( i=0; i<tables.length; i+=1 ) {
    UploadedTableData cur_table = tables[i];
    println( "table " + java.lang.Integer.toString( i ) +
     " name: " + cur_table.getName() );

    // Print out delete result set.
    println( "Deletes" );
    printRSInfo( cur_table.getDeletes() );

    // Print out insert result set.
    println( "Inserts" );
    printRSInfo( cur_table.getInserts() );

    // print out update result set
    println( "Updates" );
    printUpdateRSInfo( cur_table.getUpdates() );
  }
}
```

The printRSInfo method prints out an insert, update, or delete result set and accepts a single java.sql.ResultSet object. Detailed column information, including column labels, is provided by the ResultSetMetaData object returned by the ResultSet getMetaData method. Column labels are available only if the client has the SendColumnNames option turned on. The printRow method prints out each row in a result set.

```
public void printRSInfo( ResultSet rs )
  throws SQLException, IOException {

  // Obtain the result set metadata.
  ResultSetMetaData md = rs.getMetaData();
  String columnHeading = "";


  // Print out column headings.
  for( int c=1; c <= md.getColumnCount(); c = c + 1 ) {
    columnHeading += md.getColumnLabel(c);
    if( c < md.getColumnCount() ) {
      columnHeading += ", ";
    }
  }


  println( columnHeading );
  while( rs.next() ) {

    // Print out each row.
    printRow( rs, md.getColumnCount() );
  }

  // Close the java.sql.ResultSet.
```



```
  rs.close();
}
```

The printRow method, shown below, uses the ResultSet getString method to obtain each column value.

```
public void printRow( ResultSet rs, int col_count )
  throws SQLException, IOException {
  String row = new String( "( " );
  int c;


  for( c=1; c<=col_count; c+=1 ) {
    // Get a column value.
    String cur_col = rs.getString( c );


    // Check for null values.
    if( cur_col == null ) {
      cur_col = "<NULL>";
    }


    // Add the column value to the row string.
    row += cur_col;
    if( c < col_count ) {
      row += ", ";
    }
  }


  row += " )";

  // Print out the row.
  println( row );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called HandleUpload for the handle_UploadData connection event when synchronizing the script version ver1. You run this system procedure against your MobiLink consolidated database.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'handle_UploadData',
  'TestScripts.Test.HandleUpload' )
```

The following .NET method processes the upload for the remoteOrders table.

```
using System;
using System.Data;
using System.IO;
using iAnywhere.MobiLink.Script;
using iAnywhere.MobiLink;

namespace MyScripts
{
    /// <summary>
    /// Tests that scripts are called correctly for most sync events.
    /// </summary>
    public class MyClass
    {
 public MyClass( DBConnectionContext cc )
```



```
 {
 }

 ~MyClass()
 {
 }

 public void handleUpload( UploadData ut )
 {
     // Get an UploadedTableData instance representing the
     // remoteOrders table.
     UploadedTableData remoteOrdersTable = ut.GetUploadedTableByName
("remoteOrders");
     // Get inserts uploaded by the MobiLink client.
     IDataReader dr = remoteOrdersTable.GetInserts();

     while( dr.Read() ) {

  // Get the primary key.
  int pk = dr.GetInt32( 0 );

  // Get the uploaded num_ordered value.
  int num_ordered = dr.GetInt32( 1 );

  // The current insert row is now ready to be uploaded to
  // wherever you want it to go (a file, a Web Service, -- whatever )
  //...

     }

     dr.Close();
 }
    }
}


using System;
using System.Data;
using System.IO;
using iAnywhere.MobiLink.Script;
using iAnywhere.MobiLink;

namespace MyScripts
{
    /// <summary>
    /// Tests that scripts are called correctly for most sync events.
    /// </summary>
    public class MyUpload
    {
 public MyUpload( DBConnectionContext cc )
 {
 }

 ~MyUpload()
 {
 }

 public void handleUpload( UploadData ut )
 {
     int i;
     UploadedTableData[] tables = ut.GetUploadedTables();

     for( i=0; i<tables.Length; i+=1 ) {
  UploadedTableData cur_table = tables[i];
```

```
 Console.Write( "table " + i + " name: " + cur_table.GetName() );
 // Print out delete result set.
 Console.Write( "Deletes" );
 printRSInfo( cur_table.GetDeletes() );

 // Print out insert result set.
 Console.Write( "Inserts" );
 printRSInfo( cur_table.GetInserts() );

 // print out update result set
 Console.Write( "Updates" );
 printUpdateRSInfo( cur_table.GetUpdates() );
    }
}

public void printRSInfo( IDataReader dr )
{
    // Obtain the result set metadata.
    DataTable dt = dr.GetSchemaTable();
    DataColumnCollection cc = dt.Columns;
    DataColumn dc;
    String columnHeading = "";

    // Print out column headings.
    for( int c=0; c < cc.Count; c = c + 1 ) {
 dc = cc[ c ];
 columnHeading += dc.ColumnName;
 if( c < cc.Count - 1 ) {
     columnHeading += ", ";
 }
    }
    Console.Write( columnHeading );

    while( dr.Read() ) {
 // Print out each row.
 printRow( dr, cc.Count );
    }

    // Close the java.sql.ResultSet.
    dr.Close();
}

public void printUpdateRSInfo( UpdateDataReader utr )
{
    // Obtain the result set metadata.
    DataTable dt = utr.GetSchemaTable();
    DataColumnCollection cc = dt.Columns;
    DataColumn dc;
    String columnHeading = "TYPE, ";

    // Print out column headings.
    for( int c = 0; c < cc.Count; c = c + 1 ) {
 dc = cc[ c ];
 columnHeading += dc.ColumnName;
 if( c < cc.Count - 1 ) {
     columnHeading += ", ";
 }
    }
    Console.Write( columnHeading );

    while( utr.Read() ) {
 // Print out the new values for the row.
 utr.SetNewRowValues();
 Console.Write( "NEW:" );
```



```
    printRow( utr, cc.Count );

    // Print out the old values for the row.
    utr.SetOldRowValues();
    Console.Write( "OLD:" );
    printRow( utr, cc.Count );
       }

       // Close the java.sql.ResultSet.
       utr.Close();
   }

   public void printRow( IDataReader dr, int col_count )
   {
       String row = "( ";
       int c;

       for( c = 0; c < col_count; c = c + 1 ) {
    // Get a column value.
    String cur_col = dr.GetString( c );

    // Check for null values.
    if( cur_col == null ) {
        cur_col = "<NULL>";
    }

    // Add the column value to the row string.
    row += cur_col;
    if( c < col_count ) {
        row += ", ";
    }
       }

       row += " )";

       // Print out the row.
       Console.Write( row );
   }
      }
  }
```

modify_error_message connection event

The script can be used to customize the message text (error, warning, and information) that is sent to remote databases.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.error_message	VARBINARY(1024). This is an IN-OUT parameter.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2
s.error_code	INT.	3

Default action

None.

Remarks

SQL scripts for the modify_error_message event must be implemented as stored procedures.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example downloads everything from one day ago, regardless of whether the databases were synchronized since then.

The following SQL statement creates the ModifyLastErrorMessage stored procedure:


```
CREATE PROCEDURE ModifyLastErrorMessage(
  inout error_message VARBINARY(1024),
  in username VARCHAR(128),
  in error_code INT )
BEGIN
  SELECT dateadd(day, -1, last_download_time )
  INTO last_download_time
END
```

The following call to a MobiLink system procedure assigns ModifyLastErrorMessage to the modify_error_message connection event for the script version modify_ts_test:

```
CALL ml_add_connection_script(
  'modify_ts_test',
  'modify_error_message',
  'CALL ModifyErrorMessage (
    {ml s.error_message},
    {ml s.username},
    {ml s.error_code} )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called modifyLastErrorMessage as the script for the modify_error_message connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'modify_error_message',
  'ExamplePackage.ExampleClass.modifyLastErrorMessage' )
```

Following is the sample Java method modifyLastErrorMessage. It prints the current error message and error code.

```
public String modifyLastErrorMessage(
  String last_error_message,
  String user_name,
  int error_code ) {
  java.lang.System.out.println( "error message: " +
   last_error_message );
  java.lang.System.out.println( "error code: " +
   String.valueOf(error_code) );
  return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ModifyLastErrorMessage as the script for the modify_error_message connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'modify_error_message,
  'TestScripts.Test.ModifyLastErrorMessage' )
```

Following is a sample .NET method ModifyLastErrorMessage. It prints the current error code and error message.

```
public string ModifyLastErrorMessage (
  string errorMessage,
  string userName,
```



```
        string errorCode  ) {
        System.Console.WriteLine( "error message: " + errorMessage );
        System.Console.WriteLine( "error code: " +  errorCode );
        return ( null );
    }
```


modify_last_download_timestamp connection event

The script can be used to modify the last download time for the current synchronization.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_download	TIMESTAMP. The last download time for any synchronized table. This is an INOUT parameter.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2

Default action

None.

Remarks

Use this script when you want to modify the last_download timestamp for the current synchronization. If this script is defined, the MobiLink server uses the modified last_download timestamp as the last_download timestamp passed to the download scripts. A typical use of this script is to recover from losing data on the remote; you can reset the last_download timestamp to an early time such as 1900-01-01 00:00 so that the next synchronization will download all the data.

SQL scripts for the modify_last_download_timestamp event must be implemented as stored procedures. The MobiLink server passes in the last_download timestamp as the first parameter to the stored procedure, and replaces the timestamp by the first value passed out by the stored procedure.

This script is executed just before the prepare_for_download script, in the same transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

- "Using last download times in scripts" on page 98
- "How download timestamps are generated and used" on page 99
- "modify_next_last_download_timestamp connection event" on page 360

SQL example

The following SQL statement creates a stored procedure. The following syntax is for Oracle consolidated databases:

```
CREATE PROCEDURE ModifyDownloadTimestamp(
        download_timestamp  OUT DATE,
        user_name   IN VARCHAR )
        AS
        BEGIN
         -- N is the maximum replication latency in the consolidated cluster
        download_timestamp := download_timestamp - N;
        END;
```

The following syntax is for SQL Anywhere, Adaptive Server Enterprise, and Microsoft SQL Server consolidated databases:

```
CREATE PROCEDURE ModifyDownloadTimestamp
        @download_timestamp DATETIME OUTPUT,
        @t_name    VARCHAR( 128 )
        AS
        BEGIN
         -- N is the maximum replication latency in consolidated cluster
        SELECT @download_timestamp = @download_timestamp - N
        END
```

The following call to a MobiLink system procedure assigns the ModifyDownloadTimestamp stored procedure to the modify_last_download_timestamp event. The following syntax is for a SQL Anywhere consolidated database:

```
CALL ml_add_connection_script(
  'my_version',
  'modify_last_download_timestamp',
  '{CALL ModifyDownloadTimestamp(
   {ml s.last_download},
   {ml s.username} ) }' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called modifyLastDownloadTimestamp as the script for the modify_last_download_timestamp connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'modify_last_download_timestamp',
  'ExamplePackage.ExampleClass.modifyLastDownloadTimestamp' )
```

Following is the sample Java method modifyLastDownloadTimestamp. It prints the current and new timestamp and modifies the timestamp that is passed in.

```
public String modifyLastDownloadTimestamp(
     Timestamp last_download_time,
     String user_name ) {
   java.lang.System.out.println( "old date: " +
```



```
     last_download_time.toString() );
    last_download_time.setDate(
     last_download_time.getDate() -1 );
    java.lang.System.out.println( "new date: " +
     last_download_time.toString() );
    return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ModifyLastDownloadTimestamp as the script for the modify_last_download_timestamp connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'modify_last_download_timestamp',
  'TestScripts.Test.ModifyLastDownloadTimestamp' )
```

Following is the sample .NET method ModifyLastDownloadTimestamp.

```
public string ModifyLastDownloadTimestamp(
  DateTime lastDownloadTime,
  string userName ) {
  System.Console.WriteLine( "old date: " +
   last_download_time.ToString() );
  last_download_time = DateTime::Now;
  System.Console.WriteLine( "new date: " +
   last_download_time.ToString() );
  return( null );
}
```


modify_next_last_download_timestamp connection event

The script can be used to modify the last download time for the next synchronization.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.next _last_download	TIMESTAMP. This is an INOUT parameter. The MobiLink server generates this value immediately after the upload is committed.	1
s.last_download	TIMESTAMP. This is an IN parameter. This is the last download time for the current synchronization. It can be useful in avoiding duplication, by making sure you don't modify the next_last_download timestamp to be earlier than the last_download timestamp.	2
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	3

Default action

None.

Remarks

This script allows you to change the next_last_download timestamp, which effectively changes the last_download timestamp for the next synchronization. This allows you to reset the next synchronization without affecting the current synchronization.

SQL scripts for the modify_next_last_download_timestamp event must be implemented as stored procedures. The MobiLink server passes in the next_last_download timestamp as the first parameter to the stored procedure, and replaces the timestamp by the first value passed out by the stored procedure.

This script is executed in the download transaction, after downloading user tables.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98
- "How download timestamps are generated and used" on page 99
- "modify_last_download_timestamp connection event" on page 357

SQL example

The following example shows one application of this script. Create a stored procedure. The following syntax is for a SQL Anywhere consolidated database:

```
CREATE PROCEDURE ModifyNextDownloadTimestamp(
  inout download_timestamp TIMESTAMP ,
  in last_download TIMESTAMP ,
  in user_name VARCHAR(128) )
  BEGIN
    SELECT dateadd(hour, -1, download_timestamp )
    INTO download_timestamp
END
```

Install the script into your SQL Anywhere consolidated database:

```
CALL ml_add_connection_script(
  'modify_ts_test',
  'modify_next_last_download_timestamp',
  'CALL  ModifyNextDownloadTimestamp (
   {ml s.next_last_download},
   {ml s.last_download},
   {ml s.username} )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called modifyNextDownloadTimestamp as the script for the modify_next_last_download_timestamp connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'modify_next_last_download_timestamp',
   'ExamplePackage.ExampleClass.modifyNextDownloadTimestamp' )
```

Following is the sample Java method modifyNextDownloadTimestamp. It sets the download timestamp back an hour.

```
public String modifyNextDownloadTimestamp(
  Timestamp download_timestamp,
  Timestamp last_download,
  String user_name ) {
  download_timestamp.setHours(
  download_timestamp.getHours() -1 );
```



```
    return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ModifyNextDownloadTimestamp as the script for the modify_next_last_download_timestamp connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
   'ver1',
   'modify_next_last_download_timestamp',
   'TestScripts.Test.ModifyNextDownloadTimestamp' )
```

Following is the sample .NET method ModifyNextDownloadTimestamp. It sets the download timestamp back an hour.

```
public string ModifyNextDownloadTimestamp (
  DateTime download_timestamp,
  DateTime last_download,
  string user_name ) {
  download_timestamp = download_timestamp.AddHours( -1 );
  return ( null );
}
```


modify_user connection event

Provide the MobiLink user name.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name. This is an INOUT parameter.	1

Default action

None.

Remarks

The MobiLink server provides the user name as a parameter when it calls scripts; the user name is sent by the MobiLink client. In some cases, you may want to have an alternate user name. This script allows you to modify the user name used in calling MobiLink scripts.

The username parameter must be long enough to hold the user name.

SQL scripts for the modify_user event must be implemented as stored procedures.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "authenticate_user connection event" on page 256
- "authenticate_user_hashed connection event" on page 261
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example maps a remote database user name to the ID of the user using the device, by using a mapping table called user_device. This technique can be used when the same person has multiple remotes (such as a PDA and a laptop) requiring the same synchronization logic (based on the user's name or id).

The following call to a MobiLink system procedure assigns the ModifyUser stored procedure to the modify_user event. This syntax is for a SQL Anywhere consolidated database.

```
CALL ml_add_connection_script(
 'ver1',
 'modify_user',
 'call ModifyUser( {ml s.username} )' )
```


The following SQL statement creates the ModifyUser stored procedure.

```
CREATE PROCEDURE ModifyUser( INOUT u_name varchar(128) )
BEGIN
 SELECT user_name
    INTO u_name
    FROM user_device
    WHERE device_name = u_name
END
```

Java example

The following call to a MobiLink system procedure registers a Java method called modifyUser as the script for the modify_user connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'modify_user',
   'ExamplePackage.ExampleClass.modifyUser' )
```

Following is the sample Java method modifyUser. It gets the user ID from the database and then uses it to set the user name.

```
public String ModifyUser(
  InOutString io_user_name )
  throws SQLException {
  Statement uid_select = curConn.createStatement();
  ResultSet uid_result = uid_select.executeQuery(
   "SELECT rep_id FROM SalesRep WHERE name = '" +

   io_user_name.getValue() + "' " );
  uid_result.next();
  io_user_name.setValue(
   java.lang.Integer.toString(uid_result.getInt( 1 ));
   uid_result.close();
   uid_select.close();
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ModUser as the script for the modify_user connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'modify_user',
  'TestScripts.Test.ModUser'
)
```

Following is the sample .NET method ModUser.

```
public string ModUser(
  string user ) {
  return ( "SELECT rep_id FROM SalesRep WHERE name = '" + user + "' " );
}
```


prepare_for_download connection event

Processes any required operations between the upload and download transactions.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.last_download	TIMESTAMP. The last download time of any synchronized table.	1
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	2

Default action

None.

Remarks

The MobiLink server executes this script as a separate transaction, between the upload transaction and the start of the download transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "end_upload connection event" on page 325
- "begin_download connection event" on page 267
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]
- "Using last download times in scripts" on page 98

Java example

The following call to a MobiLink system procedure registers a Java method called prepareForDownload as the script for the prepare_for_download event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
```



```
    'prepare_for_download',
    'ExamplePackage.ExampleClass.prepareForDownload' )
```

Following is the sample Java method prepareForDownload. It calls a Java method to modify some rows in the database.

```
public String prepareForDownload(
  Timestamp ts,
  String user ) {
  adjustUploadedRows( _syncConn, user );
  return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called PrepareForDownload as the script for the prepare_for_download connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
 'ver1',
 'prepare_for_download',
 'TestScripts.Test.PrepareForDownload'
)
```

Following is the sample .NET method PrepareForDownload. It calls a .NET method to modify some rows in the database.

```
public string PrepareForDownload(
  DateTime timestamp,
  string user ) {
  AdjustUploadedRows ( _syncConn, user );
  return ( null );
}
```


report_error connection event

Allows you to log errors and to record the actions selected by the handle_error script.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.action_code	INTEGER. This is an INOUT parameter. This parameter is mandatory.	1
s.error_code	INTEGER.	2
s.error_message	TEXT.	3
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	4
s.table	VARCHAR(128).	5

Default action

None.

Remarks

This script allows you to log errors and to record the actions selected by the handle_error script. This script is executed after the handle_error event, whether or not a handle_error script is defined. It is always executed in its own transaction, on a different database connection than the synchronization connection (the administrative/information connection).

The error code and error message allow you to identify the nature of the error. The action code value is returned by the last call to an error handling script for the SQL operation that caused the current error.

If the error happened as part of synchronization, the user name is supplied. Otherwise, this value is NULL.

If the error happened while manipulating a particular table, the table name is supplied. Otherwise, this value is NULL. The table name is the name of a table in the remote database. This name may or may not have a direct counterpart in the consolidated database, depending on the design of the synchronization system.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "handle_error connection event" on page 339
- "handle_odbc_error connection event" on page 343
- "report_odbc_error connection event" on page 370
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example works with a SQL Anywhere consolidated database. It inserts a row into a table used to record synchronization errors.

```
CALL ml_add_connection_script(
 'ver1',
 'report_error',
 'INSERT INTO sync_error(
   action_code,
   error_code,
   error_message,
   user_name,
   table_name )
  VALUES (
   {ml s.action_code},
   {ml s.error_code},
   {ml s.error_message},
   {ml s.username},
   {ml s.table} )'  )
```

Java example

The following call to a MobiLink system procedure registers a Java method called reportError as the script for the report_error connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'report_error,
   'ExamplePackage.ExampleClass.reportError' )
```

Following is the sample Java method reportError. It logs the error to a table using the JDBC connection provided by MobiLink. It also sets the action code.

```
public String reportError(
  ianywhere.ml.script.InOutInteger actionCode,
  int errorCode,
  String errorMessage,
  String user,
  String table )
  throws java.sql.SQLException {
   // Insert error information in a table,
   JDBCLogError( _syncConn, errorCode, errorMessage,
    user, table );
   actionCode.setValue( getActionCode( errorCode ) );
   return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ReportError as the script for the report_error connection event when synchronizing the script version ver1.


```
CALL ml_add_dnet_connection_script(
   'ver1',
   'report_error,
   'TestScripts.Test.ReportError' )
```

Following is the sample .NET method reportError. It logs the error to a table using a .NET method.

```
public string ReportError(
  ref int actionCode,
  int errorCode,
  string errorMessage,
  string user,
  string table ) {
  LogError(_syncConn, errorCode, errorMessage, user, table);
}
```


report_odbc_error connection event

Allows you to log errors and to record the actions selected by the handle_odbc_error script.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.action_code	INTEGER. This is an INOUT parameter. This parameter is mandatory.	1
s.ODBC_state	VARCHAR(5).	2
s.error_message	TEXT.	3
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128).	4
s.table	VARCHAR(128).	5

Default action

None.

Remarks

This script allows you to log errors and to record the actions selected by the handle_odbc_error script. This script is executed after the handle_odbc_error event, whether or not a handle_odbc_error script is defined. It is always executed in its own transaction, on a different database connection than the synchronization connection (the administrative/information connection).

The error code and error message allow you to identify the nature of the error. The action code value is returned by the last call to an error handling script for the SQL operation that caused the current error.

If the error happened as part of synchronization, the user name is supplied. Otherwise, this value is NULL.

If the error happened while manipulating a particular table, the table name is supplied. Otherwise, this value is NULL. The table name is the name of a table in the remote database. This name may or may not have a direct counterpart in the consolidated database, depending on the design of the synchronization system.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "handle_error connection event" on page 339
- "handle_odbc_error connection event" on page 343
- "report_error connection event" on page 367
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example works with a SQL Anywhere consolidated database. It inserts a row into a table used to record synchronization errors.

```
CALL ml_add_connection_script(
 'ver1',
 'report_odbc_error',
 'INSERT INTO sync_error(
    action_code,
    odbc_state,
    error_message,
    user_name,
    table_name )
  VALUES(
   {ml s.action_code},
   {ml s.ODBC_state},
   {ml s.error_message},
   {ml s.username},
   {ml s.table} )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called reportODBCError as the script for the report_odbc_error event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'report_odbc_error',
   'ExamplePackage.ExampleClass.reportODBCError' )
```

Following is the sample Java method reportODBCError. It logs the error to a table using the JDBC connection provided by MobiLink. It also sets the action code.

```
public String reportODBCError(
  ianywhere.ml.script.InOutInteger actionCode,
  String ODBCState,
  String errorMessage,
  String user,
  String table )
  throws java.sql.SQLException {
  JDBCLogError( _syncConn, ODBCState, errorMessage,
   user, table );
  actionCode.setValue( getActionCode( ODBCState ) );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ReportODBCError as the script for the report_odbc_error event when synchronizing the script version ver1.


```
CALL ml_add_dnet_connection_script(
   'ver1',
   'report_odbc_error',
   'TestScripts.Test.reportODBCError' )
```

Following is the sample .NET method ReportODBCError. It logs the error to a table using a .NET method.

```
public string ReportODBCError (
  ref int actionCode,
  string ODBCState,
  string errorMessage,
  string user,
  string table ) {
  LogError(_syncConn, ODBCState, errorMessage, user, table);
  return ( null );
}
```


resolve_conflict table event

Defines a process for resolving a conflict in a specific table.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2

Default action

None.

Remarks

When a row is updated on a remote database, the MobiLink client saves a copy of the original values. The client sends both old and new values to the MobiLink server.

When the MobiLink server receives an updated row, it compares the original values with the present values in the consolidated database. The comparison is carried out using the upload_fetch script.

If the old uploaded values do not match the current values in the consolidated database, the row conflicts. Instead of updating the row, the MobiLink server inserts both old and new values into the consolidated database. The old and new rows are handled using the upload_old_row_insert and upload_new_row_insert scripts, respectively. If you are using cursor-based uploads the rows are handled using old_row_cursor and new_row_cursor, respectively.

Once the values have been inserted, the MobiLink server executes the resolve_conflict script. It provides the opportunity to resolve the conflict. You can implement any scheme of your choosing.

This script is executed once per conflict.

Alternatively, instead of defining the resolve_conflict script, you can resolve conflicts in a set-oriented fashion by putting conflict-resolution logic either in your end_upload_rows script or in your end_upload table script.

You can have one resolve_conflict script for each table in the remote database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "upload_old_row_insert table event" on page 398
- "upload_new_row_insert table event" on page 395
- "upload_update table event" on page 410
- "end_upload_rows table event" on page 333
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following statement defines a resolve_conflict script suited to the CustDB sample application for an Oracle installation. It calls a stored procedure ULResolveOrderConflict.

```
exec ml_add_table_script(
   'custdb', 'ULOrder', 'resolve_conflict',
   'begin ULResolveOrderConflict();
end; ')

CREATE OR REPLACE PROCEDURE ULResolveOrderConflict()
AS
  new_order_id integer;
  new_status   varchar(20);
  new_notes   varchar(50);
BEGIN
  -- approval overrides denial
  SELECT order_id, status, notes
    INTO new_order_id, new_status, new_notes
    FROM ULNewOrder
   WHERE syncuser_id = SyncUserID;
  IF new_status = 'Approved' THEN
    UPDATE ULOrder o
       SET o.status = new_status, o.notes =

new_notes
     WHERE o.order_id = new_order_id;
  END IF;
  DELETE FROM ULOldOrder;
  DELETE FROM ULNewOrder;
END;
```

Java example

The following call to a MobiLink system procedure registers a Java method called resolveConflict as the script for the resolve_conflict table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
   'ver1',
   'table1',
   'resolve_conflict',
   'ExamplePackage.ExampleClass.resolveConflict' )
```


Following is the sample Java method resolveConflict. It calls a Java method that uses the JDBC connection provided by MobiLink to resolve the conflict.

```
public String resolveConflict(
  String user,
  String table) {
  resolveRows(_syncConn, user );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called ResolveConflict as the script for the resolve_conflict table event when synchronizing the script version ver1.

```
CALL ml_add_dnet_table_script(
   'ver1',
   'table1',
   'resolve_conflict',
   'TestScripts.Test.ResolveConflict' )
```

Following is the sample .NET method ResolveConflict. It calls a .NET method that resolves the conflict.

```
public string ResolveConflict(
  String user,
  String table) {
  ResolveRows(_syncConn, user );
}
```


synchronization_statistics connection event

Tracks synchronization statistics.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.warnings	INTEGER. The number of warnings issued during the synchronization.	2
s.errors	INTEGER. The number of errors that occurred during the synchronization.	3
s.deadlocks	INTEGER. The number of deadlocks in the consolidated database that were detected for the synchronization.	4
s.synchronized_tables	INTEGER. The number of client tables that were involved in the synchronization.	5
s.connection_retries	INTEGER. The number of times the MobiLink server retried the connection to the consolidated database.	6

Default action

None.

Remarks

The synchronization_statistics event allows you to gather, for any user and connection, various statistics about the current synchronization. The synchronization_statistics connection script is called just prior to the commit at the end of the end synchronization transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "download_statistics connection event" on page 300
- "download_statistics table event" on page 303
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405
- "synchronization_statistics table event" on page 379
- "time_statistics connection event" on page 382
- "time_statistics table event" on page 385
- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts synchronization statistics into the sync_con_audit table.

```
CALL ml_add_connection_script(
  'ver1',
  'synchronization_statistics',
 'INSERT INTO sync_con_audit(
   ml_user,
   warnings,
   errors,
   deadlocks,
   synchronized_tables,
   connection_retries)
  VALUES (
   {ml s.username},
   {ml s.warnings},
   {ml s.errors},
   {ml s.deadlocks},
   {ml s.synchronized_tables},
   {ml s.connection_retries})' )
```

Once statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following call to a MobiLink system procedure registers a Java method called synchronizationStatisticsConnection as the script for the synchronization_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'synchronization_statistics',
  'ExamplePackage.ExampleClass.synchronizationStatisticsConnection'
)
```


Following is the sample Java method synchronizationStatisticsConnection. It logs some of the statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String synchronizationStatisticsConnection(
  String user,
  int warnings,
  int errors,
  int deadlocks,
  int synchronizedTables,
  int connectionRetries ) {
  java.lang.System.out.println(
   "synch statistics number of deadlocks: "
   + deadlocks ;
  return( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called SyncStats as the script for the synchronization_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'synchronization_statistics',
  'TestScripts.Test.SyncStats'
)
```

Following is the sample .NET method SyncStats. It logs some of the statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string SyncStats(
  string user,
  int warnings,
  int errors,
  int deadLocks,
  int syncedTables,
  int connRetries ) {
  System.Console.WriteLine( "synch statistics
   number of deadlocks: " + deadlocks ;
  return( null );
}
```


synchronization_statistics table event

Tracks synchronization statistics.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username or s.remote_id	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2
s.warnings	INTEGER. The number of warnings that occurred for the table during the synchronization.	3
s.errors	INTEGER. The number of errors that were related to the table during the synchronization.	4

Default action

None.

Remarks

The synchronization_statistics event allows you to gather, for any user and table, the number of warnings and errors that occurred during synchronization. The synchronization_statistics table script is called just prior to the commit at the end of the end synchronization transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "download_statistics connection event" on page 300
- "download_statistics table event" on page 303
- "upload_statistics connection event" on page 401

- "upload_statistics table event" on page 405
- "synchronization_statistics connection event" on page 376
- "time_statistics connection event" on page 382
- "time_statistics table event" on page 385
- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts synchronization statistics into the sync_tab_audit table.

```
CALL ml_add_table_script(
  'ver1',
  'table1',
  'INSERT INTO sync_tab_audit (
    ml_user,
    table,
    warnings,
    errors)
   VALUES (
    {ml s.username},
    {ml s.table},
    {ml s.warnings},
    {ml s.errors} ) ' )
```

Once synchronization statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following call to a MobiLink system procedure registers a Java method called synchronizationStatisticsTable as the script for the synchronization_statistics table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'synchronization_statistics',
  'ExamplePackage.ExampleClass.synchronizationStatisticsTable'
)
```

Following is the sample Java method synchronizationStatisticsTable. It logs some of the statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String synchronizationStatisticsTable(
  String user,
  String table,
  int warnings,
  int errors ) {
  java.lang.System.out.println( "synch statistics for
   table: " + table + " errors: " + errors );
  return( null );
}
```


.NET example

The following call to a MobiLink system procedure registers a .NET method called SyncTableStats as the script for the synchronization_statistics table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'synchronization_statistics',
  'TestScripts.Test.SyncTableStats'
)
```

Following is the sample .NET method SyncTableStats. It logs some of the statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string SyncTableStats(
  string user,
  string table,
  int warnings,
  int errors ) {
  System.Console.WriteLine( "synch statistics for
   table: " + table + " errors: " + errors );
  return( null );
}
```


time_statistics connection event

Tracks time statistics by user and event.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.event_name	VARCHAR(128)	2
s.num_calls	INTEGER. The number of times the script was called.	3
s.min_time	INTEGER. Milliseconds. The shortest time it took to execute a script during this synchronization.	4
s.max_time	INTEGER. Milliseconds. The longest time it took to execute a script during this synchronization.	5
s.total_time	INTEGER. Milliseconds. The total time it took to execute all scripts in the synchronization. (This is not the same as the length of the synchronization.)	6

Default action

None.

Remarks

The time_statistics event allows you to gather time statistics for any user during synchronization. The statistics are gathered only for those events for which there is a corresponding script. The script gathers

aggregate data for occasions where a single event occurs multiple times. The script can be especially useful for time comparisons across users, events and tables.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "time_statistics table event" on page 385
- "download_statistics connection event" on page 300
- "download_statistics table event" on page 303
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405
- "synchronization_statistics connection event" on page 376
- "synchronization_statistics table event" on page 379
- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts statistical information into the time_statistics table.

```
CALL ml_add_connection_script(
  'ver1',
  'time_statistics',
  'INSERT INTO time_statistics (
    id,
    ml_user,
    table,
    event_name,
    num_calls,
    min_time,
    max_time,
    total_time)
   VALUES (
    ts_id.nextval,
    {ml s.username},
    {ml s.event_name},
    {ml s.num_calls},
    {ml s.min_time},
    {ml s.max_time},
    {ml s.total_time} ) ' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called timeStatisticsConnection as the script for the time_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
  'ver1',
  'time_statistics',
  'ExamplePackage.ExampleClass.timeStatisticsConnection' )
```

Following is the sample Java method timeStatisticsConnection. It prints statistics for the prepare_for_download event. (Note that printing statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String timeStatisticsConnection(
  String username,
```



```
    String table_name,
    String event_name,
    int num_calls,
    int min_time,
    int max_time,
    int total_time ) {
    if( event_name.equals( "prepare_for_download") ) {
      java.lang.System.out.println(
       "prepare_for_download num_calls: " + num_calls +
       "total_time: " + total_time );
    }
    return ( null );
  }
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called TimeStats as the script for the time_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'time_statistics',
  'TestScripts.Test.TimeStats'
)
```

Following is the sample .NET method TimeStats. It prints statistics for the prepare_for_download event. (Note that printing statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string TimeStats(
  string user,
  string eventName,
  int numCalls,
  int minTime,
  int maxTime,
  int totTime ) {
  if( event_name=="prepare_for_download") {
    System.Console.WriteLine(
     "prepare_for_download num_calls: " + num_calls +
     "total_time: " + total_time );
  }
  return ( null );
}
```


time_statistics table event

Tracks time statistics.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2
s.event_name	VARCHAR(128)	3
s.num_calls	INTEGER. The number of times the script was called.	4
s.min_time	INTEGER. Milliseconds. The shortest time it took to execute a script during the synchronization of this table.	5
s.max_time	INTEGER. Milliseconds. The longest time it took to execute a script during the synchronization of this table.	6
s.total_time	INTEGER. Milliseconds. The total time it took to execute all scripts in the synchronization of the table. (This is not the same as the length of the synchronization.)	7

Default action

None.

Remarks

The time_statistics table event allows you to gather time statistics for any user and table during synchronization. The statistics are gathered only for those events for which there is a corresponding script. The script gathers aggregate data for occasions where a single event occurs multiple times. The script can be especially useful for time comparisons across users, events and tables.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "time_statistics connection event" on page 382
- "download_statistics connection event" on page 300
- "download_statistics table event" on page 303
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405
- "synchronization_statistics connection event" on page 376
- "synchronization_statistics table event" on page 379
- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts statistical information into the time_statistics table.

```
CALL ml_add_table_script (
 'ver1',
 'table1',
 'time_statistics',
 'INSERT INTO time_statistics(
   ml_user,
   table,
   event_name,
   num_calls,
   min_time,
   max_time,
   total_time)
  VALUES (
   {ml s.username},
   {ml s.table},
   {ml s.event_name},
   {ml s.num_calls},
   {ml s.min_time},
   {ml s.max_time},
   {ml s.total_time} )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called timeStatisticsTable as the script for the time_statistics table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'time_statistics',
  'ExamplePackage.ExampleClass.timeStatisticsTable' )
```


Following is the sample Java method timeStatisticsTable. It prints statistics for the upload_old_row_insert event.

```
public String timeStatisticsConnection(
  String username,
  String table_name,
  String event_name,
  int num_calls,
  int min_time,
  int max_time,
  int total_time ) {
  if( event_name.equals( "upload_old_row_insert") ) {
    java.lang.System.out.println(
     "upload_old_row_insert num_calls: " + num_calls +
     "total_time: " + total_time );
  }
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called TimeTableStats as the script for the time_statistics table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'time_statistics',
  'TestScripts.Test.TimeTableStats'
)
```

Following is the sample .NET method TimeTableStats. It prints statistics for the upload_old_row_insert event.

```
public string TimeTableStats(
  string user,
  string table,
  string eventName,
  int numCalls,
  int minTime,
  int maxTime,
  int totTime ) {
  if( event_name == "upload_old_row_insert") {
    System.Console.WriteLine(
      "upload_old_row_insert num_calls: " + num_calls +
      "total_time: " + total_time );
  }
  return ( null );
}
```


upload_delete table event

Provides an event that the MobiLink server uses during processing of the upload to handle rows deleted from the remote database.

Parameters

Parameter name for SQL scripts	Order
r.*pk-column-1*	1
...	...
r.*pk-column-N*	*N*
r.*column-1*	*N* + 1
...	...
r.*column-M*	*N* + *M*

Default action

None.

Remarks

The statement-based upload_delete script handles rows that are deleted on the remote database. The action taken at the consolidated database can be a DELETE statement, but need not be.

You can have one upload_delete script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "upload_insert table event" on page 393
- "upload_update table event" on page 410

SQL example

This example is taken from the Contact sample and can be found in *Samples\MobiLink\Contact \build_consol.sql*. It marks customers that are deleted from the remote database as inactive.

```
CALL ml_add_table_script(
  'ver1',
  'table1',
  'upload_delete',
  'UPDATE Customer
   SET active = 0
   WHERE cust_id={ml r.cust_id}' )
```


Java example

The following call to a MobiLink system procedure registers a Java method called uploadDeleteTable as the script for the upload_delete table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_delete',
  'ExamplePackage.ExampleClass.uploadDeleteTable' )
```

Following is the sample Java method uploadDeleteTable. It calls genUD which dynamically generates an UPLOAD statement.

```
public String uploadDeleteTable() {
  return( genUD(_curTable) );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadDelete as the script for the upload_delete table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_delete',
  'TestScripts.Test.UploadDelete'
)
```

Following is the sample .NET method UploadDelete. It calls genUD which dynamically generates an UPLOAD statement.

```
public string UploadDelete( object pk1 ) {
  return( genUD(_curTable) );
}
```


upload_fetch table event

Fetches rows from a synchronized table in the consolidated database for the purpose of row-level conflict detection.

Parameters

Parameter name for SQL scripts	Order
r.*primary-key-1*	1
r.*primary-key-2*	2
…	…
r.*primary-key-N*	N

Default action

None.

Remarks

The statement-based upload_fetch script fetches rows from a synchronized table for the purposes of conflict detection. It is a companion to the upload_update event.

The columns of the result set must match the number of columns being uploaded from the remote database for this table. If the values returned do not match the pre-image in the uploaded row, a conflict is identified.

Do not use READPAST table hints in upload_fetch scripts. If the script skips a locked row using READPAST, the synchronization logic will think that the row was deleted. Depending on what scripts you have defined, this will either cause the uploaded update to be ignored or it will trigger conflict resolution. Ignoring the update is likely to be unacceptable behavior, and may be harmful. Triggering conflict resolution may not be a problem, depending on the resolution logic you have implemented.

You can have only one upload_fetch or upload_fetch_column_conflict script for each table in the remote database.

This script may be ignored if none of the following scripts are defined: upload_new_row_insert, upload_old_row_insert, and resolve_conflict.

See also

- “Script parameters” on page 221
- “Adding and deleting scripts” on page 228
- “Detecting conflicts” on page 114
- “resolve_conflict table event” on page 373
- “upload_delete table event” on page 388
- “upload_insert table event” on page 393
- “upload_update table event” on page 410

SQL example

The following SQL script is taken from the Contact sample and can be found in *Samples\MobiLink\Contact \build_consol.sql* in the SQL Anywhere installation. It is used to identify conflicts that occur when rows updated in the remote database Product table are uploaded. This script selects rows from a table also named Product, but depending on your consolidated and remote database schema, the two table names may not match.

```
CALL ml_add_table_script(
  'ver1',
  'table1',
  'upload_fetch',
  'SELECT id, name, size, quantity, unit_price
   FROM Product
   WHERE id={ml r.id}' )
```

Java example

This script must return valid SQL.

The following call to a MobiLink system procedure registers a Java method called uploadFetchTable as the script for the upload_fetch table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_fetch',
  'ExamplePackage.ExampleClass.uploadFetchTable' )
```

Following is the sample Java method uploadFetchTable. It calls genUF to dynamically generate an UPLOAD statement.

```
public String uploadFetchTable() {
  return( genUF(_curTable) );
}
```

.NET example

This script must return valid SQL.

The following call to a MobiLink system procedure registers a .NET method called UploadFetchTable as the script for the upload_fetch table event when synchronizing the script version ver1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_fetch',
  'TestScripts.Test.UploadFetchTable' )
```

Following is the sample .NET method UploadFetchTable. It calls GenUF to dynamically generate an UPLOAD statement.

```
public string UploadFetchTable() {
  return( GenUF(_curTable) );
}
```


upload_fetch_column_conflict table event

Fetches rows from a synchronized table in the consolidated database for the purpose of column-level conflict detection.

Parameters

Parameter name for SQL scripts	Order
r.*pk-column-1*	1
...	...
r.*pk-column-N*	*N*
r.*column-1*	*N* + 1
…	…
r.*column-M*	*N* + *M*

Default action

None.

Remarks

The statement-based upload_fetch_column_conflict script fetches columns from a synchronized table for the purposes of conflict detection. It is a companion to the upload_update event.

This script only detects a conflict when two users update the same column. Different users can update the same row, as long as they don't update the same column, without generating a conflict.

For example, using the upload_fetch_column_conflict script, you could avoid detecting a conflict when one of your remote users updated the quant column of the ULOrder table, and another remote user updated the notes column for the same row. You would only detect a conflict if they both updated the quant column.

You can have only one upload_fetch or upload_fetch_column_conflict script for each table in the remote database.

This script may be ignored if none of the following scripts are defined: upload_new_row_insert, upload_old_row_insert, and resolve_conflict.

See also

- “Script parameters” on page 221
- “Adding and deleting scripts” on page 228
- “Detecting conflicts” on page 114
- “upload_fetch table event” on page 390
- “resolve_conflict table event” on page 373
- “upload_delete table event” on page 388
- “upload_insert table event” on page 393
- “upload_update table event” on page 410

upload_insert table event

Provides an event that the MobiLink server uses during processing of the upload to handle rows inserted into the remote database.

Parameters

Parameter name for SQL scripts	Order
r.*pk-column-1*	1
...	...
r.*pk-column-N*	*N*
r.*column-1*	*N*+1
...	...
r.*column-M*	*N+M*

Default action

None.

Remarks

The statement based upload_insert script performs direct inserts of column values.

You can have one upload_insert script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- “Script parameters” on page 221
- “Adding and deleting scripts” on page 228
- “upload_delete table event” on page 388
- “upload_update table event” on page 410
- “upload_fetch table event” on page 390

SQL example

This example handles inserts that were made on the Customer table in the remote database. The script inserts the values into a table named Customer in the consolidated database. The final column of the table identifies the Customer as active. The final column does not appear in the remote database.

```
CALL ml_add_table_script(
 'ver1',
 'Customer',
 'upload_insert',
 'INSERT INTO Customer(
   cust_id,
   name,
   rep_id,
   active )
```



```
  VALUES (
   {ml r.cust_id},
   {ml r.name},
   {ml r.rep_id},
   1 )' )
```

Java example

The following call to a MobiLink system procedure registers a Java method called uploadInsertTable as the script for the upload_insert table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_insert',
  'ExamplePackage.ExampleClass.uploadInsertTable' )
```

Following is the sample Java method uploadInsertTable. It dynamically generates an UPLOAD statement. This syntax is for SQL Anywhere consolidated databases.

```
public String uploadInsertTable() {
  return("INSERT INTO " + _curTable + getCols(_curTable)
    + " VALUES " + getQM(_curTable));
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadInsert as the script for the upload_insert table event when synchronizing the script version ver1 and the table table1. This syntax is for SQL Anywhere consolidated databases.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_insert',
  'TestScripts.Test.UploadInsert'
)
```

Following is the sample .NET method UploadInsert. It returns an UPLOAD statement for the ULCustomer table.

```
public static string UploadInsert() {
   return("INSERT INTO ULCustomer( cust_id, cust_name ) VALUES ( {ml
r.cust_id}, {ml r.cust_name} )");
 }
```


upload_new_row_insert table event

Conflict resolution scripts for statement-based uploads commonly require access to the old and new values of rows uploaded from the remote database. This event allows you to handle the new, updated values of rows uploaded from the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name. This parameter is optional.	Optional
r.*pk-column-1*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	1 (2 if username is referenced)
...		...
r.*pk-column-N*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	*N* (*N*+1 if username is referenced)
r.*column-1*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	*N* + 1 (*N*+2 if username is referenced)
...	...	...
r.*column-M*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	*N* + *M* (*N*+*M*+1 if username is referenced)

Default action

None.

Remarks

When a MobiLink client sends an updated row to the MobiLink server, it includes not only the new values (the post-image), but also a copy of the old row values (the pre-image). When the pre-image does not match the current values in the consolidated database, a conflict is detected.

This event allows you to save post-image values to a table. You can use this event to assist in developing conflict resolution procedures for statement-based updates. The parameters for this event hold new row values from the remote database before the update is carried out on the corresponding consolidated database table. This event is also used to insert rows in statement-based, forced-conflict mode.

The script for this event is usually an insert statement that inserts the new row into a temporary table for use by a resolve_conflict script.

You can have one upload_new_row_insert script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Handling conflicts" on page 113
- "resolve_conflict table event" on page 373
- "upload_old_row_insert table event" on page 398
- "upload_update table event" on page 410
- "Forced conflicts" on page 121
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

This example handles updates made on the product table in the remote database. The script inserts the new value of the row into a global temporary table named product_conflict. The final column of the table identifies the row as a new row.

```
CALL ml_add_table_script(
 'ver1',
 'table1',
 'upload_new_row_insert',
 'INSERT INTO DBA.product_conflict(
   id,
   name,
   size,
   quantity,
   unit_price,
   row_type )
  VALUES(
   {ml r.id},
   {ml r.name},
   {ml r.size},
   {ml r.quantity},
   {ml r.unit_price},
   ''New'' )' )
```


Java example

The following call to a MobiLink system procedure registers a Java method called uploadNewRowInsertTable as the script for the upload_new_row_insert table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_new_row_insert',
  'ExamplePackage.ExampleClass.uploadNewRowInsertTable'
)
```

Following is the sample Java method uploadNewRowInsertTable. It dynamically generates an INSERT statement. This syntax is for SQL Anywhere consolidated databases.

```
public String uploadNewRowInsertTable() {
  return("insert into" + _curTable + "_new" +
    getCols(_curTable) + "values" + getNamedParams(_curTable));
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadNewRowInsertTable as the script for the upload_new_row_insert table event when synchronizing the script version ver1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_new_row_insert',
  'TestScripts.Test.UploadNewRowInsertTable'
)
```

Following is the sample .NET method UploadNewRowInsertTable. It dynamically generates an INSERT statement. This syntax is for SQL Anywhere consolidated databases.

```
public string UploadNewRowInsertTable() {
  return("insert into" + _curTable + "_new" +
    GetCols(_curTable) + "values" + GetNamedParams(_curTable));
}
```


upload_old_row_insert table event

Conflict resolution scripts for statement-based uploads commonly require access to the old and new values of rows uploaded from the remote database. This event allows you to handle the old values of rows uploaded from the remote database.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name. This parameter is optional.	Optional
r.*pk-column-1*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	1 (2 if username is referenced)
...		...
r.*pk-column-N*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	*N* (*N*+1 if username is referenced)
r.*column-1*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	*N* + 1 (*N*+2 if username is referenced)
...	...	...
r.*column-M*	A column value from the old (pre-image) row, where the named parameter is specified as a column name prefaced by **r.**	*N* + *M* (*N*+*M*+1 if username is referenced)

Default action

None.

Remarks

When a MobiLink client sends an updated row to the MobiLink server, it includes not only the new values (the post-image), but also a copy of the old row values (the pre-image). When the pre-image does not match the current values in the consolidated database, a conflict is detected.

This event allows you to save pre-image values to a table. You can use this event to assist in developing conflict resolution procedures for statement-based updates. The parameters for this event hold old row values from the remote database before the update is carried out on the corresponding consolidated database table. This event is also used to insert rows in statement-based, forced-conflict mode.

The script for this event is usually an insert statement that inserts the old row into a temporary table for use by a resolve_conflict script.

You can have one upload_old_row_insert script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Handling conflicts" on page 113
- "resolve_conflict table event" on page 373
- "upload_new_row_insert table event" on page 395
- "upload_update table event" on page 410
- "Forced conflicts" on page 121
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

This example handles updates made on the product table in the remote database. The script inserts the old value of the row into a global temporary table named product_conflict. The final column of the table identifies the row as an old row.

```
CALL ml_add_table_script(
 'ver1',
 'table1',
 'upload_old_row_insert',
 'INSERT INTO DBA.product_conflict (
   id,
   name,
   size,
   quantity,
   unit_price,
   row_type )
  VALUES (
   {ml r.id},
   {ml r.name},
   {ml r.size},
   {ml r.quantity},
   {ml r.unit_price},
   ''Old'' )' )
```


Java example

The following call to a MobiLink system procedure registers a Java method called uploadOldRowInsertTable as the script for the upload_old_row_insert table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_old_row_insert',
  'ExamplePackage.ExampleClass.uploadOldRowInsertTable'
)
```

Following is the sample Java method uploadOldRowInsertTable. It dynamically generates an INSERT statement.

```
public String uploadOldRowInsertTable() {
  return( "old" + getCols(_curTable) +
    "values" + getNamedParams(_curTable));
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadOldRowInsertTable as the script for the upload_old_row_insert table event when synchronizing the script version ver1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_old_row_insert',
  'TestScripts.Test.UploadOldRowInsertTable'
)
```

Following is the sample .NET method UploadOldRowInsertTable. It dynamically generates an UPLOAD statement.

```
public string UploadOldRowInsertTable() {
  return( "old" + GetCols(_curTable) +
    "values" + GetNamedParams(_curTable));
}
```


upload_statistics connection event

Tracks synchronization statistics for upload operations.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.warnings	INTEGER. The number of warnings that occurred.	2
s.errors	INTEGER. The number of errors that occurred.	3
s.inserted_rows	INTEGER. The number of rows that were successfully inserted in the consolidated database.	4
s.deleted_rows	INTEGER. The number of rows that were successfully deleted from the consolidated database.	5
s.updated_rows	INTEGER. The number of rows that were successfully updated in the consolidated database.	6
s.conflicted_inserts	INTEGER. Always zero.	7
s.conflicted_deletes	INTEGER. Always zero.	8
s.conflicted_updates	INTEGER. The number of update rows that caused conflict. A row is included only when a resolve conflict script was successfully called for it.	9

Parameter name for SQL scripts	Description	Order
s.ignored_inserts	INTEGER. The total number of upload insert rows that were ignored. They were ignored because 1) there is no upload_insert script in normal mode or no upload_new_row_insert script in forced conflict mode; or 2) errors occurred when the MobiLink server was invoking the corresponding script and the handle_error or handle_odbc_error event returned 1000.	10
s.ignored_deletes	INTEGER. The number of upload delete rows that caused errors while the upload_delete script was invoked, when the handle_error or handle_odbc_error are defined and returned 1000, or when there is no upload_delete script defined for the given table.	11
s.ignored_updates	INTEGER. The number of upload update rows that caused conflict but a resolve conflict script was not successfully called or no upload_update script was defined.	12
s.bytes	INTEGER. The amount of memory used within the MobiLink server to store the upload.	13
s.deadlocks	INTEGER. The number of deadlocks in the consolidated database that were detected for the synchronization.	14

Default action

None.

Remarks

The upload_statistics event allows you to gather, for any user, statistics on uploads. The upload_statistics connection script is called just prior to the commit at the end of the upload transaction.

See also

- “Script parameters” on page 221
- “Adding and deleting scripts” on page 228
- “download_statistics connection event” on page 300
- “download_statistics table event” on page 303
- “upload_statistics table event” on page 405
- “synchronization_statistics connection event” on page 376
- “synchronization_statistics table event” on page 379
- “time_statistics connection event” on page 382
- “time_statistics table event” on page 385
- “MobiLink Monitor” on page 145

- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL example

The following example inserts synchronization statistics for upload operations into the table upload_summary_audit.

```
CALL ml_add_connection_script (
 'ver1',
 'upload_statistics',
 'INSERT INTO upload_summary_audit (
   ml_user,
   warnings,
   errors,
   inserted_rows,
   deleted_rows,
   updated_rows,
   conflicted_inserts,
   conflicted_deletes,
   conflicted_updates,
   bytes,
   ignored_inserts,
   ignored deletes,
   ignored_updates,
   bytes, deadlocks )
  VALUES (
   {ml s.username},
   {ml s.warnings},
   {ml s.errors},
   {ml s.inserted_rows},
   {ml s.deleted_rows},
   {ml s.updated_rows},
   {ml s.conflicted_inserts},
   {ml s.conflicted_deletes},
   {ml s.conflicted_updates},
   {ml s.ignored_inserts},
   {ml s.ignored_deletes},
   {ml s.ignored_updates},
   {ml s.bytes},
   {ml s.deadlocks} ) ' )
```

Once statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following call to a MobiLink system procedure registers a Java method called uploadStatisticsConnection as the script for the upload_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_java_connection_script(
   'ver1',
   'upload_statistics',
   'ExamplePackage.ExampleClass.uploadStatisticsConnection' )
```

Following is the sample Java method uploadStatisticsConnection. It logs some statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String uploadStatisticsConnection(
  String user,
```



```
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictedInserts,
  int conflictedDeletes,
  int conflictedUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks ) {
  java.lang.System.out.println( "updated rows: " +
   updatedRows );
  return ( null );
}
```

.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadStats as the script for the upload_statistics connection event when synchronizing the script version ver1.

```
CALL ml_add_dnet_connection_script(
  'ver1',
  'upload_statistics',
  'TestScripts.Test.UploadStats'
)
```

Following is the sample .NET method UploadStats. It logs some statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string UploadStats (
  string user,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictInserts,
  int conflictDeletes,
  int conflictUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks ) {
  System.Console.WriteLine( "updated rows: " +
    updatedRows );
  return ( null );
}
```


upload_statistics table event

Tracks synchronization statistics for upload operations for a specific table.

Parameters

In the following table, the description provides the SQL data type. If you are writing your script in Java or .NET, you should use the appropriate corresponding data type. See "SQL-Java data types" on page 420 and "SQL-.NET data types" on page 466.

In SQL scripts, you can specify event parameters by name or with a question mark, but you cannot mix names and question marks within a script. If you use question marks, the parameters must be in the order shown below and are optional only if no subsequent parameters are specified (for example, you must use parameter 1 if you want to use parameter 2). If you use named parameters, you can specify any subset of the parameters in any order.

Parameter name for SQL scripts	Description	Order
s.remote_id	VARCHAR(128). The MobiLink remote ID. You can only reference the remote ID if you are using named parameters.	Not applicable
s.username	VARCHAR(128). The MobiLink user name.	1
s.table	VARCHAR(128). The table name.	2
s.warnings	INTEGER. The number of warnings issued in the upload of the table.	3
s.errors	INTEGER. The number of errors, including handled errors, that occurred in the upload of the table.	4
s.inserted_rows	INTEGER. The number of rows that were successfully inserted in the consolidated database.	5
s.deleted_rows	INTEGER. The number of rows that were successfully deleted from the consolidated database.	6
s.updated_rows	INTEGER.	7
s.conflicted_inserts	INTEGER. Always zero.	8
s.conflicted_deletes	INTEGER. Always zero.	9
s.conflicted_updates	INTEGER. The number of update rows that caused conflict. A row is included only when a resolve conflict script was successfully called for it.	10

Parameter name for SQL scripts	Description	Order
s.ignored_inserts	INTEGER. The total number of upload insert rows that were ignored. They were ignored because 1) there is no upload_insert script in normal mode or no upload_new_row_insert script in forced conflict mode; or 2) errors occurred when the MobiLink server was invoking the corresponding script and the handle_error or handle_odbc_error event returned 1000.	11
s.ignored_deletes	INTEGER. The number of upload delete rows that caused errors while the upload_delete script was invoked, when the handle_error or handle_odbc_error are defined and returned 1000, or when there is no upload_delete script defined for the given table.	12
s.ignored_updates	INTEGER. The number of upload update rows that caused conflict but a resolve conflict script was not successfully called or no upload_update script was defined.	13
s.bytes	INTEGER. The amount of memory used within the MobiLink server to store the upload.	14
s.deadlocks	INTEGER. The number of deadlocks in the consolidated database that were detected for the synchronization.	15

Default action

None.

Remarks

The upload_statistics event allows you to gather, for any user, vital statistics on synchronization happenings as they apply to any table. The upload_statistics table script is called just prior to the commit at the end of the upload transaction.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "download_statistics connection event" on page 300
- "upload_statistics connection event" on page 401
- "upload_statistics table event" on page 405
- "synchronization_statistics connection event" on page 376
- "synchronization_statistics table event" on page 379
- "time_statistics connection event" on page 382
- "time_statistics table event" on page 385

- "MobiLink Monitor" on page 145
- "Using remote IDs and MobiLink user names in scripts" [*MobiLink - Client Administration*]

SQL Example

The following example inserts a row into a table used to track upload statistics.

```
CALL ml_add_connection_script(
 'ver1',
 'upload_statistics',
 'INSERT INTO my_upload_statistics (
   user_name,
   table_name,
   num_warnings,
   num_errors,
   inserted_rows,
   deleted_rows,
   updated_rows,
   conflicted_inserts,
   conflicted_deletes,
   conflicted_updates,
   ignored_inserts,
   ignored_deletes,
   ignored_updates, bytes,
   deadlocks )
  VALUES(
   {ml s.username},
   {ml s.table},
   {ml s.warnings},
   {ml s.errors},
   {ml s.inserted_rows},
   {ml s.deleted_rows},
   {ml s.updated_rows},
   {ml s.conflicted_inserts},
   {ml s.conflicted_deletes},
   {ml s.conflicted_updates},
   {ml s.ignored_inserts},
   {ml s.ignored_deletes},
   {ml s.ignored_updates},
   {ml s.bytes},
   {ml s.deadlocks} )'  )
```

The following example works with an Oracle consolidated database.

```
CALL ml_add_connection_script(
 'ver1',
 'upload_statistics',
 'INSERT INTO upload_tables_audit (
  id,
  user_name,
  table,
  warnings,
  errors,
  inserted_rows,
  deleted_rows,
  updated_rows,
  conflicted_inserts,
  conflicted_deletes,
  conflicted_updates,
  ignored_inserts,
  ignored_deletes,
  ignored_updates,
```



```
    bytes,
    deadlocks )

  VALUES (
    ut_audit.nextval,
    {ml s.username},
    {ml s.table},
    {ml s.warnings},
    {ml s.errors},
    {ml s.inserted_rows},
    {ml s.deleted_rows},
    {ml s.updated_rows},
    {ml s.conflicted_inserts},
    {ml s.conflicted_deletes},
    {ml s.conflicted_updates},
    {ml s.ignored_inserts},
    {ml s.ignored_deletes},
    {ml s.ignored_updates},
    {ml s.bytes},
    {ml s.deadlocks} )' )
```

Once statistics are inserted into the audit table, you may use these statistics to monitor your synchronizations and make optimizations where applicable.

Java example

The following call to a MobiLink system procedure registers a Java method called uploadStatisticsTable as the script for the upload_statistics table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_statistics',
  'ExamplePackage.ExampleClass.uploadStatisticsTable' )
```

Following is the sample Java method uploadStatisticsTable. It logs some statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public String uploadStatisticsTable(
  String user,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictedInserts,
  int conflictedDeletes,
  int conflictedUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks ) {
  java.lang.System.out.println( "updated rows: " +
    updatedRows );
  return ( null );
}
```


.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadTableStats as the script for the upload_statistics table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_statistics',
  'TestScripts.Test.UploadTableStats'
)
```

Following is the sample .NET method uploadStatisticsTable. It logs some statistics to the MobiLink output log. (Note that logging statistics to the MobiLink output log might be useful at development time but would slow down a production server.)

```
public string UploadTableStats(
  string user,
  string table,
  int warnings,
  int errors,
  int insertedRows,
  int deletedRows,
  int updatedRows,
  int conflictInserts,
  int conflictDeletes,
  int conflictUpdates,
  int ignoredInserts,
  int ignoredDeletes,
  int ignoredUpdates,
  int bytes,
  int deadlocks ) {
  System.Console.WriteLine( "updated rows: " +
    updatedRows );
  return ( null );
}
```

upload_update table event

Provides an event that the MobiLink server uses during processing of the upload to handle rows updated at the remote database.

Parameters

Parameter	Order
r.*column-1*	1
...	...
r.*column-M*	M
r.*pk-column-1*	M + 1
...	...
r.*pk-column-N*	M + N
o.*column-N*	M + N + 1
...	...
o.*column-M*	M + N + M

Default action

None.

Remarks

The statement-based upload_update script may perform direct updates of column values as specified in the UPLOAD statement.

The WHERE clause must include all of the primary key columns that are being synchronized. The SET clause must contain all of the non-primary key columns that are being synchronized.

You use as many non-primary key columns in your SET clause as exist in the table, and MobiLink will send the correct number of column values. Similarly, in the WHERE clause, you can have any number of primary keys, but all must be specified here, and MobiLink will send the correct values. MobiLink sends these column values and primary key values in the order the columns or primary keys appear in a MobiLink report of your schema. You can use the -vh option to determine the column ordering for this table schema.

For example, in the following upload_update script, the question marks are in good order:

```
UPDATE MyTable
  SET column_1 = ?, . . ., column_M = ?
  WHERE pk_column_1 = ? AND ... AND pk_column_N = ?
```

You can have one upload_update script for each table in the remote database.

For Java and .NET applications, this script must return valid SQL.

To use the upload_update script to detect conflicts, include all non-primary key columns in the WHERE clause:

UPDATE *table-name*
SET *col1* **= {ml r.***col1***},** *col2* **= {ml r.** *col2* **}** …
WHERE *pk1* **= {ml r.***pk1***} AND** *pk2* **= {ml r.***pk2***}** …
AND *col1* **= {ml o.***col1***} AND** *col2* **={ml o.***col2***}** …

In this statement, col1 and col2 are the non-primary key columns, while pk1 and pk2 are primary key columns. The values passed to the second set of non-primary key columns are the pre-image of the updated row. The WHERE clause compares old values uploaded from the remote to current values in the consolidated database. If the values do not match, the update is ignored, preserving the values already on the consolidated database.

See also

- "Script parameters" on page 221
- "Adding and deleting scripts" on page 228
- "Detecting conflicts with upload_update scripts" on page 115
- "Resolving conflicts with upload_update scripts" on page 118
- "upload_delete table event" on page 388
- "upload_fetch table event" on page 390
- "upload_insert table event" on page 393

SQL example

This example handles updates made to the Customer table in the remote database. The script updates the values in a table named Customer in the consolidated database.

```
CALL ml_add_table_script(
  'ver1',
  'table1',
  'upload_update',
'UPDATE Customer
   SET name = {ml r.name}, rep_id = {ml r.rep_id}
      WHERE cust_id = {ml o.cust_id}')
```

Java example

The following call to a MobiLink system procedure registers a Java method called uploadUpdateTable as the script for the upload_update table event when synchronizing the script version ver1.

```
CALL ml_add_java_table_script(
  'ver1',
  'table1',
  'upload_update',
  'ExamplePackage.ExampleClass.uploadUpdateTable' )
```

Following is the sample Java method uploadUpdateTable. It calls a method called genUU to dynamically generate an UPLOAD statement.

```
public String uploadUpdateTable() {
  return( genUU(_curTable) );
}
```


.NET example

The following call to a MobiLink system procedure registers a .NET method called UploadUpdate as the script for the upload_update table event when synchronizing the script version ver1 and the table table1.

```
CALL ml_add_dnet_table_script(
  'ver1',
  'table1',
  'upload_update',
  'TestScripts.Test.UploadUpdate'
)
```

Following is the sample .NET method UploadUpdate. It calls a method called GenUU to dynamically generate an UPLOAD statement.

```
public string UploadUpdate() {
  return ( genUU(_curTable) );
}
```

Part III. MobiLink Server APIs

This part describes the MobiLink Server APIs for Java and .NET.

CHAPTER 11

Writing Synchronization Scripts in Java

Contents

Introduction to Java synchronization logic

You control the actions of the MobiLink server by writing synchronization scripts. You can implement these scripts in SQL, .NET or Java. Java synchronization logic can function just as SQL logic functions: the MobiLink server can make calls to Java methods on the occurrence of MobiLink events just as it accesses SQL scripts on the occurrence of MobiLink events. A Java method can return a SQL string to MobiLink.

This section tells you how to set up, develop, and run Java synchronization logic. It includes a sample application and the MobiLink server API for Java Reference.

See also

- "Tutorial: Using Java Synchronization Logic" [*MobiLink - Getting Started*]
- "Options for writing synchronization logic" [*MobiLink - Getting Started*]
- "Writing Synchronization Scripts" on page 213

Setting up Java synchronization logic

When you install SQL Anywhere, the installer automatically sets the location of the MobiLink server API for Java classes. When you start the MobiLink server, it automatically includes these classes in your classpath. The MobiLink server API for Java classes are installed to the *java\mlscript.jar* subdirectory of your SQL Anywhere installation.

♦ To implement synchronization scripts in Java

1. Create your own class or classes. Write a method for each required synchronization script. These methods must be public. The class must be public in the package.

   See "Methods" on page 421.

   Each class with non-static methods should have a public constructor. The MobiLink server automatically instantiates each class the first time a method in that class is called.

   See "Constructors" on page 420.

2. When compiling the class, you must include the JAR file *java\mlscript.jar*.

   For example,

   ```
   javac MyClass.java -classpath "c:\Program Files\SQL Anywhere 10\java
   \mlscript.jar"
   ```

3. In the MobiLink system tables on your consolidated database, specify the name of the package, class, and method to call for each synchronization script. One class is permitted per script version.

   For example, you can add this information to the MobiLink system tables using the ml_add_java_connection_script stored procedure or the ml_add_java_table_script stored procedure.

   For example, the following SQL statement, when run in a SQL Anywhere database, specifies that for the script version ver1, myPackage.myClass.myMethod should be run whenever the authenticate_user connection-level event occurs. The method that is specified must be the fully qualified name of a public Java method, and the name is case sensitive.

   ```
   call ml_add_java_connection_script( 'ver1',
   'authenicate_user', 'myPackage.myClass.myMethod' )
   ```

   For more information about adding scripts, see:

   - "System procedures to add or delete scripts" on page 532
   - "ml_add_java_connection_script" on page 536
   - "ml_add_java_table_script" on page 537

4. Instruct the MobiLink server to load classes. A vital part of setting up Java synchronization logic is to tell the virtual machine where to look for Java classes. There are two ways to do this:

   - Use the mlsrv10 -sl java -cp option to specify a set of directories or jar files in which to search for classes. For example, at the command line, enter:

     ```
     mlsrv10 -c "dsn=consolidated1" -sl java (-cp %classpath%;c:\local\Java
     \myclasses.jar)
     ```


The MobiLink server automatically appends the location of the MobiLink server API for Java classes (java\mlscript.jar) to the set of directories or jar files. The -sl java option also forces the Java VM to load on server startup.

For more information about the available Java options, see "-sl java option" on page 66.

- Explicitly set the classpath. To set the classpath for user-defined classes, use a statement such as the following:

  ```
  SET classpath=%classpath%;c:\local\Java\myclasses.jar
  ```

  If your system classpath includes your Java synchronization logic classes, you do not need to make changes to your MobiLink server command line.

  You can use the -sl java option to force the Java virtual machine to load at server startup. Otherwise, the Java virtual machine is started when the first Java method is executed.

  For more information about the available Java options, see "-sl java option" on page 66.

5. On UNIX, if you want to load a specific JRE, you should set the LD_LIBRARY_PATH (LIBPATH on AIX, SHLIB_PATH on HP-UX) to include the directory containing the JRE. The directory must be listed before any of the SQL Anywhere installation directories.

See also

- "Writing Java synchronization logic" on page 419
- "Java synchronization example" on page 426
- "Tutorial: Using Java Synchronization Logic" [*MobiLink - Getting Started*]
- "MobiLink server API for Java Reference" on page 431
- "Options for writing synchronization logic" [*MobiLink - Getting Started*]
- "Writing Synchronization Scripts" on page 213

Writing Java synchronization logic

Writing Java synchronization logic requires knowledge of MobiLink events, some knowledge of Java, and knowledge of the MobiLink server API for Java.

For a complete description of the API, see "MobiLink server API for Java Reference" on page 431.

Java synchronization logic can be used to maintain state information, and implement logic around the upload and download events. For example, a begin_synchronization script written in Java could store the MobiLink user name in a variable. Scripts called later in the synchronization process can access this variable. Also, you can use Java to access rows in the consolidated database, before or after they are committed.

Using Java reduces dependence on the consolidated database. Behavior is affected less by upgrading the consolidated database to a new version or switching to a different database management system.

Direct row handling

You can use MobiLink direct row handling to communicate remote data to any central data source, application, or web service. Direct row handling uses special classes in the MobiLink server APIs for Java or .NET for direct access to synchronized data.

See "Direct Row Handling" on page 517.

Class instances

The MobiLink server instantiates your classes at the connection level. When an event is reached for which you have written a non-static Java method, the MobiLink server automatically creates an instance of the class, if it has not already done so on the present connection.

See "Constructors" on page 466.

All methods directly associated with a connection-level or table-level event for one script version *must belong to the same class*.

For each database connection, once a class has been instantiated, the class persists until that connection is closed. Thus, the same instance may well be used for multiple consecutive synchronization sessions. Information present in public or private variables will thus persist across synchronizations that occur on the same connection unless explicitly cleared.

You can also use static classes or variables. In this case, the values are available across all connections.

The MobiLink server automatically deletes your class instances only when the connection to the consolidated database is closed.

Transactions

The normal rules regarding transactions apply to Java methods. The start and duration of database transactions is critical to the synchronization process. Transactions must be started and ended only by the

MobiLink server. Explicitly committing or rolling back transactions on the synchronization connection within a Java method violates the integrity of the synchronization process and can cause errors.

These rules apply only to the database connections created by the MobiLink server and, in particular, to SQL statements returned by methods. If your classes create other database connections, you are free to manage them as you please.

SQL-Java data types

The following table shows SQL data types and the corresponding Java data types.

SQL data type	Corresponding Java data type
VARCHAR	java.lang.String
CHAR	java.lang.String
INTEGER	int or Integer
BINARY	byte[ ]
TIMESTAMP	java.sql.Timestamp
INOUT INTEGER	ianywhere.ml.script.InOutInteger
INOUT VARCHAR	ianywhere.ml.script.InOutString
INOUT CHAR	ianywhere.ml.script.InOutString
INOUT BYTEARRAY	ianywhere.ml.script.InOutByteArray

The MobiLink server automatically adds the ianywhere.ml.script package to your classpath if it is not already present. However, when you compile your class you need to add the path of *java\mlscript.jar*, located in your SQL Anywhere installation directory.

Constructors

The constructor of your class may have one of two possible signatures.

```
public MyScriptClass (
ianywhere.ml.script.DBConnectionContext sc )
```

or

```
public MyScriptClass ( )
```

The synchronization context passed to you is for the connection through which the MobiLink server is synchronizing the current user.

The DBConnectionContext.getConnection method returns the same database connection that MobiLink is using to synchronize the present user. You can execute statements on this connection, but you must not commit or roll back the transaction. The MobiLink server manages the transactions.

The MobiLink server prefers to use constructors with the first signature. It only uses the non-argument constructor if a constructor with the first signature is not present.

See "DBConnectionContext interface" on page 431.

Methods

In general, you implement one method for each synchronization event. These methods must be public. If they are private, the MobiLink server cannot use them and will fail to recognize that they exist.

The names of the methods are not important, as long as the names match the names specified in the ml_script table in the consolidated database. In the examples included in the documentation, however, the method names are the same as those of the MobiLink events because this naming convention makes the Java code easier to read.

The signature of your method should match the signature of the script for that event, except that you can truncate the parameter list if you do not need the values of parameters at the end of the list. You should accept only the parameters you need, because overhead is associated with passing the parameters.

You cannot, however, overload the methods. Only one method prototype per class may appear in the ml_script system table.

Registering methods

After creating a method, you must register it. Registering the method creates a reference to the method in the MobiLink system tables on the consolidated database, so that the method is called when the event occurs. You register methods in the same way that you add synchronization scripts, except instead of adding the entire SQL script to the MobiLink system table, you add only the method name.

See "Adding and deleting scripts" on page 228.

Return values

Methods called for a SQL-based upload or download must return a valid SQL language statement. The return type of these methods must be java.lang.String. No other return types are allowed.

The return type of all other scripts must either be java.lang.String or void. No other types are allowed. If the return type is a string and not null, the MobiLink server assumes that the string contains a valid SQL statement and executes this statement in the consolidated database as it would an ordinary SQL-language synchronization script. If a method ordinarily returns a string but does not want to execute a SQL statement against the database upon its return, it can return null.

Debugging Java classes

MobiLink provides various information and facilities that you may find helpful when debugging your Java code. This section describes where you can find this information and how you can exploit these capabilities.

Information in the MobiLink server's log file

The MobiLink server writes messages to an output log file. The synchronization server log file contains the following information:

- The Java Runtime Environment. You can use the -jrepath option to request a particular JRE when you start the MobiLink server. The default path is the path of the JRE installed with SQL Anywhere 10.
- The path of the standard Java classes loaded. If you did not specify these explicitly, the MobiLink server automatically adds them to your classpath before invoking the Java virtual machine.
- The fully specified names of the specific methods invoked. You can use this information to verify that the MobiLink server is invoking the correct methods.
- Any output written in a Java method to java.lang.System.out or java.lang.System.err is redirected to the MobiLink server log file.
- The mlsrv10 command line option -verbose can be used.

  See "-v option" on page 73.

Using a Java debugger

You can debug your Java classes using a standard Java debugger. Specify the necessary parameters using the -sl java option on the mlsrv10 command line.

See "-sl java option" on page 66.

Specifying a debugger causes the Java virtual machine to pause and wait for a connection from a Java debugger.

Printing information from Java

Alternatively, you may choose to add statements to your Java methods that print information to the MobiLink output log, using java.lang.System.err or java.lang.System.out. Doing so can help you track the progress and behavior of your classes.

> **Performance tip**
> Printing information in this manner is a useful monitoring tool, but is not recommended in a production scenario.

The same technique can be exploited to log arbitrary synchronization information or collect statistical information on how your scripts are used.

Writing your own test driver

You may want to write your own driver to exercise your Java classes. This approach can be helpful because it isolates the actions of your Java methods from the rest of the MobiLink system.

Handling MobiLink server errors in Java

When scanning the log is not sufficient, you can monitor your applications programmatically. For example, you can send messages of a certain type in an email.

You can write methods that are passed a class representing every error or warning message that is printed to the log. This may help you monitor and audit a MobiLink server.

The following code installs a LogListener for all warning messages, and writes the information to a file.

```
class TestLogListener implements LogListener {
  FileOutputStream _out_file;
  public TestLogListener( FileOutputStream out_file ) {
    _out_file       = out_file;
  }


  public void messageLogged(  ServerContext   sc,
    LogMessage msg ) {
    String  type;
    String  user;
    try {
      if(msg.getType() == LogMessage.ERROR) {
        type = "ERROR";
      } else if(msg.getType() == LogMessage.WARNING) {
        type = "WARNING";
      } else {
        type = "UNKNOWN!!!";
      }


      user = msg.getUser();
      if( user == null ) {
        user = "NULL";
      }
      _out_file.write(
        ("Caught msg type=" + type +
         " user=" + user +
         " text=" +msg.getText() +
         "\n").getBytes() );
      _out_file.flush();
    } catch( Exception e ) {
      // Print some error output to the MobiLink log.
      e.printStackTrace();
    }
  }
}
```

The following code registers TestLogListener to receive warning messages. Call this code from anywhere that has access to the ServerContext such as a class constructor or synchronization script.

```
// ServerContext serv_context;
serv_context.addWarningListener(
      new MyLogListener( ll_out_file ));
```

See also

- “addErrorListener method” on page 448
- “removeErrorListener method” on page 448

- "addWarningListener method" on page 449
- "removeWarningListener method" on page 449
- "LogListener interface" on page 443
- "LogMessage class" on page 444

User-defined start classes

You can define start classes that are loaded automatically when the server is started. The purpose of this feature is to allow you to write Java code that executes at the time the MobiLink server starts the JVM—before the first synchronization. This means you can create connections or cache data before a user synchronization request.

You do this with the DMLStartClasses option of the mlsrv10 -sl java option. For example, the following is part of a mlsrv10 command line. It causes mycl1 and mycl2 to be loaded as start classes.

```
-sl java(-DMLStartClasses=com.test.mycl1,com.test.mycl2)
```

Classes are loaded in the order in which they are listed. If the same class is listed more than once, more than one instance is created.

All start classes must be public and must have a public constructor that either accepts no arguments or accepts one argument of type ianywhere.ml.script.ServerContext.

The names of loaded start classes are output to the MobiLink log with the message "Loaded JAVA start class: *classname*".

For more information about Java virtual machine options, see "-sl java option" on page 66.

To see the start classes that are constructed at server start time, see "getStartClassInstances method" on page 446.

Example

Following is a template start class. It starts a daemon thread that processes events and creates a database connection. (Not all start classes will need to create a thread but if a thread is spawned it should be a daemon thread.)

```
import ianywhere.ml.script.*;
import java.sql.*;

public class StartTemplate extends
   Thread implements ShutdownListener {
  ServerContext   _sc;
  Connection      _conn;
  boolean         _exit_loop;

  public StartTemplate( ServerContext sc )
    throws SQLException {
    // Perform setup first so that an exception
    // causes MobiLink startup to fail.
    _sc       = sc;
    // Create a connection for use later.
    _conn      = _sc.makeConnection();
    _exit_loop = false;
    setDaemon( true );
```



```
      start();
    }

    public void run() {
      _sc.addShutdownListener( this );
      // run() cannot throw exceptions.
      try {
         handlerLoop();
         _conn.close();
         _conn = null;
      } catch( Exception e ) {
         // Print some error output to the MobiLink log.
         e.printStackTrace();
         // This thread shuts down and so does not
         // need to be notified of shutdown.
         _sc.removeShutdownListener( this );
         // Ask server to shutdown so that this fatal
         // error will be fixed.
         _sc.shutdown();
      }
      // Shortly after return this thread will no longer
      // exist.
      return;
    }


    // stop our event handler loop
    public void shutdownPerformed( ServerContext sc ) {
      try {
         // Wait max 10 seconds for thread to die.
         join( 10*1000);
      } catch( Exception e ) {
         // Print some error output to the MobiLink log.
         e.printStackTrace();
      }
    }

    private void handlerLoop()
      throws InterruptedException {
      while( !_exit_loop ) {
        // Handle events in this loop. Sleep not
        // needed, block on event queue.
        sleep( 1*1000 );
      }
    }
  }
```

Java synchronization example

Java synchronization logic works with MobiLink and common Java classes to provide you with flexibility in deploying applications using MobiLink server. The following section introduces you to this extended range of functionality using a simple example.

This section describes a working example of Java synchronization logic. Before you try to use this class or write your own class, use the following checklist to ensure you have all the pieces in place before compiling the class.

- Plan your desired functionality using, for example, pseudocode.
- Create a map of database tables and columns.
- Configure the consolidated database for Java synchronization by ensuring you have specified in the MobiLink system tables the language type and location of the Java synchronization methods.

  See "Setting up Java synchronization logic" on page 417.
- Create a list of associated Java classes that are called during the running of your Java class.
- Store your Java classes in a location that is in the classpath for MobiLink server.

Plan

The Java synchronization logic for this example points to the associated Java files and classes that contain functionality needed for the example to work. It will show you how to create a class CustEmpScripts. It shows you how to set up a synchronization context for the connection. Finally, the example provides Java methods to

- Authenticate a MobiLink user
- Perform download and upload operations using cursors for each database table.

Schema

The tables to be synchronized are emp and cust. The emp table has three columns called emp_id, emp_name and manager. The cust table has three columns called cust_id, cust_name and emp_id. All columns in each table are synchronized. The mapping from consolidated to remote database is such that the table names and column names are identical in both databases. One additional table, an audit table, is added to the consolidated database.

Java class files

The files used in the example are included in the *Samples\MobiLink\JavaAuthentication* directory.

Setup

The following code sets up the Java synchronization logic. The import statements tell the Java virtual machine the location of needed files. The public class statement declares the class.

```
// Use a package when you create your own script.
import ianywhere.ml.script.InOutInteger;
import ianywhere.ml.script.DBConnectionContext;
```



```
import ianywhere.ml.script.ServerContext;
import java.sql.*;
public class CustEmpScripts {
  // Context for this synchronization connection.
  DBConnectionContext _conn_context;
  // Same connection MobiLink uses for sync.
  // Do not commit or close this.
  Connection _sync_connection;
  Connection _audit_connection;
  //Get a user id given the user name. On audit connection.
  PreparedStatement _get_user_id_pstmt;
  // Add record of user logins added. On audit connection.
  PreparedStatement _insert_login_pstmt;
  // Prepared statement to add a record to the audit table.
  // On audit connection.
  PreparedStatement _insert_audit_pstmt;
  // ...
}
```

The CustEmpScripts constructor sets up all the prepared statements for the authenticateUser method. It sets up member data.

```
public CustEmpScripts( DBConnectionContext cc )
  throws SQLException {
  try {
    _conn_context = cc;
    _sync_connection = _conn_context.getConnection();

    ServerContext serv_context =
    _conn_context.getServerContext();
    _audit_connection = serv_context.makeConnection();

    // Get the prepared statements ready.
    _get_user_id_pstmt =
    _audit_connection.prepareStatement(
        "select user_id from ml_user where name = ?"
        );
    _insert_login_pstmt =


    _audit_connection.prepareStatement(
      "insert into login_added( ml_user, add_time )
      " + " values( ?, { fn CONVERT( { fn NOW() },
      SQL_VARCHAR ) })"
      );
    _insert_audit_pstmt =
      _audit_connection.prepareStatement(
      "insert into login_audit( ml_user_id,
      audit_time, audit_action ) " +
      " values( ?, { fn CONVERT( { fn NOW() },
      SQL_VARCHAR ) }, ? ) "
      );
  } catch ( SQLException e ) {
    freeJDBCResources();
    throw e;
  } catch ( Error e ) {
    freeJDBCResources();
    throw e;
  }
}
```

The finalize method cleans up JDBC resources if end_connection is not called. It calls the freeJDBCResources method, which frees allocated memory and closes the audit connection.


```
protected void finalize()
  throws SQLException, Throwable {
  super.finalize();
  freeJDBCResources();
}

private void freeJDBCResources()
  throws SQLException {
  if( _get_user_id_pstmt != null ) {
    _get_user_id_pstmt.close();
  }
  if( _insert_login_pstmt != null ) {
    _insert_login_pstmt.close();
  }
  if( _insert_audit_pstmt != null ) {
    _insert_audit_pstmt.close();
  }
  if( _audit_connection != null ) {
    _audit_connection.close();
  }
  _conn_context        = null;
  _sync_connection    = null;
  _audit_connection   = null;
  _get_user_id_pstmt  = null;
  _insert_login_pstmt = null;
  _insert_audit_pstmt = null;
}
```

The endConnection method cleans up resources once the resources are not needed.

```
public void endConnection()
  throws SQLException {
  freeJDBCResources();
}
```

The authenticateUser method below approves all user logins and logs user information to database tables. If the user is not in the ml_user table they are logged to login_added. If the user id is found in ml_user then they are logged to login_audit. In a real system you would not ignore the user_password, but this sample approves all users for simplicity. The endConnection method throws SQLException if any of the database operations fail with an exception.

```
public void authenticateUser(
  InOutInteger authentication_status,
  String user_name )
  throws SQLException {
    boolean new_user;
    int user_id;

    // Get ml_user id.
    _get_user_id_pstmt.setString( 1, user_name );
    ResultSet user_id_rs =
    _get_user_id_pstmt.executeQuery();
    new_user = !user_id_rs.next();
    if( !new_user ) {
      user_id = user_id_rs.getInt(1);
    } else {
      user_id = 0;
    }


  user_id_rs.close();
  user_id_rs = null;
```



```
    // In this tutorial always allow the login.
    authentication_status.setValue( 1000 );
    if( new_user ) {
      _insert_login_pstmt.setString( 1, user_name );
      _insert_login_pstmt.executeUpdate();
      java.lang.System.out.println( "user: " +
        user_name + " added. " );
    } else {
      _insert_audit_pstmt.setInt( 1, user_id );
      _insert_audit_pstmt.setString( 2, "LOGIN ALLOWED" );
      _insert_audit_pstmt.executeUpdate();
    }
    _audit_connection.commit();
    return;
  }
```

The following methods use SQL code to act as cursors on the database tables. Since these are cursor scripts, they must return a SQL string.

```
  public static String empUploadInsertStmt() {
    return( "INSERT INTO emp(
      emp_id, emp_name) VALUES( ?, ?) " );
  }

  public static String empUploadDeleteStmt() {
    return( "DELETE FROM emp WHERE emp_id = ?" );
  }

  public static String empUploadUpdateStmt() {
    return( "UPDATE emp SET emp_name = ?
      WHERE emp_id = ? " );
  }

  public static String empDownloadCursor() {
    return( "SELECT emp_id, emp_name FROM emp" );
  }

  public static String custUploadInsertStmt() {
    return( "INSERT INTO cust(
      cust_id, emp_id, cust_name)
      VALUES ( ?, ?, ? ) " );
  }

  public static String custUploadDeleteStmt() {
    return( "DELETE FROM cust WHERE cust_id = ? " );
  }

  public static String custUploadUpdateStmt() {
    return( "UPDATE cust
      SET emp_id = ?, cust_name = ?
      WHERE cust_id = ? " );
  }

  public static String custDownloadCursor() {
    return( "SELECT cust_id, emp_id, cust_name
      FROM cust" );
  }
```

Use the following command to compile the code:

```
javac -cp %sqlany10%\java\mlscript.jar CustEmpScripts.jar
```


Run the MobiLink server with the location of CustEmpScripts.class in the classpath. Following is a partial command line:

```
mlsrv10 ... -sl java (-cp <class_location>)
```


MobiLink server API for Java Reference

This section explains the MobiLink Java interfaces and classes, and their associated methods and constructors. To use these classes, reference the assembly *\java\mlscript.jar* in your SQL Anywhere installation directory.

DBConnectionContext interface

Synopsis

public **ianywhere.ml.script.DBConnectionContext**
Interface for obtaining and accessing information about the current database connection. A DBConnectionContext instance is passed to the constructor of classes containing scripts. If context is required for a background thread or beyond the lifetime of a connection, use the ServerContext class.

See also

- "Constructors" on page 420
- "ServerContext interface" on page 445

Example

The following example shows you how to create a class level DBConnectionContext instance to use in your synchronization scripts. The DBConnectionContext getConnection method obtains a java.sql.Connection instance representing the current connection with the MobiLink consolidated database.

```
import ianywhere.ml.script.*;
import java.io.*;
import java.sql.*;

public class OrderProcessor {

  DBConnectionContext _cc;

  public OrderProcessor( DBConnectionContext cc ) {
    _cc = cc;
  }

  // The method used for the handle_DownloadData event.
  public void HandleEvent()
    throws SQLException {
    java.sql.Connection my_connection = _cc.getConnection();
     // ...
  }

 // ...

}
```

Caution
A DBConnectionContext instance should not be used outside the thread that calls into your Java code.

getConnection method

Syntax

public java.sql.Connection **getConnection( )**
throws java.sql.SQLException

Remarks

Returns the existing connection with the MobiLink consolidated database as a JDBC connection. The java.sql.Connection object returned by this method represents same connection that the MobiLink server uses to execute SQL scripts.

This connection must not be committed, closed or altered in any way that would affect the MobiLink server use of this connection. The connection returned is only valid for the lifetime of the underlying MobiLink connection. Do not use the connection after the end_connection event has been called for that connection.

If an error occurs binding the existing connection as a JDBC connection then it throws java.sql.SQLException

If a server connection with full access is required, use ServerContext.makeConnection().

Returns

The existing connection with the MobiLink consolidated database as a JDBC connection.

getDownloadData method

Syntax

public DownloadData **getDownloadData()**

Remarks

Returns a DownloadData instance for the current synchronization. Use the DownloadData class to create the download for direct row handling.

Returns

A DownloadData instance for the current synchronization.

See also

- "DownloadData interface" on page 434
- "Direct Row Handling" on page 517

Example

The following example shows you how to obtain a DownloadData instance for the current synchronization using the DBConnectionContext getDownloadData method.

```
DBConnectionContext _cc;

// Your class constructor.
public OrderProcessor( DBConnectionContext cc ) {
 _cc = cc;
```



```
}

// The method used for the handle_DownloadData event.
public void HandleDownload() throws SQLException {
  // get the DownloadData for the current synchronization
  DownloadData my_dd = _cc.getDownloadData();

  // ...

}

// ...
```

getProperties method

Syntax

public java.util.Properties **getProperties( )**

Remarks

Returns the properties for this connection, based on this connection's script version. Properties are stored in the ml_property table.

Returns

The properties for this connection.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

getRemoteID method

Syntax

public java.lang.String **getRemoteID( )**

Remarks

Returns the remote ID of the database currently synchronizing on this connection. If your remote database is prior to version 10, it returns the MobiLink user name.

Returns

The remote ID.

See also

- "Remote IDs" [*MobiLink - Client Administration*]

getServerContext method

Syntax

public ServerContext **getServerContext( )**

Remarks

Returns the ServerContext for this MobiLink server.

Returns

The ServerContext for this MobiLink server.

getVersion method

Syntax

public java.lang.String **getVersion( )**

Remarks

Returns the script version string.

Returns

The script version string.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

DownloadData interface

Encapsulates download data operations for direct row handling. To obtain a DownloadData instance, use the DBConnectionContext getDownloadData method.

Use the DownloadData.getDownloadTables and getDownloadTableByName methods to return DownloadTableData instances.

This download data is available through DBConnectionContext. It is not valid to access the download data before the begin_synchronization event or after the end_download event. It is not valid to access DownloadData in an upload-only synchronization.

See also

- "DownloadTableData interface" on page 437
- "handle_DownloadData connection event" on page 335
- DBConnectionContext "getDownloadData method" on page 432
- "Direct Row Handling" on page 517

Example

The following example shows you how to obtain a DownloadData instance for the current synchronization using the DBConnectionContext getDownloadData method.

```
DBConnectionContext _cc;

// Your class constructor.
public OrderProcessor( DBConnectionContext cc ) {
  _cc = cc;
}

// The method used for the handle_DownloadData event.
public void HandleDownload() throws SQLException {
  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.getDownloadData();

  // ...

}

// ...
```

getDownloadTableByName method

Syntax

public DownloadTableData getDownloadTableByName(
string *table-name*);

Remarks

Gets the named download table for this synchronization. Will return NULL if there is no table with the given name in this synchronization.

Parameters

- **table_name** The name of the table for which you want the download data.

Returns

A DownloadTableData instance representing the specified table, or null if a table of the given name does not exist for the current synchronization.

See also

- "DownloadData interface" on page 434
- "DownloadTableData interface" on page 437
- "DBConnectionContext interface" on page 431
- "Direct Row Handling" on page 517

Example

The following example uses the getDownloadTableByName method to return a DownloadTableData instance for the remoteOrders table.

Note
This example assumes you have a DBConnectionContext instance called _cc.

```
// The method used for the handle_DownloadData event.
public void HandleDownload() throws SQLException {
  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.getDownloadData();

  // Get the DownloadTableData for the remoteOrders table.
  DownloadTableData my_download_table = my_dd.getDownloadTableByName
("remoteOrders");

  // ...
}
```

getDownloadTables method

Syntax

public DownloadTableData[ ] **getDownloadTables( )**

Remarks

Gets an array of all the tables for download data in this synchronization. The operations performed on this table will be sent to the remote database.

Returns

An array of DownloadTableData objects for the current synchronization. The order of tables in the array is the same as the upload order of the remote.

See also

- "DownloadData interface" on page 434
- "DownloadTableData interface" on page 437
- "DBConnectionContext interface" on page 431
- "Direct Row Handling" on page 517

Example

The following example uses the DownloadData.getDownloadTables method to obtain an array of DownloadTableData objects for the current synchronization. The example assumes you have a DBConnectionContext instance called _cc.

```
// The method used for the handle_DownloadData event.
public void HandleDownload()
  throws SQLException {
  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.getDownloadData();

  // Get an array of tables to set download operations.
     DownloadTableData[] download_tables = my_dd.getDownloadTables();

  // Get the first table in the DownloadTableData array.
     DownloadTableData my_download_table = download_tables[0];

   // ...
```



```
    }
  }
```

DownloadTableData interface

Encapsulates table operations for MobiLink direct downloads. Use this interface to set the data operations that will be downloaded to the client. To obtain DownloadTableData instances for the current synchronization, use the DownloadData interface. You can use the DownloadTableData.getUpsertPreparedStatement and getDeletePreparedStatement methods to obtain Java prepared statements for insert and update, and delete operations, respectively. The java.sql.PreparedStatement.executeUpdate method registers an operation for download.

> **Note**
> You must set all column values for insert and update prepared statements. For delete operations you set primary key values.
> You cannot have both the delete and upsert prepared statements open at the same time.

Consult your Java SDK documentation for more information about java.sql.PreparedStatement.

See also

- "DownloadData interface" on page 434
- "handle_DownloadData connection event" on page 335
- "Direct Row Handling" on page 517

Example

Assume you use a table called remoteOrders in MobiLink client databases.

```
CREATE TABLE remoteOrders (
     pk  INT NOT NULL,
     col1 varchar(200),
     PRIMARY KEY ( pk )
 );
```

The following example uses the DownloadData.getDownloadTableByName method to return a DownloadTableData instance representing the remoteOrders table.

```
// The method used for the handle_DownloadData event
public void HandleDownload()
  throws SQLException {
  // _cc is a DBConnectionContext instance.

  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.getDownloadData();

  // Get the DownloadTableData for the remoteOrders table.
  DownloadTableData td = my_dd.getDownloadTableByName("remoteOrders");

  // User defined-methods to set download operations.
  setDownloadInserts(td);
  setDownloadDeletes(td);
```



```
  // ...
}
```

In this example, the setDownloadInserts method uses the DownloadTableData.getUpsertPreparedStatement to obtain a prepared statement for rows you want to insert or update. The PreparedStatement.setInt and PreparedStatement.setString methods set the column values you want to insert into the remote database.

```
void setDownloadInserts( DownloadTableData td ) {
     java.sql.PreparedStatement insert_ps = td.getUpsertPreparedStatement();
     // The following method calls are the same as the following SQL
statement:
     // INSERT INTO remoteOrders(pk, col1) values( 2300, "truck" );
     insert_ps.setInt( 1, 2300 );
     insert_ps.setString( 2, "truck" );
     int update_result = insert_ps.executeUpdate();
     if( update_result == 0 ) {
      // Insert was filtered because it was uploaded
      // in the same synchronization.
     } else {
         // Insert was not filtered.
     }
     insert_ps.close();
 }
```

The setDownloadDeletes method uses the DownloadTableData.getDeletePreparedStatement to obtain a prepared statement for rows you want to delete. The java.sql.PreparedStatement.setInt method sets the primary key values for rows you want to delete in the remote database and the java.sql.PreparedStatement.executeUpdate method registers the row values for download.

```
void setDownloadDeletes( DownloadTableData td ) {
  java.sql.PreparedStatement delete_ps = td.getDeletePreparedStatement();
  // The following method calls are the same as the following SQL statement:
  // DELETE FROM remoteOrders  where pk=2300;
  delete_ps.setInt( 1, 2300 );
  delete_ps.executeUpdate();
  delete_ps.close();
}
```

getDeletePreparedStatement method

Syntax

public java.sql.PreparedStatement **getDeletePreparedStatement( )** throws **SQLException**

Remarks

Returns a java.sql.PreparedStatement instance that allows the user to add delete operations to the download. The prepared statement applies to the download table and contains a parameter for each primary key column in the table.

The prepared statement applies to the DownloadTableData instance and contains a parameter for each primary key column in the table.

To include a delete operation in the download, set all columns in your java.sql.PreparedStatement and then call the java.sql.PreparedStatement.executeUpdate method.

Note
You must set all primary key values for download delete operations.

Returns

A java.sql.PreparedStatement instance for adding delete operations to the download.

Exceptions

- **SQLException** Thrown if there is a problem retrieving the delete java.sql.PreparedStatement instance.

See also

- "DownloadTableData interface" on page 437
- "Direct Row Handling" on page 517

Example

In the following example, the setDownloadDeletes method uses the DownloadTableData.getDeletePreparedStatement to obtain a prepared statement for rows you want to delete. The java.sql.PreparedStatement.setInt method sets the primary key values for rows you want to delete in the remote database and the java.sql.PreparedStatement.executeUpdate method sets the row values in the download.

```
void setDownloadDeletes( DownloadTableData td ) {
  java.sql.PreparedStatement delete_ps = td.getDeletePreparedStatement();
  // This is the same as executing the following SQL statement:
  // DELETE FROM remoteOrders  where pk=2300;
  delete_ps.setInt( 1, 2300 );
  delete_ps.executeUpdate();
  delete_ps.close();
 }
```

getUpsertPreparedStatement method

Syntax

public java.sql.PreparedStatement **getUpsertPreparedStatement( )** throws **SQLException**

Remarks

Returns a java.sql.PreparedStatement instance which allows the user to add upsert (insert or update) operations to the download of a synchronization. The prepared statement applies to the DownloadTableData instance and contains a parameter for each column in the table.

To include an insert or update operation in the download, set all column values in your java.sql.PreparedStatement and then call the java.sql.PreparedStatement.executeUpdate method. Calling java.sql.PreparedStatement.executeUpdate on the prepared statement will return 0 if the insert or update operation was filtered and will return 1 if the operation was not filtered. An operation is filtered if it was uploaded in the same synchronization.

> **Note**
> You must set all column values for download insert and update operations.

Returns

A java.sql.PreparedStatement instance for adding upsert operations to the download.

Exceptions

- **SQLException** Thrown if there is a problem retrieving the upsert java.sql.PreparedStatement instance.

See also

- "DownloadTableData interface" on page 437
- "Direct Row Handling" on page 517

Example

In the following example, the setDownloadInserts method uses the DownloadTableData.getUpsertPreparedStatement to obtain a prepared statement for rows you want to insert or update. The java.sql.PreparedStatement.setInt and PreparedStatement.setString methods set the column values, and the PreparedStatement.executeUpdate method sets the row values in the download.

```
void setDownloadInserts( DownloadTableData td ) {
  java.sql.PreparedStatement insert_ps = td.getUpsertPreparedStatement();
  // This is the same as executing the following SQL statement:
  // INSERT INTO remoteOrders(pk, col1) values( 2300, "truck" );
  insert_ps.setInt( 1, 2300 );
  insert_ps.setString( 2, "truck" );
  int update_result = insert_ps.executeUpdate();
  if( update_result == 0 ) {
    // Insert was filtered because it was uploaded
    // in the same synchronization.
  } else {
    // Insert was not filtered.
  }
    insert_ps.close();
 }
```

getName method

Syntax

public java.lang.String **getName( )**

Remarks

Returns the table name for the DownloadTableData instance. You can also access the table name using the java.sql.ResultSetMetaData instance returned by the DownloadTableData.getMetaData method.

Returns

The table name for the DownloadTableData instance.

See also

- "DownloadTableData interface" on page 437
- DownloadTableData "getMetaData method" on page 441
- "Direct Row Handling" on page 517

getMetaData method

Syntax

public java.sql.ResultSetMetaData **getMetaData( )**

Remarks

Gets the metadata for the DownloadTableData instance. The metadata is a standard java.sql.ResultSetMetaData object.

If you want the metadata to contain column name information, specify in your client that column names should be sent with the upload.

Consult your Java SDK documentation for more information about java.sql.ResultSetMetaData.

Returns

The metadata for the DownloadTableData instance.

See also

- "DownloadTableData interface" on page 437
- "Direct Row Handling" on page 517
- SQL Anywhere clients: "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*]
- UltraLite: "Send Column Names synchronization parameter" [*MobiLink - Client Administration*]

InOutByteArray interface

Passed into methods to enable the functionality of an in/out parameter passed to a SQL script.

getValue method

Syntax

public byte[ ] **getValue( )**

Remarks

Returns the value of this byte array parameter.

Returns

The value of this byte array parameter.

setValue method

Syntax

public void **setValue(** byte[ ] *new_value* **)**

Remarks

Sets the value of this byte array parameter.

Parameters

- **new_value** The value for this byte array to take.

InOutInteger interface

Passed into methods to enable the functionality of an in/out parameter passed to a SQL script.

getValue method

Syntax

public int **getValue( )**

Remarks

Returns the value of this integer parameter.

Returns

The value of this integer parameter.

setValue method

Syntax

public void **setValue(** int *new_value* **)**

Remarks

Sets the value of this integer parameter.

Parameters

- **new_value** The value for this integer to take.

InOutString interface

Passed into methods to enable the functionality of an in/out parameter passed to a SQL script.

getValue method

Syntax

public java.lang.String **getValue( )**

Remarks

Returns the value of this string parameter.

Returns

The value of this string parameter.

setValue method

Syntax

public void **setValue(** java.lang.String *new_value* **)**

Remarks

Sets the value of this String parameter.

Parameters

- **new_value** The value for this String to take.

LogListener interface

The listener interface for catching messages that are printed to the log.

See also

- "Handling MobiLink server errors in Java" on page 423

messageLogged method

Syntax

public void **messageLogged(**
ServerContext *sc*,
LogMessage *message* **)**

Remarks

Invoked when a message is printed to the log.

Parameters

- **sc** The context for the server that is printing the message.
- **message** The LogMessage that has been sent to the MobiLink log.

LogMessage class

Holds the data associated with a log message.

Extends java.lang.Object.

See also

- "Handling MobiLink server errors in Java" on page 423

Constants

Syntax

int **ERROR**

Remarks

The log message is an error.

Syntax

int **WARNING**

Remarks

The log message is a warning.

getType method

Syntax

public int **getType( )**

Remarks

Accessor for this message type.

Returns

The type of this message, which can be either LogMessage.ERROR or LogMessage.WARNING.

getUser method

Syntax

public java.lang.String **getUser( )**

Remarks

Accessor for this message user. If the message has no user, then the user is NULL.

Returns

The user associated with this message.

getText method

Syntax

public java.lang.String **getText( )**

Remarks

Accessor for the message text.

Returns

The main text of this message.

ServerContext interface

An instantiation of all the context that is present for the duration of the MobiLink server. This context can be held as static data and used in a background thread. It is valid for the duration of the Java virtual machine invoked by MobiLink.

To access a ServerContext instance, use the DBConnectionContext.getServerContext method.

addShutdownListener method

Syntax

public void **addShutdownListener(** ShutdownListener *sl* **)**

Remarks

Adds the specified ShutdownListener that is to receive notification before the server context is destroyed. On shutdown, the method ShutdownListener.shutdownPerformed (ianywhere.ml.script.ServerContext) is called.

Parameters

- **sl** The ShutdownListener to notify on shutdown.

removeShutdownListener method

Syntax

public void **removeShutdownListener(** ShutdownListener *sl* **)**

Remarks

Removes the specified ShutdownListener from the list of listeners that are to receive notification before the server context is destroyed.

Parameters

- **sl** The ShutdownListener to no longer notify on shutdown.

shutdown method

Syntax

public void **shutdown( )**

Remarks

Forces the server to shut down.

getStartClassInstances method

Syntax

public java.lang.Object[ ] **getStartClassInstances( )**
Gets an array of the start classes that were constructed at server start time. The array length is zero if there are no start classes.

Returns

An array of start classes that were constructed at server start time, or an array of length zero if there are no start classes.

Example

Following is an example that uses getStartClassInstances():

```
Object objs[] = sc.getStartClassInstances();
int i;
for( i=0; i < objs.length; i += 1 ) {
  if( objs[i] instanceof MyClass ) {
    // Use class.
  }
}
```

See also

- "User-defined start classes" on page 424

getProperties method

Syntax

public java.util.Properties **getProperties(**
java.lang.String *component*,
java.lang.String *set* **)**

Remarks

Returns the set of properties associated with a given component and property set. These are stored in the MobiLink system table ml_property.

Parameters

- **component** The component.
- **set** The property set.

Returns

The set of properties, which may be empty.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

getPropertiesByVersion method

Syntax

public java.util.Properties **getPropertiesByVersion(** java.lang.String *script_version* **)**

Remarks

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property. The script version is stored in the prop_set_name column when the component_name is ScriptVersion.

Parameters

- **script_version** The script version for which to return associated properties.

Returns

The set of properties associated with the given script version.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

getPropertySetNames method

Syntax

public java.util.Properties **getPropertySetNames(**
java.lang.String *component_name* **)**

Remarks

Returns the list of property set names for a given component. These are stored in the MobiLink system table ml_property.

Parameters

- **component_name** The name of the component for which to list property names.

Returns

The list of property set names for the given component.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

makeConnection method

Syntax

public java.sql.Connection **makeConnection( )**
throws **java.sql.SQLException**

Remarks

Creates a new server connection. If an error occurs when opening a new connection, the method throws java.sql.SQLException.

Returns

The new server connection.

Exceptions

- **SQLException** Thrown if an error occurred opening the new connection.

addErrorListener method

Syntax

public void **addErrorListener(** LogListener *ll* **)**

Remarks

Adds the specified LogListener to receive a notification when an error is printed.

When an error is printed, the following method is called LogListener.messageLogged (ianywhere.ml.script.ServerContext, ianywhere.ml.script.LogMessage).

Parameters

- **ll** The LogListener to notify.

See also

"messageLogged method" on page 443

removeErrorListener method

Syntax

public void **removeErrorListener(** LogListener *ll* **)**

Remarks

Removes the specified LogListener from the list of listeners that are to receive a notification when an error is printed.

Parameters

- **ll** The LogListener to no longer notify.

addWarningListener method

Syntax

public void **addWarningListener(** LogListener *ll* **)**

Remarks

Adds the specified LogListener to receive a notification when a warning is printed.

The following method will be called: LogListener.messageLogged(ianywhere.ml.script.ServerContext, ianywhere.ml.script.LogMessage).

Parameters

- **ll** The LogListener to notify.

removeWarningListener method

Syntax

public void **removeWarningListener(** LogListener *ll* **)**

Remarks

Removes the specified LogListener from the list of listeners that are to receive a notification when a warning is printed.

Parameters

- **ll** The LogListener to no longer notify.

ServerException class

Thrown to indicate that there is an error condition that makes any further synchronization on the server impossible. Throwing this exception causes the MobiLink server to shut down.

ServerException constructors

Syntax

public **ServerException( )**

Remarks

Constructs a ServerException with no detail message.

Syntax

public **ServerException(** java.lang.String *s* **)**

Remarks

Constructs a ServerException with a specified detail message.

Parameters

- **s** The detailed message.

ShutdownListener interface

The listener interface for catching server shutdowns. Use this interface to ensure that all resources, threads, connections, and so on are cleaned up before the server exits.

shutdownPerformed method

Syntax

public void **shutdownPerformed(** ServerContext *sc***)**

Remarks

Invoked before the ServerContext is destroyed due to server shutdown.

Parameters

- **sc** The context for the server that is being shut down.

SynchronizationException class

Thrown to indicate that there is an error condition that makes the completion of the current synchronization impossible. Throwing this exception will force the MobiLink server to rollback.

SynchronizationException constructors

Syntax

public **SynchronizationException( )**

Remarks

Constructs a SynchronizationException with no detail message.

Syntax

public **SynchronizationException(** java.lang.String *s* **)**

Remarks

Constructs a SynchronizationException with the specified detail message.

Parameters

- **s** A detail message.

UpdateResultSet

A result set object including special methods for accessing the pre-image (old) and post-image (new) values of a specified row. To obtain an UpdateResultSet instance, use the DownloadTableData.getUpdates method.

UpdateResultSet extends java.sql.ResultSet and adds the setNewRowValues and setOldRowValues methods. Otherwise it can be used as a regular resultset. Consult your Java SDK documentation for more information about java.sql.ResultSet

See also

- DownloadTableData "getUpdates method" on page 456
- "Handling conflicts for direct uploads" on page 522
- "Direct Row Handling" on page 517

setNewRowValues method

Syntax

public void **setNewRowValues( )**

Remarks

Sets the mode of this result set to return new column values (the post update row). The result set represents the latest updated values in the remote client database. This is the default mode.

See also

- "UpdateResultSet" on page 451
- "Handling conflicts for direct uploads" on page 522
- "Direct Row Handling" on page 517

setOldRowValues method

Syntax

public void **setOldRowValues( )**

Remarks

Sets the mode of this result set to return old column values (the pre update row). In this mode, the UpdateResultSet represents old column values obtained by the client in the last synchronization.

See also

- "UpdateResultSet" on page 451
- "Handling conflicts for direct uploads" on page 522
- "Direct Row Handling" on page 517

UploadData interface

Encapsulates upload operations for direct row handling. An UploadData instance representing a single upload transaction is passed to the handle_UploadData synchronization event.

> **Caution**
> You must handle direct row handling upload operations in the method registered for the handle_UploadData event. The UploadData is destroyed after each call to the registered method. Do not create a new instance of UploadData to use in subsequent events.

Use the UploadData.getUploadedTables or UploadData.getUploadedTableByName methods to obtain UploadedTableData instances.

A synchronization will have one UploadData unless the remote database is using transactional upload.

See also

- "UploadedTableData interface" on page 454
- "handle_UploadData connection event" on page 347
- "Direct Row Handling" on page 517
- "Handling direct uploads" on page 521

Example

See "handle_UploadData connection event" on page 347.

getUploadedTableByName method

Syntax

public UploadedTableData **getUploadedTableByName(**
  java.lang.String *table_name*
)

Remarks

Returns a UploadedTableData instance representing the specified table.

Parameters

- **table_name** The name of the uploaded table for which you want the uploaded data.

Returns

An UploadedTableData instance representing the specified table, or null if a table of the given name does not exist for the current synchronization.

See also

- "UploadData interface" on page 452
- "UploadedTableData interface" on page 454
- "Direct Row Handling" on page 517

Example

Assume you use a method called HandleUpload for the handle_UploadData synchronization event. The following example uses the getUploadedTableByName method to return an UploadedTableData instance for the remoteOrders table.

```
// The method used for the handle_UploadData event.

public void HandleUpload( UploadData ut )
  throws SQLException, IOException {

     UploadedTableData uploaded_t1 = ut.getUploadedTableByName
("remoteOrders");

  // ...
}
```

getUploadedTables method

Syntax

public UploadedTableData[] **getUploadedTables( )**

Remarks

Returns an array of UploadedTableData objects for the current synchronization. The order to the tables in the array is the same order that MobiLink uses for SQL row handling, and is the optimal order for preventing referential integrity violations. If your data source is a relational database, use this table order.

Returns

An array of UploadedTableData objects for the current synchronization. The order of tables in the array is the same as the upload order of the client.

See also

- "UploadData interface" on page 452
- "UploadedTableData interface" on page 454
- "Direct Row Handling" on page 517

Example

Assume you use a method called HandleUpload for the handle_UploadData synchronization event. The following example uses the getUploadedTables method to return UploadedTableData instances for the current upload transaction.

```
// The method used for the handle_UploadData event.

public void HandleUpload( UploadData ud ) {
  throws SQLException, IOException {

  UploadedTableData tables[] = ud.getUploadedTables();

  //...
}
```

UploadedTableData interface

Encapsulates table operations for direct row handling uploads. You can use an UploadedTableData instance to obtain a table's insert, update, and delete operations for a single upload transaction. Use the UploadedTableData.getInserts, UploadedTableData.getUpdates, and UploadedTableData.getDeletes methods to return standard JDBC java.sql.ResultSet objects.

Consult your Java SDK documentation for more information about java.sql.ResultSet and java.sql.ResultSetMetaData.

Table metadata can be accessed using the UploadedTableData.getMetaData method or the result sets returned by getInserts, getUpdates, and getDeletes. The delete result set only includes primary key columns for a table.

See also

- "UploadData interface" on page 452
- "handle_UploadData connection event" on page 347
- "Direct Row Handling" on page 517

getDeletes method

Syntax

public java.sql.ResultSet **getDeletes( )**

Remarks

Returns a java.sql.ResultSet object representing delete operations uploaded by a MobiLink client. The result set contains primary key values for deleted rows.

Returns

A java.sql.ResultSet object that represents delete operations uploaded by a MobiLink client.

See also

- "UploadedTableData interface" on page 454

- "handle_UploadData connection event" on page 347
- "Direct Row Handling" on page 517

Example

Assume your remote client contains a table called remoteOrders. The following example uses the DownloadTableData.getDeletes method to obtain a result set of deleted rows. In this case, the delete result set includes a single primary key column.

```
import ianywhere.ml.script.*;
import java.sql.*;

// ...


  // The method used for the handle_UploadData event.

  public void HandleUpload( UploadData ut )
    throws SQLException, IOException {
   // Get an UploadedTableData for the remoteOrders table.
      UploadedTableData remoteOrdersTable = ut.getUploadedTableByName
("remoteOrders");

      // Get deletes uploaded by the MobiLink client.
      java.sql.ResultSet delete_rs = remoteOrdersTable.getDeletes();

       while( delete_rs.next() ) {

      // Get primary key values for deleted rows.
         int deleted_id = delete_rs.getInt(1);

      // ...

    }
      delete_rs.close();
}
```

getInserts method

Syntax

public java.sql.ResultSet **getInserts( )**

Remarks

Returns a java.sql.ResultSet object representing insert operations uploaded by a MobiLink client. Each insert is represented by one row in the result set.

Returns

A java.sql.ResultSet object representing insert operations uploaded by a MobiLink client.

See also

- "UploadedTableData interface" on page 454
- "Direct Row Handling" on page 517

Example

Assume your remote client contains a table called remoteOrders. The following example uses the DownloadTableData.getInserts method to obtain a result set of inserted rows. The code obtains the order amount for each row in the current upload transaction.

```
import ianywhere.ml.script.*;
import java.sql.*;

// ...


  // The method used for the handle_UploadData event

  public void HandleUpload( UploadData ut )
    throws SQLException, IOException {
    // Get an UploadedTableData instance representing the remoteOrders table.
       UploadedTableData remoteOrdersTable = ut.getUploadedTableByName
("remoteOrders");

       // Get inserts uploaded by the MobiLink client.
       java.sql.ResultSet rs = remoteOrdersTable.getInserts();

       while( rs.next() ) {

    // get the uploaded order_amount
       double order_amount = rs.getDouble("order_amount");

    // ...

  }
     rs.close();
}
```

getUpdates method

Syntax

public UpdateResultSet **getUpdates( )**

Remarks

Returns a UpdateResultSet object representing update operations uploaded by a MobiLink client. Each update is represented by one row including all column values. UpdateResultSet extends java.sql.ResultSet to include special methods for MobiLink conflict detection.

Returns

An UpdateResultSet object representing update operations uploaded by a MobiLink client.

See also

- “UploadedTableData interface” on page 454
- “UpdateResultSet” on page 451
- “handle_UploadData connection event” on page 347
- “Handling conflicts for direct uploads” on page 522
- “Direct Row Handling” on page 517

Example

Assume your remote client contains a table called remoteOrders. The following example uses the UploadedTableData.getUpdates method to obtain a result set of updated rows. The code obtains the order amount for each row.

```
import ianywhere.ml.script.*;
import java.sql.*;

// ...


  // the method used for the handle_UploadData event

  public void HandleUpload( UploadData ut )
    throws SQLException, IOException {
    // Get an UploadedTableData instance representing the remoteOrders table.
       UploadedTableData remoteOrdersTable = ut.getUploadedTableByName
("remoteOrders");

       // Get inserts uploaded by the MobiLink client.
       java.sql.ResultSet rs = remoteOrdersTable.getUpdates();

       while( rs.next() ) {

    // Get the uploaded order_amount.
       double order_amount = rs.getDouble("order_amount");

    // ...

  }
     rs.close();
}
```

getName method

Syntax

public java.lang.String **getName( )**

Remarks

Returns the table name for the UploadedTableData instance. You can also access the table name using the java.sql.ResultSetMetaData instance returned by the getMetaData method.

Returns

The table name for the UploadedTableData instance.

See also

- "UploadedTableData interface" on page 454
- UploadedTableData "getMetaData method" on page 458
- "handle_UploadData connection event" on page 347
- "Direct Row Handling" on page 517

Example

The following example obtains the name of each uploaded table in a single upload transaction.


```
import ianywhere.ml.script.*;
import java.sql.*;

// ...

// The method used for the handle_UploadData event.

public void HandleUpload( UploadData ud ) {
  throws SQLException, IOException {
     int i;

  // Get UploadedTableData instances.
  UploadedTableData tables[] = ud.getUploadedTables();

     for( i=0; i<tables.length; i+=1 ) {

    // Get the table name.
    String table_name = tables[i].getName();

    // ...
  }
}
```

getMetaData method

Syntax

public java.sql.ResultSetMetaData **getMetaData( )**

Remarks

Gets the metadata for the UploadedTableData instance. The metadata is a standard java.sql.ResultSetMetaData instance.

If you want the ResultSetMetaData to contain column name information, you must specify the client option to send column names.

Consult your Java SDK documentation for more information about java.sql.ResultSetMetaData.

Returns

The metadata for the UploadedTableData instance.

See also

- dbmlsync: "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*]
- UltraLite: "Send Column Names synchronization parameter" [*MobiLink - Client Administration*]

Example

The following example obtains a java.sql.ResultSetMetaData instance for an uploaded table called remoteOrders. The code uses the ResultSetMetaData.getColumnCount and getColumnLabel methods to compile a list of column names.

```
import ianywhere.ml.script.*;
import java.sql.*;

// ...
```



```
// The method used for the handle_UploadData event.

public void HandleUpload( UploadData ut ) {
  throws SQLException, IOException {
  // Get an UploadedTableData instance representing the remoteOrders table.
     UploadedTableData remoteOrdersTable = ut.getUploadedTableByName
("remoteOrders");

     // get inserts uploaded by the MobiLink client
     java.sql.ResultSet rs = remoteOrdersTable.getInserts();

  // Obtain the result set metadata.
     java.sql.ResultSetMetaData md = rs.getMetaData();
  String columnHeading = "";

  // Compile a list of column names.
  for( int c=1; c <= md.getColumnCount(); c += 1 ) {
       columnHeading += md.getColumnLabel();

    if( c < md.getColumnCount() ) {
          columnHeading += ", ";
      }
     }
  //...
}
```

In this case, a method called HandleUpload handles the handle_UploadData synchronization event.

See also

- “UploadedTableData interface” on page 454
- “Direct Row Handling” on page 517
- “SendColumnNames (scn) extended option” [*MobiLink - Client Administration*]
- “handle_UploadData connection event” on page 347

CHAPTER 12

Writing Synchronization Scripts in .NET

Contents

Introduction to .NET synchronization logic

MobiLink supports Visual Studio .NET programming languages for writing synchronization scripts. To write MobiLink scripts in .NET, you can use any language that lets you create valid .NET assemblies. In particular, the following languages are tested and documented:

- C#
- Visual Basic .NET
- C++

.NET synchronization logic can function just as SQL logic functions: the MobiLink server can make calls to .NET methods on the occurrence of MobiLink events just as it accesses SQL scripts on the occurrence of MobiLink events. A .NET method can return a SQL string to MobiLink.

This section tells you how to set up, develop, and run .NET synchronization logic for C#, Visual Basic .NET, and C++. It includes a sample application and the MobiLink server API for .NET Reference.

See also

- "Tutorial: Using .NET Synchronization Logic" [*MobiLink - Getting Started*]
- "Options for writing synchronization logic" [*MobiLink - Getting Started*]
- "Writing Synchronization Scripts" on page 213

Setting up .NET synchronization logic

When you implement synchronization scripts in .NET, you must tell MobiLink where to find the packages, classes, and methods that are contained in your assemblies.

♦ To implement synchronization scripts in .NET

1. Create your own class or classes. Write a method for each required synchronization event. These methods must be public.

   For more information about methods, see "Methods" on page 467.

   Each class with non-static methods should have a public constructor. The MobiLink server automatically instantiates each class the first time a method in that class is called for a connection.

   See "Constructors" on page 466.

2. Create one or more assemblies. While compiling, reference *iAnywhere.MobiLink.Script.dll*, which contains a repository of MobiLink server API classes to use in your own .NET methods. *iAnywhere.MobiLink.Script.dll* is located in the *Assembly\v1* subdirectory of your SQL Anywhere installation.

   You can compile your class on the command line, or using Visual Studio .NET or another .NET development environment.

   See "MobiLink server API for .NET reference" on page 478.

3. Compile your project.

   For example, compile from Visual Studio .NET as follows:

   a. From the VS.NET Project menu, choose Add Existing Item.

   b. Locate *iAnywhere.MobiLink.Script.dll*.

      From the Open dropdown list, choose Link File.

      Note: For Visual Studio .NET, always use the Link File method. Do not use the Add Reference option to reference *iAnywhere.MobiLink.Script.dll*. The Add Reference option duplicates *iAnywhere.MobiLink.Script.dll* in the same physical directory as your class assembly, creating problems for the MobiLink server.

   c. Use the Build menu to build your assembly.

   You can also compile from the command line, as follows:

   Replace *dll-path* with the path to *iAnywhere.MobiLink.Script.dll*. for example, in C#:

   ```
   csc /out:dll-pathout.dll /target:library /reference:dll-
   pathiAnywhere.MobiLink.Script.dll sync_v1.cs
   ```

4. In the MobiLink system tables in your consolidated database, specify the name of the package, class, and method to call for each synchronization script. No more than one class is permitted per script version.

For example, you can add this information to the MobiLink system tables using the ml_add_dnet_connection_script stored procedure or the ml_add_dnet_table_script stored procedure. The following SQL statement, when run in a SQL Anywhere database, specifies that myNamespace.myClass.myMethod should be run whenever the authenticate_user connection-level event occurs.

```
CALL ml_add_dnet_connection_script(
  'version1',
  'authenicate_user',
  'myNamespace.myClass.myMethod' )
```

Note:
The fully qualified method name is case sensitive.

As a result of this procedure call, the script_language column of the ml_script system table contains the word **dnet**. The script column contains the qualified name of a public .NET method.

See "ml_add_dnet_connection_script" on page 534 and "ml_add_dnet_table_script" on page 535.

You can also add this information using Sybase Central.

See "Adding and deleting scripts" on page 228.

5. Instruct the MobiLink server to load assemblies and start the CLR. You tell MobiLink where to locate these assemblies using options on the mlsrv10 command line. There are two options to choose from:

   - **Use -sl dnet (-MLAutoLoadPath)** This sets the given path to the application base directory and loads all the private assemblies within it. You should use this option in most cases. For example, to load all assemblies located in *dll-path*, enter:

     ```
     mlsrv10 -c "dsn=consolidated1" -sl dnet(-MLAutoLoadPath=dll-path)
     ```

     When you use the -MLAutoLoadPath option you cannot specify a domain when entering the fully qualified method name for the event script.

     See "Loading assemblies" on page 473 and "-sl dnet option" on page 65.

   - **Use -sl dnet (-MLDomConfigFile)** This option requires a configuration file that contains domain and assembly settings. You should use this option when you have shared assemblies, when you don't want to load all the assemblies in a directory, or when for some other reason you need to use a configuration file.

     For more information about loading shared assemblies, see "Loading assemblies" on page 473. For more information about the mlsrv10 option -sl dnet, see "-sl dnet option" on page 65.

     Note:
     You can use the -MLAutoLoadPath option or the -MLDomConfigFile option, but not both.

Writing .NET synchronization logic

To write .NET synchronization logic, you require knowledge of MobiLink events, some knowledge of .NET, and familiarity with the MobiLink server API for .NET.

For a complete description of the API, see "MobiLink server API for .NET reference" on page 478.

.NET synchronization logic can be used to maintain state information, and implement logic around the upload and download events. For example, a begin_synchronization script written in .NET could store the MobiLink user name in a variable. Scripts called later in the synchronization process can access this variable. Also, you can use .NET to access rows in the consolidated database, before or after they are committed.

Using .NET also reduces dependence on the consolidated database. Behavior is affected less by upgrading the consolidated database to a new version or switching to a different database management system.

Direct row handling

You can use MobiLink direct row handling to communicate remote data to any central data source, application, or web service. Direct row handling uses special classes in the MobiLink server APIs for Java or .NET for direct access to synchronized data.

See "Direct Row Handling" on page 517.

Class instances

The MobiLink server instantiates your classes at the database connection level. When an event is reached for which you have written a non-static .NET method, the MobiLink server automatically instantiates the class, if it has not already done so on the present database connection.

See "Constructors" on page 466.

> **Note:**
> All methods directly associated with a connection-level or table-level event for one script version must belong to the same class.

For each database connection, once a class has been instantiated, the class persists until that connection is closed. Thus, the same instance may well be used for multiple consecutive synchronization sessions. Information present in public or private variables will thus persist across synchronizations that occur on the same connection unless explicitly cleared.

You can also use static classes or variables. In this case, the values are available across all connections in the same domain.

The MobiLink server automatically deletes your class instances only when the connection to the consolidated database is closed.

Transactions

The normal rules regarding transactions apply to .NET methods. The start and duration of database transactions is critical to the synchronization process. Transactions must be started and ended only by the MobiLink server. Explicitly committing or rolling back transactions on the synchronization connection within a .NET method violates the integrity of the synchronization process and can cause errors.

These rules apply only to the database connections created by the MobiLink server and, in particular, to SQL statements returned by methods. If your classes create other database connections, you are free to manage them as you please.

SQL-.NET data types

The following table shows SQL data types and the corresponding .NET data types for MobiLink script parameters.

SQL data type	Corresponding .NET data type
VARCHAR	string
CHAR	string
INTEGER	int
BINARY	byte [ ]
TIMESTAMP	DateTime
INOUT INTEGER	ref int
INOUT VARCHAR	ref string
INOUT CHAR	ref string
INOUT BYTEARRAY	ref byte [ ]

Constructors

The constructor of your class take no parameters or take one iAnywhere.MobiLink.Script.DBConnectionContext as parameter. For example:

```
public ExampleClass( iAnywhere.MobiLink.Script.DBConnectionContext cc )
```

or

```
public ExampleClass()
```

The synchronization context passed to you is for the connection through which the MobiLink server is synchronizing the current user.

The DBConnectionContext.GetConnection method returns the same database connection that MobiLink is using to synchronize the present user. You can execute statements on this connection, but you must not commit or roll back the transaction. The MobiLink server manages the transactions.

The MobiLink server uses the constructor that takes a iAnywhere.MobiLink.Script.DBConnectionContext parameter if it exists. If it does not, it uses the void constructor.

See "DBConnectionContext interface" on page 481.

Methods

In general, you implement one method for each synchronization event. These methods must be public. If they are private, the MobiLink server cannot use them and fails to recognize that they exist.

The names of the methods are not important, as long as the names match the names specified in the ml_script table in the consolidated database. In the examples included in the documentation, however, the method names are the same as those of the MobiLink events. This naming convention makes the .NET code easier to read.

The signature of your method should match the signature of the script for that event, except that you can truncate the parameter list if you do not need the values of parameters at the end of the list. You should accept only the parameters you need, because overhead is associated with passing the parameters.

You cannot, however, overload the methods. Only one method prototype per class may appear in the ml_script system table.

Registering methods

After creating a method, you must register it. Registering the method creates a reference to the method in the MobiLink system tables on the consolidated database, so that the method is called when the event occurs. You register methods in the same way that you add synchronization scripts, except instead of adding the entire SQL script to the MobiLink system table, you add only the qualified method name.

See "Adding and deleting scripts" on page 228.

Return values

Methods called for a SQL-based upload or download must return a valid SQL language statement. The return type of these methods must be String. No other return types are allowed.

The return type of all other scripts must either be string or void. No other types are allowed. If the return type is a string and not null, the MobiLink server assumes that the string contains a valid SQL statement and executes this statement in the consolidated database as it would an ordinary SQL-language synchronization script. If a method ordinarily returns a string but does not want to execute a SQL statement against the database upon its return, it can return null.

User-defined start classes

You can define start classes that are loaded automatically when the server is started. The purpose of this feature is to allow you to write .NET code that executes at the time the MobiLink server starts the CLR—

before the first synchronization. This means you can create connections or cache data before the first user synchronization request in the server instance.

You do this with the MLStartClasses option of the mlsrv10 -sl dnet option. For example, the following is part of a mlsrv10 command line. It causes mycl1 and mycl2 to be loaded as start classes.

```
-sl dnet(-MLStartClasses=MyNameSpace.MyClass.mycl1,MyNameSpace.MyClass.mycl2)
```

Classes are loaded in the order in which they are listed. If the same class is listed more than once, more than one instance is created.

All start classes must be public and must have a public constructor that either accepts no arguments or accepts one argument of type MobiLink.Script.ServerContext.

The names of loaded start classes are output to the MobiLink log with the message "Loaded .NET start class: *classname*".

For more information about .NET CLR, see "-sl dnet option" on page 65.

To see the start classes that are constructed at server start time, see "GetStartClassInstances method" on page 498.

Example

Following is a template start class. It starts a daemon thread that processes events and creates a database connection. (Not all start classes will need to create a thread but if a thread is spawned it should be a daemon thread.)

```
using System;
using System.IO;
using System.Threading;
using iAnywhere.MobiLink.Script;

namespace TestScripts {
  public class MyStartClass {
    ServerContext    _sc;
    bool             _exit_loop;
    Thread           _thread;
    OdbcConnection   _conn;

    public MyStartClass( ServerContext sc ) {
      // Perform setup first so that an exception will
      // cause MobiLink startup to fail.
      _sc          = sc;
      // Create connection for use later.
      _conn        = _sc.makeConnection();
      _exit_loop   = false;
      _thread      = new Thread( new ThreadStart( run ) ) ;
      _thread.IsBackground = true;

      _thread.Start();
    }

    public void run() {
      ShutdownCallback callback = new ShutdownCallback( shutdownPerformed );

      _sc.ShutdownListener += callback;
      // run() can't throw exceptions.
      try {
        handlerLoop();
```



```
            _conn.close();
            _conn = null;
          } catch( Exception e ) {
            // Print some error output to the MobiLink log.
            Console.Error.Write( e.ToString() );
            // There is no need to be notified of shutdown.
            _sc.ShutdownListener -= callback;
            // Ask server to shut down so this fatal error will
            // be fixed.
            _sc.Shutdown();
          }
          // Shortly after return, this thread will no longer
          // exist.
          return;
        }

        public void shutdownPerformed(
          ServerContext sc ) {
          // Stop the event handler loop.
          try {
            _exit_loop = true;
            // Wait a maximum of 10 seconds for thread to die.
            _thread.Join( 10*1000 );
          } catch( Exception e ) {
            // Print some error output to the MobiLink log.
            Console.Error.Write( e.ToString() );
          }
        }

        private void handlerLoop() {
          while( !_exit_loop ) {
            // Handle events in this loop.
            Thread.Sleep( 1*1000 );
          }
        }
      }
    }
```

Printing information from .NET

You may choose to add statements to your .NET methods that print information to the MobiLink log using System.Console. Doing so can help you track the progress and behavior of your classes.

Performance tip
Printing information in this manner to the MobiLink log is a useful monitoring tool, but is not recommended in a production scenario.

The same technique can be exploited to log arbitrary synchronization information or collect statistical information on how your scripts are used.

Handling MobiLink server errors with .NET

When scanning the log is not sufficient, you can monitor your applications programmatically. For example, you can send messages of a certain type in an email.

You can write methods that are passed a class representing every error or warning message that is printed to the log. This may help you monitor and audit a MobiLink server.

The following code installs a listener for all error messages and prints the information to a StreamWriter.

```
class TestLogListener {
  public TestLogListener( StreamWriter output_file ) {
    _output_file    = output_file;
  }
  public void errCallback( ServerContext sc, LogMessage lm ) {
    string type;
    string user;
    if( lm.Type==LogMessage.MessageType.ERROR ) {
      type = "ERROR";
    } else if( lm.Type==LogMessage.MessageType.WARNING ) {
      type = "WARNING";
    } else {
      type = "INVALID TYPE!!";
    }
    if( lm.User == null ) {
      user = "null";
    } else {
      user = lm.User;
    }

    _output_file.WriteLine( "Caught msg type=" + type +
      " user=" + user +
      " text=" + lm.Text );
    _output_file.Flush();
  }
  StreamWriter _output_file;
}
```

The following code registers the TestLogListener. Call this code from somewhere that has access to the ServerContext such as a class constructor or synchronization script.

```
// ServerContext serv_context;
TestLogListener etll = new TestLogListener(log_listener_file);
serv_context.ErrorListener += new LogCallback(etll.errCallback);
```

See also

- "LogCallback delegate" on page 496
- ErrorListener and WarningListener in "ServerContext interface" on page 498
- "LogMessage class" on page 497
- "MessageType enumeration" on page 497

Debugging .NET synchronization logic

The following procedure describes one way you can debug your .NET scripts using Microsoft Visual Studio .NET.

♦ To debug .NET scripts

1. Compile your code with debugging information turned on:

- On the csc command line, set the **/debug+** option.

  or

- Use Microsoft Visual Studio .NET settings to set debug output.
  - From the File menu choose Build ► Configuration Manager. From the Active Solution Configuration dropdown list, select Debug.
  - Build your assembly.

2. Close running instances of Visual Studio .NET that contain your source files.

   In the next step, you start a new Visual Studio .NET instance to debug the MobiLink server and your .NET synchronization scripts.

3. Start Visual Studio .NET using a command line option to debug the MobiLink server.

   - At a command prompt, navigate to the *Common7\IDE* subdirectory of your Visual Studio .NET installation.
   - Start devenv (the Visual Studio .NET IDE) using the /debugexe option.

     For example, type the following command line to debug the MobiLink server. Remember to specify mlsrv10 options, including the connection string and the option to load .NET assemblies.

     ```
     devenv /debugexe %sqlany10%\win32\mlsrv10.exe -c ...
     ```

     This causes Visual Studio .NET to start and mlsrv10.exe to appear in the Solution Explorer window.

4. Set up Microsoft Visual Studio for debugging .NET code:

   - In the Visual Studio Solution Explorer window, right-click mlsrv10.exe and choose Properties.
   - Change **Debugger Type** from **Auto** to **Mixed** or **Managed Only**.

     This ensures Visual Studio .NET will only debug your .NET synchronization scripts.

5. Open the associated .NET source files and set break points.

   Note: Open the source files individually in the mlsrv10 solution. Do not open the original solution or project file.

6. Start MobiLink from the Debug menu or using F5.

   If prompted, save *mlsrv10.sln* in an appropriate location.

   If the dialog No Symbolic Information appears, click OK to debug anyway. You are debugging the managed .NET synchronization scripts that MobiLink calls, not the MobiLink server itself.

7. Perform a synchronization that causes the code with a breakpoint to be executed by MobiLink.

.NET synchronization techniques

This section describes techniques you can use to tackle common .NET synchronization tasks.

Uploading or downloading rows

For information about how to upload or download rows via .NET, see "Direct Row Handling" on page 517.

Loading shared assemblies

This section details options to load .NET assemblies and details the process to load shared assemblies.

Loading assemblies

A .NET assembly is a package of types, metadata, and executable code. In .NET applications, all code must be in an assembly. Assembly files have the extension *.dll* or *.exe*.

There are the following types of assemblies:

- **Private assemblies** A private assembly is a file in the file system.
- **Shared assemblies** A shared assembly is an assembly that is installed in the global assembly cache.

Before MobiLink can load a class and call a method of that class, it must locate the assembly that contains the class. MobiLink only needs to locate the assembly that it calls directly. The assembly can then call any other assemblies it needs.

For example, MobiLink calls MyAssembly, and MyAssembly calls UtilityAssembly and NetworkingUtilsAssembly. In this situation, MobiLink only needs to be configured to find MyAssembly.

MobiLink provides the following ways to load assemblies:

- **Use -sl dnet (-MLAutoLoadPath)** This option only works with private assemblies. It sets the path to the application base directory and loads all the assemblies within it. This option is simple to use and sufficient in most cases.

  When you use the -MLAutoLoadPath option you cannot specify a domain when entering the fully qualified method name for the event script.

  When you specify a path and directory with -MLAutoLoadPath, MobiLink does the following:

  - sets this path as the application base path
  - loads all classes in all files ending with *.dll* or *.exe* in the directory that you specified
  - creates one application domain and loads into that domain all user classes that do not have a domain specified

  Assemblies in the global assembly cache cannot be called directly with this option. To call these shared assemblies, use -MLDomConfigFile.

- **Use -sl dnet (-MLDomConfigFile)** This option works with both private and shared assemblies. It requires a configuration file that contains domain and assembly settings. You should use this option when you have shared assemblies, when you don't want to load all the assemblies in the application base path, or when for some other reason you need to use a configuration file.

  With this option, MobiLink reads the settings in the specified domain configuration file. A domain configuration file contains configuration settings for one or more .NET domains. If there is more than

one domain represented in the file, the first one that is specified is used as the default domain. (The default domain is used when scripts do not have a domain specified.)

When loading assemblies, MobiLink tries to load the assembly first as private, and then attempts to load the assembly from the global assembly cache. Private assemblies must be located in the application base directory. Shared assemblies are loaded from the global assembly cache.

With the -MLDomConfigFile option, only assemblies that are specified in the domain configuration file can be called directly from event scripts.

Sample domain configuration file

A sample domain configuration file called *mlDomConfig.xml* is installed with MobiLink. You can write your own file from scratch, or edit the sample to suit your needs. The sample file is located in the SQL Anywhere path, in

MobiLink\setup\dnet\mlDomConfig.xml

Following is the content of the sample domain configuration file *mlDomConfig.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns="iAnywhere.MobiLink.mlDomConfig"
xsi:schemaLocation='iAnywhere.MobiLink.mlDomConfig mlDomConfig.xsd'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' >
   <domain>
      <name>SampleDomain1</name>
      <appBase>C:\scriptsDir</appBase>
      <configFile></configFile>
      <assembly name="Assembly1" />
      <assembly name="Assembly2" />
   </domain>
   <domain>
      <name>SampleDomain2</name>
      <appBase>\Dom2assembly</appBase>
      <configFile>\Dom2assembly\AssemblyRedirects.config</configFile>
      <assembly name="Assembly3" />
      <assembly name="Assembly4" />
   </domain>
</config>
```

Following is an explanation of the contents of *mlDomConfig.xml*:

- **name** is the domain name, used when specifying the domain in an event script. An event script with the format `"DomainName:Namespace.Class.Method"` would require a domain called DomainName be in the domain configuration file.

  You must specify at least one domain name.

- **appBase** is the directory that the domain should use as its application base directory. All private assemblies are loaded by the .NET CLR based on this directory. You must specify appBase.

- **configFile** is the .NET application configuration file that should be used for the domain. This can be left blank. It is usually used to modify the default assembly binding and loading behavior. Refer to your .NET documentation for more information about application configuration files.

- **assembly** is the name of an assembly that MobiLink should load and search when resolving type references in event scripts. You must specify at least one assembly. If an assembly is used in more than one domain, it must be specified as an assembly in each domain. If the assembly is private, it must be in the application base directory for the domain.

For more information about the mlsrv10 option -sl dnet, see "-sl dnet option" on page 65.

.NET synchronization example

This example modifies an existing application to describe how to use .NET synchronization logic to handle the authenticate_user event. It creates a C# script for authenticate_user called *AuthUser.cs*. This script looks up the user's password in a table called user_pwd_table and authenticates the user based on that password.

♦ To create your .NET synchronization script

1. Add the table user_pwd_table to the database. Execute the following SQL statements in Interactive SQL:

```
CREATE TABLE user_pwd_table (
  user_name  varchar(128) PRIMARY KEY NOT NULL,
  pwd        varchar(128)
)
```

2. Add a user and password to the table:

```
INSERT INTO user_pwd_table VALUES( 'user1', 'myPwd' )
```

3. Create a directory for your .NET assembly. For example, *c:\mlexample*.

4. Create a file called *AuthUser.cs* with the following contents:

See "authenticate_user connection event" on page 256.

```
using System;
using iAnywhere.MobiLink.Script;

namespace MLExample {

  public class AuthClass {
    private DBConnection  _conn;


    /// AuthClass constructor.

    public AuthClass( DBConnectionContext cc ) {
      _conn   = cc.GetConnection();
    }


    /// The DoAuthenticate method handles the 'authenticate_user'
    /// event.
    /// Note: This method does not handle password changes for
    /// advanced authorization status codes.


    public void DoAuthenticate(
      ref int authStatus,
      string user,
      string pwd,
      string newPwd ) {
      DBCommand   pwd_command = _conn.CreateCommand();
      pwd_command.CommandText = "select pwd from user_pwd_table"
        + " where user_name = ? ";
      pwd_command.Prepare();

      // add a parameter for the user name
      DBParameter user_param = new DBParameter();
```



```
            user_param.DbType = SQLType.SQL_CHAR;
            // Set the size for SQL_VARCHAR.
            user_param.Size   = (uint)user.Length;
            user_param.Value  = user;
            pwd_command.Parameters.Add( user_param );

           // Fetch the password for this user.
            DBRowReader   rr  = pwd_command.ExecuteReader();
            object[]   pwd_row  = rr.NextRow();
            if( pwd_row == null ) {
              // User is unknown.
              authStatus  = 4000;
            } else {
              if( ((string)pwd_row[0]) == pwd ) {
                // Password matched.
                authStatus = 1000;
              } else {
                // Password did not match.
                authStatus = 4000;
              }
            }
            pwd_command.Close();
            rr.Close();
            return;
          }
        }
      }
```

The MLExample.AuthClass.DoAuthenticate method handles the authenticate_user event. It accepts the user name and password and returns an authorization status code indicating the success or failure of the validation.

5. Compile the file *AuthUser.cs*. You can do this on the command line or in Visual Studio .NET.

   For example, the following command line compiles *AuthUser.cs* and generate an Assembly named *example.dll* in *c:\mlexample*.

   ```
   csc /out:c:\mlexample\example.dll /target:library /reference:C:
   \SQLAnywhere10\Assembly\v1\iAnywhere.MobiLink.Script.dll AuthUser.cs
   ```

6. Register .NET code for the authenticate_user event. The method you need to execute (DoAuthenticate) is in the namespace MLExample and class AuthClass. Execute the following SQL:

   ```
   call ml_add_dnet_connection_script( 'ex_version', 'authenticate_user',
   'MLExample.AuthClass.DoAuthenticate' )
   COMMIT
   ```

7. Run the MobiLink server with the following option. This option causes MobiLink to load all assemblies in *c:\myexample*:

   ```
   -sl dnet ( -MLAutoLoadPath=c:\mlexample )
   ```

Now, when a user synchronizes with the version ex_version, they are authenticated with the password from the table user_pwd_table.

MobiLink server API for .NET reference

This section explains the MobiLink .NET interfaces and classes, and their associated methods, properties, and constructors. To use these classes, reference the assembly *\Assembly\v1 \iAnywhere.MobiLink.Script.dll* in your SQL Anywhere installation directory.

This section focuses on C#, but there are equivalents in Visual Basic.NET and C++.

DBCommand interface

Syntax

interface **DBCommand**
Member of **iAnywhere.MobiLink.Script**

Remarks

Represents a SQL statement or database command. DBCommand can represent an update or query.

Example

For example, the following C# code uses the DBCommand interface to execute two queries:

```
DBCommand stmt = conn.CreateCommand();

stmt.CommandText = "select t1a1, t1a2 from table1 ";

DBRowReader rs = stmt.ExecuteReader();
printResultSet( rs );
rs.Close();

stmt.CommandText = "select t2a1 from table2 ";

rs = stmt.ExecuteReader();
printResultSet( rs );
rs.Close();
stmt.Close();
```

The following C# example uses DBCommand to execute an update with parameters:

```
DBCommand cstmt = conn.CreateCommand();

cstmt.CommandText = "call myProc( ?,?,? )";

cstmt.Prepare();

DBParameter param = new DBParameter();
param.DbType          = SQLType.SQL_CHAR;
param.Value           = "10000";
cstmt.Parameters.Add( param );

param                 = new DBParameter();
param.DbType          = SQLType.SQL_INTEGER;
param.Value           = 20000;
cstmt.Parameters.Add( param );

param                 = new DBParameter();
param.DbType          = SQLType.SQL_DECIMAL;
```



```
param.Precision       = 5;
param.Value           = new Decimal( 30000 );
cstmt.Parameters.Add( param );

// Execute update
DBRowReader rset = cstmt.ExecuteNonQuery();
cstmt.Close();
```

Prepare method

Syntax

void **Prepare( )**

Remarks

Prepares the SQL statement stored in CommandText for execution.

ExecuteNonQuery method

Syntax

int **ExecuteNonQuery( )**

Remarks

Executes a non-query statement. Returns the number of rows in the database affected by the SQL statement.

ExecuteReader method

Syntax

DBRowReader **ExecuteReader( )**

Remarks

Executes a query statement returning the result set. Returns a DBRowReader for retrieving results returned by the SQL statement.

Close method

Syntax

void **Close( )**

Remarks

Closes the current SQL statement or command.

CommandText property

Syntax

string **CommandText**

Remarks

The value is the SQL statement to be executed.

DBParameterCollection Parameters property

Syntax

DBParameterCollection Parameters

Remarks

Gets the iAnywhere.MobiLink.Script.DBParameterCollection for this DBCommand.

DBConnection interface

Syntax

interface **DBConnection**
Member of **iAnywhere.MobiLink.Script**

Remarks

Represents a MobiLink ODBC connection.

This interface allows user-written synchronization logic to access an ODBC connection created by MobiLink.

Commit method

Syntax

void **Commit( )**

Remarks

Commits the current transaction.

Rollback method

Syntax

void **Rollback( )**

Remarks

Rolls back the current transaction.

Close method

Syntax

void **Close( )**

Remarks

Closes the current connection.

CreateCommand method

Syntax

DBCommand **CreateCommand( )**

Remarks

Creates a SQL statement or command on this connection. Returns the newly generated DBCommand.

DBConnectionContext interface

Syntax

interface **DBConnectionContext**
Member of **iAnywhere.MobiLink.Script**

Remarks

Interface for obtaining and accessing information about the current database connection. This is passed to the constructor of classes containing scripts. If context is required for a background thread or beyond the lifetime of a connection, use a ServerContext.

For more information about constructors, see "Constructors" on page 466.

Caution
A DBConnectionContext instance should not be used outside the thread that calls into your .NET code.

GetConnection method

Syntax

iAnywhere.MobiLink.Script.DBConnection GetConnection( )
Member of **iAnywhere.MobiLink.Script.DBConnectionContext**

Remarks

Returns the existing connection to the MobiLink consolidated database. The connection is the same connection that MobiLink uses to execute SQL scripts.

This connection must not be committed, closed or altered in any way that would affect the MobiLink server use of the connection. The connection returned is only valid for the lifetime of the underlying MobiLink connection. Do not use the connection after the end_connection event has been called for the connection.

If a server connection with full access is required, use ServerContext.makeConnection().

GetDownloadData method

Syntax

DownloadData GetDownloadData();

Remarks

Returns a DownloadData instance for the current synchronization. Use the DownloadData instance to create the download for direct row handling.

Returns

A DownloadData instance for the current synchronization.

Example

The following example assumes you have a DBConnectionContext instance called _cc.

```
 // The method used for the handle_DownloadData event.
public void HandleDownload() {
  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.GetDownloadData();

  // Get an array of tables to set download operations.
  DownloadTableData[] download_tables = my_dd.GetDownloadTables();

  // Get the first table in the DownloadTableData array.
  DownloadTableData my_download_table = download_tables[0];

   // ...
}
```

GetServerContext method

Syntax

public **iAnywhere.MobiLink.Script.ServerContext.GetServerContext( )**
Member of **iAnywhere.MobiLink.Script.DBConnectionContext**

Remarks

Returns the ServerContext for this MobiLink server.

GetProperties method

Syntax

NameValueCollection **getProperties( )**

Remarks

Returns the properties for this connection, based on this connection's script version. Properties are stored in the ml_property table.

For more information, see “ml_property” on page 561 and “ml_add_property” on page 539.

GetRemoteID method

Syntax

string **GetRemoteID( )**

Remarks

Returns the remote ID of the database currently synchronizing on this connection. If your remote database is prior to version 10, it returns the MobiLink user name.

Returns

The remote ID.

See also

- "Remote IDs" [*MobiLink - Client Administration*]

GetVersion method

Syntax

string **getVersion( )**

Remarks

Returns the name of the script version.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

DBParameter class

Syntax

class **DBParameter**
Member of **iAnywhere.MobiLink.Script**

Remarks

Represents a bound ODBC parameter.

DBParameter is required to execute commands with parameters. All parameters must be in place before the command is executed.

Example

For example, the following C# code uses DBCommand to execute an update with parameters:

```
DBCommand cstmt = conn.CreateCommand();
  cstmt.CommandText = "call myProc( ?,?,? )";
```



```
cstmt.Prepare();

DBParameter param = new DBParameter();
param.DbType          = SQLType.SQL_CHAR;
param.Value           = "10000";
cstmt.Parameters.Add( param );

param                 = new DBParameter();
param.DbType          = SQLType.SQL_INTEGER;
param.Value           = 20000;
cstmt.Parameters.Add( param );

param                 = new DBParameter();
param.DbType          = SQLType.SQL_DECIMAL;
param.Precision       = 5;
param.Value           = new Decimal( 30000 );
cstmt.Parameters.Add( param );

// Execute update
DBRowReader rset = cstmt.ExecuteNonQuery();
cstmt.Close();
```

DbType property

Syntax

SQLTYPE DbType

Remarks

The value is the SQLType of this parameter.

The default value is SQLType.SQL_TYPE_NULL.

Direction property

Syntax

System.Data.ParameterDirection Direction

Remarks

The value is the Input/Output direction of this parameter.

The default value is ParameterDirection.Input.

IsNullable property

Syntax

bool **IsNullable**

Remarks

The value Indicates whether this parameter can be NULL.

The default value is false.

ParameterName property

Syntax

string **ParameterName**

Remarks

The value is the name of this parameter.

The default value is null.

Precision property

Syntax

uint **Precision**

Remarks

The value is the decimal precision of this parameter. Only used for SQLType.SQL_NUMERIC and SQLType.SQL_DECIMAL parameters.

The default value is 0.

Scale property

Syntax

short **Scale**

Remarks

The value is the resolvable digits of this parameter. Only used for SQLType.SQL_NUMERIC and SQLType.SQL_DECIMAL parameters.

The default value is 0.

Size property

Syntax

uint **Size**

Remarks

The value is the size in bytes of this parameter.

The default value is inferred from DbType.

Value property

Syntax

object **Value**

Remarks

The value is the value of this parameter.

The default value is null.

DBParameterCollection class

Syntax

class **DBParameterCollection**
inherits from **IDataParameterCollection**, **IList**, **ICollection**, **IEnumerable**
Member of **iAnywhere.MobiLink.Script**

Remarks

Collection of DBParameters. When DBCommand creates a DBParameterCollection it is empty and must be filled with appropriate parameters before the DBCommand executes.

DBParameterCollection method

Syntax

DBParameterCollection( )

Remarks

Creates an empty list of DBParameters.

Contains(string parameterName) method

Syntax

bool **Contains(string** *parameterName* **)**

Remarks

Returns true if the collection contains a parameter with the specified name.

Parameters

- **parameterName** The name of the parameter to check for.

IndexOf(string parameterName) method

Syntax

int **IndexOf(string** *parameterName* **)**

Remarks

Returns index of the parameter, or -1 if there is no parameter with the given name.

Parameters

- **parameterName** The name of the parameter to find.

RemoveAt(string parameterName) method

Syntax

void **RemoveAt(string** *parameterName* **)**

Remarks

Removes the parameter with the given name from the collection.

Parameters

- **parameterName** The name of the parameter to remove.

Add(object value) method

Syntax

int **Add(object** *value* **)**

Remarks

Adds the given parameter to the collection.

Parameters

- **value** The iAnywhere.MobiLink.Script.DBParameter instance to add to the collection.

Returns

The index of the added parameter in the collection.

Clear method

Syntax

void **Clear( )**

Remarks

Removes all parameters from the collection.

Contains(object value) method

Syntax

bool **Contains(object** *value* **)**

Remarks

Returns true if this collection contains the given iAnywhere.MobiLink.Script.DBParameter.

Parameters

- **value** The iAnywhere.MobiLink.Script.DBParameter to check for.

IndexOf(object value) method

Syntax

int **IndexOf(object** *value* **)**

Remarks

Returns the index of the given iAnywhere.MobiLink.Script.DBParameter in the collection.

Parameters

- **value** The iAnywhere.MobiLink.Script.DBParameter to find.

Insert(int index, object value) method

Syntax

void **Insert(int** *index*, **object** *value* **)**

Remarks

Inserts the given iAnywhere.MobiLink.Script.DBParameter into the collection at the specified index.

Parameters

- **value** The iAnywhere.MobiLink.Script.DBParameter to insert.
- **index** The index at which to insert the DBParameter.

Remove(object value) method

Syntax

void **Remove(object** *value* **)**

Remarks

Removes the given iAnywhere.MobiLink.Script.DBParameter from the collection.

Parameters

- **value** The iAnywhere.MobiLink.Script.DBParameter to remove.

RemoveAt(int index) method

Syntax

int **RemoveAt(int index)**

Remarks

Removes the iAnywhere.MobiLink.Script.DBParameter at the given index in the collection.

Parameters

- **index** The index of the iAnywhere.MobiLink.Script.DBParameter to remove.

CopyTo(Array array, int index) method

Syntax

void **CopyTo(Array** *array*, **int** *index* **)**

Remarks

Copies the contents of the collection into the given array starting at the specified index.

Parameters

- **array** The array to which to copy the contents of the collection.
- **index** The index in the array at which to begin copying the contents of the collection.

GetEnumerator method

Syntax

IEnumerator **GetEnumerator( )**

Remarks

Returns an enumerator for the collection.

IsFixedSize property

Syntax

bool **IsFixedSize**

Remarks

Returns false.

IsReadOnly property

Syntax

bool **IsReadOnly**

Remarks

Returns false.

Count property

Syntax

int **Count**

Remarks

The number of parameters in the collection.

IsSynchronized property

Syntax

bool **IsSynchronized**

Remarks

Returns false.

SyncRoot property

Syntax

object **SyncRoot**

Remarks

Object that can be used to synchronize the collection.

this[string parameterName] property

Syntax

object **this[string** *parameterName* **]**

Remarks

Gets or sets the iAnywhere.MobiLink.Script.DBParameter with the given name in the collection.

Parameters

- **parameterName** The name of the iAnywhere.MobiLink.Script.DBParameter to get or set.

this[int index] property

Syntax

object **this[int** *index* **]**

Remarks

Gets or sets the iAnywhere.MobiLink.Script.DBParameter at the given index in the collection.

Parameters

- **index** The index of the iAnywhere.MobiLink.Script.DBParameter to get or set.

DBRowReader interface

Syntax

interface **DBRowReader**
Member of **iAnywhere.MobiLink.Script**

Remarks

Represents a set of rows being read from a database. Executing the method DBCommand.executeReader( ) creates a DBRowReader.

The following example is a C# code fragment. It calls a function with the rows in the result set represented by the given DBRowReader.

```
DBCommand stmt = conn.CreateCommand();

stmt.CommandText = "select intCol, strCol from table1 ";

DBRowReader rs = stmt.ExecuteReader();
object[] values = rset.NextRow();

while( values != null ) {
handleRow( (int)values[0], (String)values[1] );
 values = rset.NextRow();
}
rset.Close();
stmt.Close();
```

NextRow method

Syntax

object[ ] **NextRow( )**

Remarks

Retrieves and returns the next row of values in the result set. If there are no more rows in the result set, it returns NULL.

See “SQLType enumeration” on page 502.

Close method

Syntax

void **Close( )**

Remarks

Cleans up resources used by this MLDBRowReader. After Close( ) is called, this MLDBRowReader cannot be used again.

ColumnNames property

Syntax

string[ ] **ColumnNames**

Remarks

Gets the names of all columns in the result set. The value is an array of strings corresponding to the column names in the result set.

ColumnTypes property

Syntax

SQLType[ ] **ColumnTypes**

Remarks

Gets the types of all columns in the result set. The value is an array of SQLTypes corresponding to the column types in the result set.

DownloadData interface

Syntax

interface **DownloadData**

Remarks

Encapsulates download data operations for direct row handling. To obtain a DownloadData instance, use the DBConnectionContext GetDownloadData method. Use the DownloadData.GetDownloadTables and GetDownloadTableByName methods to return DownloadTableData instances.

This download data is available through DBConnectionContext. It is not valid to access the download data before the begin_synchronization event or after the end_download event. It is not valid to access DownloadData in an upload-only synchronization.

See also

- "DownloadTableData interface" on page 494
- "handle_DownloadData connection event" on page 335
- DBConnectionContext "GetDownloadData method" on page 482
- "Direct Row Handling" on page 517

GetDownloadTables method

Syntax

DownloadTableData[ ] **GetDownloadTables( );**

Remarks

Gets an array of all the tables for download data in this synchronization. The operations performed on this table will be sent to the remote database.

Returns

An array of download table data. The order of tables in the array is the same as the upload order for the remote.

Example

The following example uses the DownloadData.GetDownloadTables method to obtain an array of DownloadTableData objects for the current synchronization. The example assumes you have a DBConnectionContext instance called _cc.

```
// The method used for the handle_DownloadData event.
public void HandleDownload() {
  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.GetDownloadData();

  // Get an array of tables to set download operations.
  DownloadTableData[] download_tables = my_dd.GetDownloadTables();

  // Get the first table in the DownloadTableData array.
  DownloadTableData my_download_table = download_tables[0];

   // ...
}
```

GetDownloadTableByName method

Syntax

DownloadTableData GetDownloadTableByName(
string *table-name*);

Remarks

Gets the named download table for this synchronization. Will return NULL if there is no table with the given name in this synchronization.

Returns

Download data for the given table name or NULL if not found.

Parameters

- **table-name** Name of the table for which you want the download data.

DownloadTableData interface

Syntax

interface DownloadTableData

Remarks

Encapsulates information for one download table for a synchronization. Use this interface to set the data operations that will be downloaded to a synchronization client site.

Example

For example, suppose you have the following table:

```
CREATE TABLE remoteOrders (
     pk  INT NOT NULL,
     col1 VARCHAR(200),
     PRIMARY KEY ( pk )
 );
```

The following example uses the DownloadData.GetDownloadTableByName method to return a DownloadTableData instance representing the remoteOrders table.

```
// The method used for the handle_DownloadData event
public void HandleDownload() {
  // _cc is a DBConnectionContext instance.

  // Get the DownloadData for the current synchronization.
  DownloadData my_dd = _cc.GetDownloadData();

  // Get the DownloadTableData for the remoteOrders table.
  DownloadTableData td = my_dd.GetDownloadTableByName("remoteOrders");

  // User defined-methods to set download operations.
  SetDownloadUpserts(td);
  SetDownloadDeletes(td);

  // ...
}
```

In this example, the SetDownloadInserts method uses DownloadTableData.GetUpsertCommand to obtain a command for the rows you want to insert or update. The IDbCommand holds the parameters that you will set to the values you want inserted on the remote database.

```
void SetDownloadInserts( DownloadTableData td ) {
      IDbCommand  upsert_cmd = td.GetUpsertCommand();
      IDataParametersCollection parameters = upsert_cmd.Parameters;
     // The following method calls are the same as the following SQL
statement:
     // INSERT INTO remoteOrders(pk, col1) values( 2300, "truck" );
      parameters[0] = 2300;
      parameters[1] = "truck";

    if( upsert_cmd.ExecuteNonQuery() > 0 ) {
         // Insert was not filtered.
     } else {
         // Insert was filtered because it was uploaded
         // in the same synchronization.
     }
```

```
        upsert_cmd.Close();
    }
```

The SetDownloadDeletes method uses the DownloadTableData.GetDeleteCommand to obtain a command for rows you want to delete.

```
void SetDownloadDeletes( DownloadTableData td ) {
   IDbCommand  delete_cmd = t2_download_dd.GetDeleteCommand();
  // The following method calls are the same as the following SQL statement:
  // DELETE FROM remoteOrders  where pk=2300;
  IDataParameterCollection parameters = delete_cmd.Parameters;
  parameters[0] =  2300;
  delete_cmd.ExecuteNonQuery();
  delete_cmd.Close();
}
```

GetDeleteCommand method

Syntax

IDbCommand **GetDeleteCommand();**

Remarks

Gets a command which allows the user to add delete operations to the download data operations. The command returned has the same number of parameters as primary key columns in this table. For the delete to be included in the download, the column values for the primary key columns must be set and the statement executed with ExecuteNonQuery().

> **Note**
> You must set all primary key values for download delete operations.

Returns

A Command for deletes in the download.

Example

See "DownloadTableData interface" on page 494.

GetName method

Syntax

string **GetName();**

Remarks

Gets the table name of this instance. This is a utility function. The table name can also be accessed via the Schema for this instance.

Returns

Table name of this instance.

GetSchemaTable method

Syntax

DataTable **GetSchemaTable();**

Remarks

Gets a DataTable instance that describes the metadata for this download table.

If you want the DataTable to contain column name information, you must specify the client option to send column names.

Returns

A DataTable that describes the column metadata.

See also

- dbmlsync: "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*]
- UltraLite: "Send Column Names synchronization parameter" [*MobiLink - Client Administration*]

GetUpsertCommand method

Syntax

IDbCommand **GetUpsertCommand();**

Remarks

Gets a command that allows you to add upsert (insert/update) operations to the direct download data operations. The command that is returned has the same number of parameters as columns in this table. The column values for the insert must be set and the statement executed with ExecuteNonQuery() for the insert/update to be included in the download. ExecuteNonQuery() on the command will return 0 if the insert/update operation was filtered and will return 1 if the insert/update was not filtered.

You cannot add or remove parameters to this command; you can only set their values.

Returns

A Command for inserts/updates for the download.

Example

See "DownloadTableData interface" on page 494.

LogCallback delegate

Syntax

delegate void **LogCallback(**
  ServerContext *sc*
  LogMessage *message*

)
Member of **iAnywhere.MobiLink.Script**

Remarks

Called when the MobiLink server prints a message.

LogMessage class

Syntax

class **LogMessage** : **iAnywhere.MobiLink.Script.LogMessage**
Member of **iAnywhere.MobiLink.Script**

Remarks

Contains information about a message printed to the log.

Type property

Syntax

LogMessage.MessageType **Type**

Remarks

The type of the log message that this instance represents.

User property

Syntax

string **User**

Remarks

The user for which this message is being logged. It may be null.

Text property

Syntax

string **Text**

Remarks

The main text of the message.

MessageType enumeration

Syntax

enum **MessageType**
Member of **iAnywhere.MobiLink.Script.LogMessage**

Remarks

Enumeration of the possible types of LogMessage.

ERROR field

Syntax

ERROR

Remarks

The log message is an error.

WARNING field

Syntax

WARNING

Remarks

The log message is a warning.

ServerContext interface

Syntax

interface **ServerContext**
Member of **iAnywhere.MobiLink.Script**

Remarks

An instantiation of all the context that is present for the duration of the MobiLink server. This context can be held as static data and used in a background thread. It is valid for the duration of the .NET CLR invoked by MobiLink.

To access a ServerContext instance, use the DBConnectionContext.getServerContext method.

GetStartClassInstances method

Syntax

object[ ] GetStartClassInstances( )
Member of **iAnywhere.MobiLink.Script.ServerContext**

Remarks

Gets an array of the start classes that were constructed at server start time. The array length is zero if there are no start classes.

For more information about user-defined start classes, see "User-defined start classes" on page 467.

Following is an example of getStartClassInstances():

```
void FindStartClass( ServerContext sc, string name )
 {
   object[] startClasses = sc.GetStartClassInstances();
```



```
    foreach( object obj in startClasses ) {
      if( obj is MyClass ) {
          // Execute some code.
      }
    }
  }
```

LogCallback ErrorListener event

This event is triggered when the MobiLink server prints an error.

LogCallback WarningListener event

This event is triggered when the MobiLink server prints a warning.

MakeConnection method

Syntax

iAnywhere.MobiLink.Script.DBConnection makeConnection( )
Member of **iAnywhere.MobiLink.Script.ServerContext**

Remarks

Creates a new server connection.

ShutDown method

Syntax

void **Shutdown( )**
Member of **iAnywhere.MobiLink.Script.ServerContext**

Remarks

Forces the server to shut down.

ShutdownListener method

Syntax

event iAnywhere.MobiLink.Script.ShutdownCallback
ShutdownListener(
iAnwyhere.MobiLink.Script.ServerContext *sc***)**
Member of **iAnywhere.MobiLink.Script.ServerContext**

Remarks

This event is triggered on shutdown. The following code is an example of how to use this event:

```
ShutdownCallback callback = new ShutdownCallback( shutdownHandler );
_sc.ShutdownListener += callback;
```



```
public void shutdownHandler( ServerContext sc )
//============================================
{
_test_out_file.WriteLine( "shutdownPerformed" );
}
```

GetProperties method

Syntax

NameValueCollection **GetProperties(**
string *component_name*
string *prop_set_name* **)**

Remarks

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

GetPropertiesByVersion method

Syntax

NameValueCollection **GetPropertiesByVersion(** string *script_version* **)**

Remarks

Returns the set of properties associated with the script version. These are stored in the MobiLink system table ml_property. The script version is stored in the prop_set_name column when the component_name is ScriptVersion.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

GetPropertySetNames method

Syntax

StringCollection **GetPropertySetNames(** string *component_name* **)**

Remarks

Returns the list of property set names for a given component. These are stored in the MobiLink system table ml_property.

See also

- "ml_property" on page 561
- "ml_add_property" on page 539

ServerException class

Syntax

public class **ServerException**
Member of **iAnywhere.MobiLink.Script**

Remarks

Used to signal MobiLink that an error has occurred with the server and it should shut down immediately.

ServerException constructors

Syntax

public **ServerException( )**
Member of **iAnywhere.MobiLink.Script.ServerException**

Remarks

Constructs a ServerException with no detail message.

Syntax

public **ServerException(string** *message* **)**
Member of **iAnywhere.MobiLink.Script.ServerException**

Remarks

Creates a new ServerException with the given message.

Parameters

- **message** The message for this ServerException.

Syntax

public **ServerException(string** *message*, **SystemException** *ie* **)**
Member of **iAnywhere.MobiLink.Script.ServerException**

Remarks

Creates a new ServerException with the given message and containing the given inner exception that caused this one.

Parameters

- **message** The message for this ServerException.
- **ie** The exception that caused this ServerException.

ShutdownCallback delegate

Syntax

sealed delegate **ShutdownCallback : System.MulticastDelegate**
Member of **iAnywhere.MobiLink.Script**

Remarks

Called when the MobiLink server is shutting down. Implementations of this delegate can be registered with the ServerContext.ShutdownListener event to be called when the MobiLink server shuts down.

SQLType enumeration

Syntax

enum **SQLType**
Member of **iAnywhere.MobiLink.Script**

Remarks

Enumeration of all possible ODBC data types.

SQL_TYPE_NULL field

Syntax

SQL_TYPE_NULL

Remarks

Null data type.

SQL_UNKNOWN_TYPE field

Syntax

SQL_UNKNOWN_TYPE

Remarks

Unknown data type.

SQL_CHAR field

Syntax

SQL_CHAR

Remarks

Single byte string. Has .NET type String.

SQL_NUMERIC field

Syntax

SQL_NUMERIC

Remarks

Numeric value of set size and precision. Has .NET type Decimal.

SQL_DECIMAL field

Syntax

SQL_DECIMAL

Remarks

Decimal number of set size and precision. Has .NET type Decimal.

SQL_INTEGER field

Syntax

SQL_INTEGER

Remarks

32-bit integer. Has .NET type Int32.

SQL_SMALLINT field

Syntax

SQL_SMALLINT

Remarks

16-bit integer. Has .NET type Int16.

SQL_FLOAT field

Syntax

SQL_FLOAT

Remarks

Floating point number with ODBC driver defined precision. Has .NET type Double.

SQL_REAL field

Syntax

SQL_REAL

Remarks

Single precision floating-point number. Has .NET type Single.

SQL_DOUBLE field

Syntax

SQL_DOUBLE

Remarks

Double precision floating point number. Has .NET type Double.

SQL_DATE field

Syntax

SQL_DATE

Remarks

A date. Has .NET type DateTime.

SQL_DATETIME field

Syntax

SQL_DATETIME

Remarks

A date and time. Has .NET type DateTime.

SQL_TIME field

Syntax

SQL_TIME

Remarks

A time. Has .NET type DateTime.

SQL_INTERVAL field

Syntax

SQL_INTERVAL

Remarks

An interval of time. Has .NET type TimeSpan.

SQL_TIMESTAMP field

Syntax

SQL_TIMESTAMP

Remarks

A time stamp. Has .NET type DateTime.

SQL_VARCHAR field

Syntax

SQL_VARCHAR

Remarks

Single byte string. Has .NET type String.

SQL_TYPE_DATE field

Syntax

SQL_TYPE_DATE

Remarks

A date. Has .NET type DateTime.

SQL_TYPE_TIME field

Syntax

SQL_TYPE_TIME

Remarks

A time. Has .NET type DateTime.

SQL_TYPE_TIMESTAMP field

Syntax

SQL_TYPE_TIMESTAMP

Remarks

A timestamp. Has .NET type DateTime.

SQL_DEFAULT field

Syntax

SQL_DEFAULT

Remarks

A default type. Has no type.

SQL_ARD_TYPE field

Syntax

SQL_ARD_TYPE

Remarks

An ARD object. Has no type.

SQL_BIT field

Syntax

SQL_BIT

Remarks

A single bit. Has .NET type Boolean.

SQL_TINYINT field

Syntax

SQL_TINYINT

Remarks

An 8-bit integer. Has .NET type SByte.

SQL_BIGINT field

Syntax

SQL_BIGINT

Remarks

A 64-bit integer. Has .NET type Int64.

SQL_LONGVARBINARY field

Syntax

SQL_LONGVARBINARY

Remarks

Variable length binary data with a driver dependent maximum length. Has .NET type byte[ ].

SQL_VARBINARY field

Syntax

SQL_VARBINARY

Remarks

Variable length binary data with a user specified maximum length. Has .NET type byte[ ].

SQL_BINARY field

Syntax

SQL_BINARY

Remarks

Fixed length binary data. Has .NET type byte[ ].

SQL_LONGVARCHAR field

Syntax

SQL_LONGVARCHAR

Remarks

Single byte string. Has .NET type String.

SQL_GUID field

Syntax

SQL_GUID

Remarks

A Global Unique ID (also called a UUID). Has .NET type Guid.

SQL_WCHAR field

Syntax

SQL_WCHAR

Remarks

Unicode character array of fixed size. Has .NET type String.

SQL_WVARCHAR field

Syntax

SQL_WVARCHAR

Remarks

Null-terminated Unicode string of user-defined maximum length. Has .NET type String.

SQL_WLONGVARCHAR field

Syntax

SQL_WLONGVARCHAR

Remarks

Null-terminated Unicode string of driver-dependent maximum length. Has .NET type String.

SynchronizationException class

Syntax

class **SynchronizationException**
Member of **iAnywhere.MobiLink.Script**

Remarks

Used to signal that a synchronization exception has occurred and that the current synchronization should be rolled back and restarted.

SynchronizationException constructors

Syntax

SynchronizationException( )
Member of **iAnywhere.MobiLink.Script.SynchronizationException**

Remarks

Constructs a SynchronizationException with no details.

Syntax

public **SynchronizationException(string** *message* **)**
Member of **iAnywhere.MobiLink.Script.SynchronizationException**

Remarks

Creates a new SynchronizationException with the given message.

Parameters

- **message** The message for this ServerException.

Syntax

SynchronizationException(string *message*, **SystemException** *ie* **)**
Member of **iAnywhere.MobiLink.Script.SynchronizationException**

Remarks

Creates a new SynchronizationException with the given message and containing the given inner exception that caused this one.

Parameters

- **message** The message for this ServerException.
- **ie** The exception that caused this ServerException.

UploadData interface

Syntax

public interface **UploadData**

Remarks

Encapsulates upload operations for direct row handling. An upload transaction contains a set of tables containing row operations. An UploadData instance representing a single upload transaction is passed to the handle_UploadData synchronization event.

> **Caution**
> You must handle direct row handling upload operations in the method registered for the handle_UploadData event. The UploadData is destroyed after each call to the registered method. Do not create a new instance of UploadData to use in subsequent events.

Use the UploadData.GetUploadedTables or UploadData.GetUploadedTableByName methods to obtain UploadedTableData instances.

A synchronization will have one UploadData unless the remote database is using transactional upload.

See also

- “Direct Row Handling” on page 517
- “UploadedTableData interface” on page 511
- “handle_UploadData connection event” on page 347
- “Handling direct uploads” on page 521

Example

See “handle_UploadData connection event” on page 347.

GetUploadedTableByName method

Syntax

UploadedTableData **GetUploadedTableByName(**
string *table-name*);

Remarks

Gets the named Uploaded table data in this uploaded transaction. Will return null if there is no table in this transaction with the given name.

Parameters

- **table-name** Name of the table for which you want the uploaded data.

Returns

Uploaded data for the given table name or NULL if not found.

Example

Assume you use a method called HandleUpload for the handle_UploadData synchronization event. The following example uses the GetUploadedTableByName method to return an UploadedTableData instance for the remoteOrders table.

```
// The method used for the handle_UploadData event.
public void HandleUpload( UploadData ut ) {
  UploadedTableData uploaded_t1 = ut.GetUploadedTableByName("remoteOrders");
  // ...
}
```

GetUploadedTables method

Syntax

UploadedTableData[ ] **GetUploadedTables();**

Remarks

Gets an array of all the uploaded table data in this uploaded transaction. The order to the tables in the array is the same order that MobiLink uses for SQL row handling, and is the optimal order for preventing referential integrity violations. If your data source is a relational database, use this table order.

Returns

An array of uploaded table data. The order of tables in the array is the same as the upload order of the client.

Example

Assume you use a method called HandleUpload for the handle_UploadData synchronization event. The following example uses the GetUploadedTables method to return UploadedTableData instances for the current upload transaction.

```
// The method used for the handle_UploadData event.
public void HandleUpload( UploadData ud ) {
  UploadedTableData[] tables = ud.GetUploadedTables();
  //...
}
```


UpdateDataReader interface

Syntax

interface **UpdateDataReader**

Remarks

Holds the update operations for one upload transaction for one table. New and old rows can both be accessed by changing the mode of the DataReader to old or new. Otherwise this can be used as a regular DataReader.

SetNewRowValues method

Syntax

void **SetNewRowValues();**

Remarks

Sets the mode of this DataReader to return new column values (the post-update row). This is the default mode.

SetOldRowValues method

Syntax

void **SetOldRowValues();**

Remarks

Sets the mode of this DataReader to return old column values (the pre-update row).

UploadedTableData interface

Syntax

public interface **UploadedTableData**

Remarks

Encapsulates information for one uploaded table for a synchronization.

The insert, update and delete operations are all accessible via the standard ADO.NET IDataReader. The table metadata can be accessed via the GetSchemaTable() call or the insert and delete data readers. The delete data reader will only include the primary key columns of the table.

GetDeletes method

Syntax

IDataReader **GetDeletes();**

Remarks

Gets a DataReader with the deletes for this uploaded table data. Each delete is represented by the primary key values needed to uniquely represent a row in this instances table.

Note: The index and order of the columns will match the array for property DataTable.PrimaryKey for the schema of this table.

Returns

A DataReader with primary key columns for deleted rows.

Example

Assume your remote client contains a table called sparse_pk. The following example uses the DownloadTableData.GetDeletes method to obtain a data reader of deleted rows. In this case, the delete datareader includes two primary key columns. Note the index of each primary key column.

```
CREATE TABLE sparse_pk (
     pcol1  INT NOT NULL,
     col2 VARCHAR(200),
     pcol3 INT NOT NULL,
     PRIMARY KEY ( pcol1, pcol3 )
 );

  using iAnywhere.MobiLink.Script;
  using System;
  using System.IO;
  using System.Data;
  using System.Text;
    ...
  // The method used for the handle_UploadData event.

  public void HandleUpload( UploadData ut ) {

   // Get an UploadedTableData for the sparse_pk table.
   UploadedTableData sparse_pk_table = ut.GetUploadedTableByName
("sparse_pk");

   // Get deletes uploaded by the MobiLink client.
   IDataReader data_reader = sparse_pk_table.GetDeletes();

   while( data_reader.Read() ) {
     StringBuilder row_str = new StringBuilder( "( " );
     row_str.Append( data_reader.GetString( 0 ) ); // pcol1
     row_str.Append( ", " );
     row_str.Append( data_reader.GetString( 1 ) ); // pcol3
     row_str.Append( " )" );
     writer.WriteLine( row_str );
   }
   data_reader.Close();
  }
```

GetInserts method

Syntax

IDataReader **GetInserts();**

Remarks

Gets a DataReader with the inserts for this uploaded table data. Each Insert is represented by one row returned by the reader.

Returns

A DataReader with inserts for this table data.

Example

```
CREATE TABLE sparse_pk (
     pcol1  INT NOT NULL,
     col2 VARCHAR(200),
     pcol3 INT NOT NULL,
     PRIMARY KEY ( pcol1, pcol3 )
);

  using iAnywhere.MobiLink.Script;
  using System;
  using System.IO;
  using System.Data;
  using System.Text;
    ...
  // The method used for the handle_UploadData event.

  public void HandleUpload( UploadData ut ) {

   // Get an UploadedTableData for the sparse_pk table.
   UploadedTableData sparse_pk_table = ut.GetUploadedTableByName
("sparse_pk");

   // Get deletes uploaded by the MobiLink client.
   IDataReader data_reader = sparse_pk_table.GetInserts();

   while( data_reader.Read() ) {
     StringBuilder row_str = new StringBuilder( "( " );
     row_str.Append( data_reader.GetString( 0 ) ); // pcol1
     row_str.Append( ", " );
     if( data_reader.IsDBNull( 1 ) ) {
       row_str.Append( "<NULL>" );
       } else {
       row_str.Append( data_reader.GetString( 1 ) ); // col2
       }
     row_str.Append( ", " );
     row_str.Append( data_reader.GetString( 2 ) ); // pcol3
     row_str.Append( " )" );
     writer.WriteLine( row_str );
}
data_reader.Close();
}
```

GetName method

Syntax

string **GetName();**

Remarks

Gets the table name of this instance. This is a utility function. The table name can also be accessed via the Schema for this instance.

Returns

The table name of this instance.

GetSchemaTable method

Syntax

DataTable **GetSchemaTable();**

Remarks

Gets a DataTable that describes the metadata for this download table.

If you want the DataTable to contain column name information, you must specify the client option to send column names.

Returns

A DataTable that describes the column metadata.

See also

- dbmlsync: "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*]
- UltraLite: "Send Column Names synchronization parameter" [*MobiLink - Client Administration*]

GetUpdates method

Syntax

UpdateDataReader **GetUpdates();**

Remarks

Gets a DataReader with the updates for this uploaded table data. Each row in the result set represent one update. The mode of the result set can be flipped between new and old column values.

Returns

A DataReader with updates for this table data.

Example

The following example illustrates how to use the GetUpdates method.

```
CREATE TABLE sparse_pk (
     pcol1  INT NOT NULL,
     col2 VARCHAR(200),
     pcol3 INT NOT NULL,
     PRIMARY KEY ( pcol1, pcol3 )
 );
```



```
  using iAnywhere.MobiLink.Script;
  using System;
  using System.IO;
  using System.Data;
  using System.Text;
    ...
  // The method used for the handle_UploadData event.

  public void HandleUpload( UploadData ut ) {

   // Get an UploadedTableData for the sparse_pk table.
   UploadedTableData sparse_pk_table = ut.GetUploadedTableByName
("sparse_pk");

   // Get deletes uploaded by the MobiLink client.
   UpdateDataReader data_reader = sparse_pk_table.GetInserts();

   while( data_reader.Read() ) {
     data_reader.SetNewRowValues();
     StringBuilder row_str = new StringBuilder( "New values ( " );
  row_str.Append( data_reader.GetString( 0 ) ); // pcol1
     row_str.Append( ", " );
  if( data_reader.IsDBNull( 1 ) ) {
        row_str.Append( "<NULL>" );
     } else {
        row_str.Append( data_reader.GetString( 1 ) ); // col2
     }
  row_str.Append( ", " );
     row_str.Append( data_reader.GetString( 2 ) ); // pcol3
     row_str.Append( " )" );

     data_reader.SetOldRowValues();
     row_str.Append( " Old Values ( " );
  row_str.Append( data_reader.GetString( 0 ) ); // pcol1
     row_str.Append( ", " );
  if( data_reader.IsDBNull( 1 ) ) {
        row_str.Append( "<NULL>" );
     } else {
        row_str.Append( data_reader.GetString( 1 ) ); // col2
     }
  row_str.Append( ", " );
     row_str.Append( data_reader.GetString( 2 ) ); // pcol3
     row_str.Append( " )" );
  writer.WriteLine( row_str );
  }
  data_reader.Close();
  }
```


CHAPTER 13

Direct Row Handling

Contents

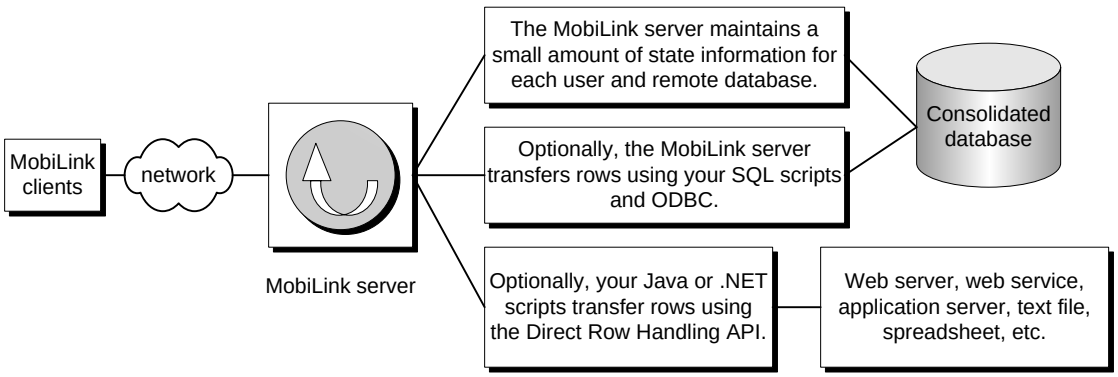
Introduction to direct row handling

MobiLink supports two ways to handle rows: SQL and direct. You can use them separately or together.

- **SQL row handling** allows you to synchronize remote data to a supported consolidated database (SQL Anywhere, Adaptive Server Enterprise, Oracle, Microsoft SQL Server, or IBM DB2). SQL-based events provide a robust interface for conflict resolution and other synchronization tasks. You can use SQL directly or you can return SQL using the MobiLink server APIs for Java and .NET.
- **Direct row handling** allows you to synchronize remote data with any central data source. Direct row handling allows you to access raw synchronized data using special MobiLink events and the MobiLink server APIs for Java and .NET.

  The data sources you can synchronize can be virtually anything, including an application, web server, web service, application server, text file, spreadsheet, non-relational database, or an RDBMS that cannot be used as a consolidated database (such as MySQL). You still need a consolidated database to store your MobiLink system tables, and many implementations of direct row handling will synchronize to both the consolidated database and another data source.

The following diagram shows the basic MobiLink architecture:



The components of direct row handling

To implement direct row handling, you can use two synchronization events along with several interfaces and methods in the MobiLink server APIs for Java and .NET.

Direct synchronization events

Direct row handling allows you to directly access the upload stream and download stream. You do this by writing Java or .NET methods for the handle_UploadData and handle_DownloadData synchronization events.

- **handle_UploadData** accepts a single UploadedData parameter that encapsulates operations uploaded by a MobiLink client for a single upload transaction. See:

- "Handling direct uploads" on page 521
- "handle_UploadData connection event" on page 347

- **handle_DownloadData** allows you to set download operations using the DownloadData interface. See:

  - "Setting direct downloads" on page 527
  - "handle_DownloadData connection event" on page 335

Components of the MobiLink server API for direct row handling

For the Java API:

- DBConnectionContext "getDownloadData method" on page 432
- "DownloadData interface" on page 434
- "DownloadTableData interface" on page 437
- "UpdateResultSet" on page 451
- "UploadData interface" on page 452
- "UploadedTableData interface" on page 454

For the .NET API:

- DBConnectionContext "GetDownloadData method" on page 482
- "DownloadData interface" on page 492
- "DownloadTableData interface" on page 494
- "UpdateDataReader interface" on page 511
- "UploadedTableData interface" on page 511
- "UploadData interface" on page 509

Quick start

To synchronize with a data source other than a consolidated database, complete the following steps.

- **Overview of setting up direct row handling**

1. Set up a consolidated database, if you do not already have one.

   Whether or not you are synchronizing to a consolidated database, you need to have a consolidated database to hold MobiLink system tables.

   See "MobiLink Consolidated Databases" on page 3.

2. If you want to handle uploads, write a public method using the UploadData interface and register it for the handle_UploadData connection event.

   See "Handling direct uploads" on page 521.

3. If you want to handle downloads, write a public method using the DownloadData interface and register it for the handle_DownloadData connection event (or another event).

   See "Setting direct downloads" on page 527.

4. If you want to use the row handling API to refer to columns by name (rather than by index), specify in your client that column names should be sent with the upload. See:

   - SQL Anywhere clients: "SendColumnNames (scn) extended option" [*MobiLink - Client Administration*]
   - UltraLite: "Send Column Names synchronization parameter" [*MobiLink - Client Administration*]

Other resources for getting started

- "Tutorial: Using Direct Row Handling" [*MobiLink - Getting Started*]
- http://ianywhere.codexchange.sybase.com/servlets/ProjectDocumentList?folderID=319
- "Setting up Java synchronization logic" on page 417
- "Setting up .NET synchronization logic" on page 463

Handling direct uploads

To handle direct uploads, you write a method for the handle_UploadData synchronization event. This event accepts one UploadData parameter. See:

- Java server API: "UploadData interface" on page 452
- .NET server API: "UploadData interface" on page 509

The handle_UploadData event is usually called once per synchronization. However, for SQL Anywhere clients that use transaction-level uploads, there can be more than one upload per synchronization, in which case handle_UploadData is called once per transaction.

For more information about dbmlsync transaction-level uploads, see "-tu option" [*MobiLink - Client Administration*].

For more information about using the handle_UploadData event to process the upload, including a detailed example, see "handle_UploadData connection event" on page 347.

For general information about writing Java or .NET synchronization scripts, see:

- "Writing Synchronization Scripts in Java" on page 415
- "Writing Synchronization Scripts in .NET" on page 461

For information about registering connection-level events, see:

- "ml_add_java_connection_script" on page 536
- "ml_add_dnet_connection_script" on page 534

Classes for direct uploads

The MobiLink server APIs for Java and .NET provide the following interfaces for handling direct uploads:

- **UploadData** Encapsulates a single upload transaction. An upload transaction contains a set of tables containing row operations. See:
  - Java API: "UploadData interface" on page 452
  - .NET API: "UploadData interface" on page 509
- **UploadedTableData** Encapsulates a table's insert, update, and delete operations uploaded by a MobiLink client. For Java, UploadedTableData methods return an instance of a JDBC ResultSet. For .NET, UploadedTableData methods return an instance of a standard IDataReader. You traverse the result set IDataReaders to process the uploaded row operations. See:
  - Java API: "UploadedTableData interface" on page 454
  - .NET API: "UploadedTableData interface" on page 511
- **UpdateResultSet** For Java, this class represents an update result set returned by the UploadedTableData getUpdates method. It extends java.sql.ResultSet to include special methods for retrieving the new and old versions of an updated row.

  See "UpdateResultSet" on page 451.

For .NET, the UpdateDataReader interface represents a set of rows returned by the UploadedTableData GetUpdates method. It extends IDataReader to include special methods for retrieving the new and old versions of an updated row.

See "UpdateDataReader interface" on page 511.

Example

See "handle_UploadData connection event" on page 347.

Handling conflicts for direct uploads

When a MobiLink client sends an updated row to the MobiLink server, it includes not only the updated values (the post-image or new row), but also a copy of the old row values (the pre-image or old row) obtained in the last synchronization with the MobiLink server. When the pre-image row does not match the current values in your central data source, a conflict is detected.

SQL-based conflict resolution

For SQL-based uploads, the MobiLink consolidated database is your central data store and MobiLink provides special events for conflict detection and resolution.

See "Handling conflicts" on page 113.

Conflict resolution with direct row handling

For direct uploads, you can access new and old rows programmatically for conflict detection and resolution.

UpdateResultSet (returned by the UploadedTableData.getUpdates method) extends standard Java or .NET result sets to include special methods for handling conflicts. setNewRowValues sets UpdateResultSet to return new updated values from a remote client (the default mode). setOldRowValues sets UpdateResultSet to return old row values.

Example

For example, User1 starts with an inventory of ten items, and then sells three and updates the Remote1 inventory value to seven items. User2 sells four items and updates the Remote2 inventory to six. When Remote1 synchronizes, the central database is updated to seven. When Remote2 synchronizes, a conflict is detected because the value of the inventory is no longer ten. To resolve this conflict programmatically, you need three row values:

- The current value in the consolidated database.
- The new row value that Remote2 uploaded.
- The old row value that Remote2 obtained during the last synchronization.

In this case, the business logic would use the following formula to calculate the new inventory value and resolve the conflict:

```
current consolidated - (old remote - new remote)
-> 7 - (10-6) = 3
```


The following procedures for Java and .NET demonstrate how you can resolve this conflict for direct uploads, using the following table as an example:

```
CREATE TABLE remoteOrders
(
    pk integer primary key not null,
    number_sold integer not null
)
```

♦ To handle direct conflicts (Java)

1. Register a method for the handle_UploadData connection event.

   For example, the following stored procedure call registers a Java method called HandleUpload for the handle_UploadData connection event when synchronizing the script version ver1. You run this stored procedure against your MobiLink consolidated database.

   ```
   call ml_add_java_connection_script( 'ver1',
      'handle_UploadData',
      'OrderProcessor.HandleUpload' )
   ```

   For more information about registering methods for synchronization events, see:

   - "Adding and deleting scripts" on page 228
   - "ml_add_java_connection_script" on page 536

2. Obtain an UpdateResultSet for a table in the upload.

   The OrderProcessor.HandleUpload method obtains an UpdateResultSet for the remoteOrders table:

   ```
   // method for handle_UploadData event
    public void HandleUpload( UploadData u_data )
    {

       // Get UploadedTableData for the remoteOrders table.
         UploadedTableData u_table = u_data.getUploadedTableByName
   ("remoteOrders");

       // Get an UpdateResultSet for the remoteOrders table.
       UpdateResultSet update_rs = u_table.getUpdates();

       // (Continued...)
   ```

3. For each update, get the current values in your central data source.

   In this example, the UpdateResultSet getInt method returns an integer value for the primary key column (the first column). You can implement and then use the getMyCentralData method to get data from your central data source.

   ```
   while( update_rs.next() )
   {
      // Get central data source values.

      // Get the primary key value.
      int pk_value = update_rs.getInt(1);

      // Get central data source values.
      int central_value = getMyCentralData(pk_value);
   ```



```
    // (Continued...)
```

4. For each update, get the old and new values uploaded by the MobiLink client.

   The example uses the UpdateResultSet setOldRowValues and UpdateResultSet setNewRowValues for old and new values, respectively.

```
// Set mode for old row values.
update_rs.setOldRowValues();


// Get an _old_ value.
int old_value = update_rs.getInt(2);


// Set mode for new row values.
update_rs.setNewRowValues();

// Get the _new_ updated value.
int new_value = update_rs.getInt(2);

// (Continued...)
```

5. For each update, check for conflicts.

   A conflict occurs when the old row value does not match the current value in the central data source. To resolve the conflict, a resolved value is calculated using business logic. If no conflict occurs, the central data source is updated with the new remote value. You can implement and then use the setMyCentralData method to perform the update.

```
// Check if there is a conflict.

if(old_value == central_value)
{
  // No conflict.
  setMyCentralData(pk_value, new_value);

}
else
{
  // Handle the conflict.
  int number_sold = old_value - new_value;
  int resolved_value = central_value - number_sold;

  setMyCentralData(pk_value, resolved_value);

  }
}
```

♦ To handle direct conflicts (.NET)

1. Register a method for the handle_UploadData connection event.

   For example, the following stored procedure call registers a .NET method called HandleUpload for the handle_UploadData connection event when synchronizing the script version ver1. You run this stored procedure against your MobiLink consolidated database.

```
call ml_add_dnet_connection_script( 'ver1',
   'handle_UploadData',
   'MyScripts.OrderProcessor.HandleUpload' )
```


For more information about registering methods for synchronization events, see:

- "Adding and deleting scripts" on page 228
- "ml_add_dnet_connection_script" on page 534

2. Obtain an UpdateDataReader for a table in the upload.

   The MyScripts.OrderProcessor.HandleUpload method obtains an UpdateResultSet for the remoteOrders table:

```
// method for handle_UploadData event
 public void HandleUpload( UploadData u_data )
 {

    // Get UploadedTableData for the remoteOrders table.
      UploadedTableData u_table = u_data.GetUploadedTableByName
("remoteOrders");

    // Get an UpdateDataReader for the remoteOrders table.
    UpdateDataReader update_dr = u_table.GetUpdates();

    // (Continued...)
```

3. For each update, get the current values in your central data source.

   In this example, the UpdateDataReader GetInt32 method returns an integer value for the primary key column (the first column). You can implement and then use the getMyCentralData method to get data from your central data source.

```
while( update_dr.Read() )
{
   // Get central data source values.

   // Get the primary key value.
   int pk_value = update_dr.GetInt32(0);

   // Get central data source values.
   int central_value = getMyCentralData(pk_value);

   // (Continued...)
```

4. For each update, get the old and new values uploaded by the MobiLink client.

   The example uses the UpdateResultSet setOldRowValues and UpdateResultSet setNewRowValues for old and new values, respectively.

```
   // Set mode for old row values.
   update_dr.SetOldRowValues();


   // Get an _old_ value.
   int old_value = update_dr.GetInt32(1);


   // Set mode for new row values.
   update_dr.SetNewRowValues();

   // Get the _new_ updated value.
   int new_value = update_dr.GetInt32(1);
```



```
    // (Continued...)
```

5. For each update, check for conflicts.

   A conflict occurs when the old row value does not match the current value in the central data source. To resolve the conflict, a resolved value is calculated using business logic. If no conflict occurs, the central data source is updated with the new remote value. You can implement and then use the setMyCentralData method to perform the update.

```
    // Check if there is a conflict.

    if(old_value == central_value)
    {
      // No conflict.
      setMyCentralData(pk_value, new_value);

    }
    else
    {
      // Handle the conflict.
      int number_sold = old_value - new_value;
      int resolved_value = central_value - number_sold;

      setMyCentralData(pk_value, resolved_value);

    }
  }
```


Setting direct downloads

To create direct downloads, use the DBConnectionContext getDownloadData method for Java or GetDownloadData method for .NET to obtain a DownloadData instance. See:

- Java: "DownloadData interface" on page 434
- .NET: "DownloadData interface" on page 492

You can create the entire direct download in the handle_DownloadData synchronization event. Alternatively, you can use other synchronization events to set direct download operations. However, you must create a handle_DownloadData script, even if its method does nothing. If you process the direct download in an event other than handle_DownloadData, the event cannot be before begin_synchronization and cannot be after end_download.

For information about the order of events, see "MobiLink complete event model" on page 244.

For more information about creating the download, including a detailed example, see "handle_DownloadData connection event" on page 335.

Classes for direct downloads

The MobiLink server APIs for Java and .NET provide the following classes for creating direct downloads:

- **DownloadData** Encapsulates download tables containing operations to send down to a remote client during synchronization. See:
  - Java: "DownloadData interface" on page 434
  - .NET: "DownloadData interface" on page 492
- **DownloadTableData** Encapsulates upsert (update and insert) and delete operations to download to a MobiLink client.

  For Java, DownloadTableData methods return an instance of a JDBC PreparedStatement. In Java, you add a row to the download by setting the prepared statement's column values and then executing the prepared statement.

  For .NET, DownloadTableData methods return an instance of a .NET IDbCommand. In .NET, you add a row to the download by setting the command's column values and then executing the command.

  See:

  - Java: "DownloadTableData interface" on page 437
  - .NET: "DownloadTableData interface" on page 494

Part IV. MobiLink Reference

This part contains MobiLink reference material.

CHAPTER 14

MobiLink Server System Procedures

Contents

MobiLink system procedures

MobiLink provides the following stored procedures to help you create your applications.

System procedures to add or delete scripts

You must add synchronization scripts to system tables in the consolidated database before you can use them. The following system procedures add or delete synchronization scripts in the consolidated database:

- "ml_add_connection_script" on page 533
- "ml_add_table_script" on page 542
- "ml_add_dnet_connection_script" on page 534
- "ml_add_dnet_table_script" on page 535
- "ml_add_java_connection_script" on page 536
- "ml_add_java_table_script" on page 537

When you use the MobiLink server API for Java or .NET, you use these stored procedures to register a method as the script for an event, so that the method is run when the event occurs. You can also use them to unregister your methods.

When you add a script using a system procedure, the script is a string. Any strings within the script need to be escaped. For SQL Anywhere, each quotation mark (') needs to be doubled so as not to terminate the string.

You cannot use system procedures to add scripts longer than 255 bytes to Adaptive Server Enterprise 11.5 or earlier. Instead, use Sybase Central or direct insertion to define longer scripts.

IBM DB2 UDB prior to version 6 only supports column names and other identifiers of 18 characters or less, and so the names are truncated. For example, ml_add_connection_script is shortened to ml_add_connection_.

Other system procedures

- "ml_add_property" on page 539
- "ml_delete_sync_state_before" on page 541
- "ml_reset_sync_state" on page 543

ml_add_column

Registers information about columns on remote databases for use by named column parameters.

Parameters

Item	Description	Remarks
1	script version name	VARCHAR(128)
2	table name	VARCHAR(128)
3	column name	VARCHAR(128)
4	type	Reserved for future use. Set to NULL.

Remarks

This procedure populates the ml_column MobiLink system table with information about the columns on the remote database. The information is used by named row parameters.

You need to run this system procedure if both of the following are true:

- Your SQL scripts contain named parameters for columns (for example, o.column-name and r.column-name).
- You are not using the Create Synchronization Model wizard.

Even if you are using the Create Synchronization Model wizard, if you modify the remote schema outside of Model mode, you need to use this stored procedure to send information about columns that are not registered in ml_column.

To delete all entries for the table name in the given script version, set the column name to NULL.

See also

- “ml_column” on page 554
- “Script parameters” on page 221

Examples

The following stored procedure call populates the ml_column MobiLink system table for col1 in MyTable for the script version Version1. This call allows you to use the named row parameters r.col1 and o.col1 in table scripts for MyTable1 in the Version1 script version.

```
CALL ml_add_column( 'Version1', 'MyTable1', 'col1', NULL )
```

The following stored procedure call deletes all entries in the ml_column MobiLink system table for MyTable1 in script version Version1:

```
CALL ml_add_column( 'Version1', 'MyTable1', NULL, NULL )
```

ml_add_connection_script

Use this system procedure to add or delete SQL connection scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	event name	VARCHAR(128)
3	script contents	For SQL Anywhere and Microsoft SQL Server, this parameter is TEXT. For Adaptive Server Enterprise, this parameter is VARCHAR(16384). For Adaptive Server Enterprise prior to 12.5, this parameter is VARCHAR(255). For DB2 UDB, this parameter is VARCHAR(4000). For Oracle, this parameter is CLOB.

Remarks

To delete a connection script, set the script contents parameter to NULL.

When you add a script, the script is inserted into the ml_script table and the appropriate references are defined to associate the script with the event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "System procedures to add or delete scripts" on page 532
- "Adding and deleting scripts" on page 228
- "ml_add_table_script" on page 542
- "ml_add_dnet_connection_script" on page 534
- "ml_add_dnet_table_script" on page 535
- "ml_add_java_connection_script" on page 536
- "ml_add_java_table_script" on page 537

Example

The following statement adds a connection script associated with the begin_synchronization event to the script version custdb in a SQL Anywhere consolidated database. The script itself is the single statement that sets the @EmployeeID variable.

```
call ml_add_connection_script( 'custdb',
    'begin_synchronization',
    'set @EmployeeID = {ml s.username}' )
```

ml_add_dnet_connection_script

Use this system procedure to register or unregister a .NET method as the script for a connection event.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)

Item	Description	Remarks
2	event name	VARCHAR(128)
3	script contents	For SQL Anywhere and Microsoft SQL Server, this parameter is TEXT. For Adaptive Server Enterprise, this parameter is VARCHAR(16384). For Adaptive Server Enterprise prior to 12.5, this parameter is VARCHAR(255). For DB2 UDB, this parameter is VARCHAR(4000). For Oracle, this parameter is CLOB.

Remarks

To unregister a method, set the script contents parameter to NULL.

The script contents value is a public method in a class in a .NET assembly (for example, MyClass.MyMethod).

When you call ml_add_dnet_connection_script, the method is associated with the event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "System procedures to add or delete scripts" on page 532
- "Adding and deleting scripts" on page 228
- "ml_add_dnet_table_script" on page 535
- "ml_add_connection_script" on page 533
- "ml_add_table_script" on page 542
- "ml_add_java_table_script" on page 537
- "Methods" on page 467
- "Writing Synchronization Scripts in .NET" on page 461

Example

The following example registers the beginDownloadConnection method of the ExampleClass class for the begin_download event.

```
call ml_add_dnet_connection_script( 'ver1',
'begin_download',
'ExamplePackage.ExampleClass.beginDownloadConnection')
```

ml_add_dnet_table_script

Use this system procedure to register or unregister a .NET method as the script for a table event.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	table name	VARCHAR(128)

Item	Description	Remarks
3	event name	VARCHAR(128)
4	script contents	For SQL Anywhere and Microsoft SQL Server, this parameter is TEXT. For Adaptive Server Enterprise, this parameter is VARCHAR(16384). For Adaptive Server Enterprise prior to 12.5, this parameter is VARCHAR(255). For DB2 UDB, this parameter is VARCHAR(4000). For Oracle, this parameter is CLOB.

Remarks

To unregister a method, set the script contents parameter to NULL.

The script value is a public method in a class in a .NET assembly (for example, MyClass.MyMethod).

When you call ml_add_dnet_table_script, the method is associated with the table, event, and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "System procedures to add or delete scripts" on page 532
- "Adding and deleting scripts" on page 228
- "ml_add_dnet_connection_script" on page 534
- "ml_add_connection_script" on page 533
- "ml_add_table_script" on page 542
- "ml_add_java_connection_script" on page 536
- "Methods" on page 467
- "Writing Synchronization Scripts in .NET" on page 461

Example

The following example assigns the empDownloadCursor method of the EgClass class to the download_cursor event for the table emp.

```
call ml_add_dnet_table_script( 'ver1', 'emp',
'download_cursor',EgPackage.EgClass.empDownloadCursor')
```

ml_add_java_connection_script

Use this system procedure to register or unregister a Java method as the script for a connection event.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	event name	VARCHAR(128)

Item	Description	Remarks
3	script contents	For SQL Anywhere and Microsoft SQL Server, this parameter is TEXT. For Adaptive Server Enterprise, this parameter is VARCHAR(16384). For Adaptive Server Enterprise prior to 12.5, this parameter is VARCHAR(255). For DB2 UDB, this parameter is VARCHAR(4000). For Oracle, this parameter is CLOB.

Remarks

To unregister a method, set the script contents parameter to NULL.

The script value is a public method in a class in the MobiLink server classpath (for example, MyClass.MyMethod).

When you ml_add_java_connection_script, the method is associated with the event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "System procedures to add or delete scripts" on page 532
- "Adding and deleting scripts" on page 228
- "ml_add_connection_script" on page 533
- "ml_add_table_script" on page 542
- "ml_add_dnet_connection_script" on page 534
- "ml_add_dnet_table_script" on page 535
- "ml_add_java_table_script" on page 537
- "Methods" on page 421
- "Writing Synchronization Scripts in Java" on page 415

Example

The following example registers the endConnection method of the CustEmpScripts class for the end_connection event.

```
call ml_add_java_connection_script( 'ver1',
'end_connection',
'CustEmpScripts.endConnection')
```

ml_add_java_table_script

Use this system procedure to register or unregister a Java method as the script for a table event.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	table name	VARCHAR(128)

Item	Description	Remarks
3	event name	VARCHAR(128)
4	script content	For SQL Anywhere and Microsoft SQL Server, this parameter is TEXT. For Adaptive Server Enterprise, this parameter is VARCHAR(16384). For Adaptive Server Enterprise prior to 12.5, this parameter is VARCHAR(255). For DB2 UDB, this parameter is VARCHAR(4000). For Oracle, this parameter is CLOB.

Remarks

To unregister a method, set the script content parameter to NULL.

The *script* value is a public method in a class in the MobiLink server classpath (for example, MyClass.MyMethod).

When you call ml_add_java_table_script, the method is associated with the table, event, and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "System procedures to add or delete scripts" on page 532
- "Adding and deleting scripts" on page 228
- "ml_add_connection_script" on page 533
- "ml_add_table_script" on page 542
- "ml_add_dnet_connection_script" on page 534
- "ml_add_dnet_table_script" on page 535
- "ml_add_java_connection_script" on page 536
- "Methods" on page 421
- "Writing Synchronization Scripts in Java" on page 415

Example

The following example registers the empDownloadCursor method of the CustEmpScripts class for the download_cursor event for the table emp.

```
call ml_add_java_table_script( 'ver1', 'emp',
'download_cursor','CustEmpScripts.empDownloadCursor')
```

ml_add_lang_connection_script

This procedure is for internal use only.

ml_add_lang_connection_script_chk

This procedure is for internal use only.

ml_add_property

Use this system procedure to add or delete MobiLink properties. This system procedure changes rows in the ml_property MobiLink system table.

Parameters

Item	Description	Remarks
1	component name	VARCHAR(128)
2	prop_set_name	VARCHAR(128)
3	property name	VARCHAR(128)
4	property value	LONG VARCHAR. For Oracle, this parameter is CLOB.

Remarks

- **component name** You can create records for which this parameter is **ScriptVersion** or **SIS**.

  To save properties by script version, set this parameter to **ScriptVersion**.

  For server-initiated synchronization properties, set this parameter to **SIS**. For more information, see "Setting properties" [*MobiLink - Server-Initiated Synchronization*].

- **prop_set_name** If the component name is **ScriptVersion**, then this parameter is the name of the script version.

  If the component name is **SIS**, then this parameter is the name of the Notifier, gateway, or carrier that you are setting a property for.

- **property name** This parameter is the name of the property.

  If the component name is **ScriptVersion**, then this parameter is a property that you define. You reference these properties using the following methods:

  - from DBConnectionContext: getVersion and getProperties
  - from ServerContext: getPropertiesByVersion, getProperties, and getPropertySetNames

  See "MobiLink server API for Java Reference" on page 431 and "MobiLink server API for .NET reference" on page 478.

  If the component name is **SIS**, then this parameter is a property of the Notifier, gateway, or carrier. See "MobiLink Notification Properties" [*MobiLink - Server-Initiated Synchronization*].

- **property value** This parameter is the value of the property.

  To delete a property, set the property value parameter to NULL.

Server-initiated synchronization

For server-initiated synchronization, the ml_add_property system procedure allows you to set properties for Notifiers, gateways, and carriers.

For example, to add the property `server=mailserver1` for an SMTP gateway called x:

```
ml_add_property( 'SIS','SMTP(x)','server','mailserver1');
```

The verbosity property applies to all Notifiers and gateways, and so you cannot specify a particular prop_set_name. To change the verbosity setting, leave the prop_set_name blank:

```
ml_add_property( 'SIS','','verbosity',2);
```

See "Setting properties" [*MobiLink - Server-Initiated Synchronization*] and "MobiLink Notification Properties" [*MobiLink - Server-Initiated Synchronization*].

Script Version

For regular MobiLink synchronization, you can use this system procedure to associate properties with a script version. In this case, set the component_name to **ScriptVersion**. You can specify whatever properties you like, and use Java and .NET classes to access them.

For example, to associate an LDAP server with a script version called MyVersion:

```
ml_add_property( 'ScriptVersion','MyVersion','ldap-server','MyServer' )
```

See also

- "ml_property" on page 561
- Java API DBConnectionContext: "getProperties method" on page 433 and "getVersion method" on page 434
- .NET API DBConnectionContext: "GetProperties method" on page 482 and "GetVersion method" on page 483
- Java API ServerContext: "getPropertiesByVersion method" on page 447, "getProperties method" on page 446, "getPropertySetNames method" on page 447
- .NET ServerContext: "GetPropertiesByVersion method" on page 500, "GetProperties method" on page 500, "GetPropertySetNames method" on page 500

ml_add_user

This procedure is for internal use only.

ml_delete_sync_state

Use this procedure to delete unused or unwanted synchronization states.

Parameters

Item	Description	Remarks
1	MobiLink user name	VARCHAR(128).
2	remote ID	VARCHAR(128).

Remarks

These parameters can be NULL. If all the parameters are NULL, the procedure does nothing.

This stored procedure deletes all the rows from the ml_subscription table for the given MobiLink user name and remote ID. It also removes this remote ID from the ml_database table, if the remote ID is no longer referenced by any rows in the ml_subscription table.

If the remote ID is NULL and the MobiLink user name is not NULL, it removes all the rows that are referenced by the given MobiLink user name from the ml_subscription table as well as all the remote IDs from the ml_database table, if these remote IDs are no longer referenced by any rows in the ml_subscription table.

If the MobiLink user name is NULL and the remote ID is not NULL, this stored procedure removes all the rows from the ml_subscription table as well as the ml_database table for the given remote ID.

The MobiLink user will not be removed by this stored procedure, even if all the remote IDs have been deleted from the ml_database table and this user is no longer referenced by any rows in the ml_subscription table. If this MobiLink user needs to be deleted, you may delete it by issuing a command such as

```
delete * from ml_user where name = 'user_name'
```

where *user_name* is the MobiLink user you want to delete.

Use this stored procedure with extreme caution because the MobiLink server will automatically add this remote ID in the ml_database and ml_subscription tables without checking its synchronization status the next time the MobiLink client requests synchronization for this remote ID. It may cause data inconsistency to delete synchronization states for a remote ID that did not have a successful synchronization in the last synchronization.

See also

- "ml_subscription" on page 580
- "ml_database" on page 556

Example

The following example cleans up MobiLink system table information about remote databases with the remote ID remote_db_for_John for the MobiLink user John:

```
CALL ml_delete_sync_state( 'John', 'remote_db_for_John' )
```

ml_delete_sync_state_before

Use this procedure to clean up the MobiLink system tables when you have dropped remote databases.

Parameters

Item	Description	Remarks
1	timestamp	TIMESTAMP. The datetime must appear in exactly the order specified in the consolidated database. If the datetime format in the consolidated database is set to 'yyyy/mm/dd hh:mm:ss.ssss', then the timestamp must appear in the order year, month, day, hour, minute, second, fraction of second.

Remarks

This stored procedure removes rows from MobiLink system tables that pertain to remote databases that are no longer being used. In particular, it does the following:

- Deletes all the rows from the ml_subscription system table that have both the last_upload_time and last_download_time earlier than the given timestamp.
- Removes remote IDs from the ml_database system table if the remote IDs are no longer referenced by any rows in the ml_subscription table.

You should not use this system procedure for a time period that is so recent that it may delete rows for remote databases that have not actually been deleted. If you do, the deletion of the rows in ml_subscription and ml_database could cause problems for remote databases that are in an "unknown state" caused by an unsuccessful upload; in that unknown state, the remote relies on the MobiLink system tables to resend data.

The timestamp provided to this procedure must have a correct date-time format because the procedure will not validate the date-time format of the parameter.

See also

- "ml_subscription" on page 580
- "ml_database" on page 556

Example

The following example cleans up MobiLink system table information about remote databases that have not synchronized since January 10, 2004. It works for a SQL Anywhere consolidated database where the date-time format in the consolidated database is yyyy/mm/dd hh:mm:ss.ssss.

```
CALL ml_delete_sync_state_before( '2004/01/10 00:00:00' )
```

ml_delete_user

This procedure is for internal use only.

ml_add_table_script

Function

Use this system procedure to add or delete SQL table scripts in the consolidated database.

Parameters

Item	Description	Remarks
1	version name	VARCHAR(128)
2	table name	VARCHAR(128)
3	event name	VARCHAR(128)
4	script contents	For SQL Anywhere and Microsoft SQL Server, this parameter is TEXT. For ASE, this parameter is VARCHAR (16384). For ASE prior to 12.5, this parameter is VARCHAR(255). For DB2 UDB, this parameter is VARCHAR (4000). For Oracle, this parameter is CLOB.

Remarks

To delete a table script, set the script contents parameter to NULL.

When you add a script, the script is inserted into the ml_script table and the appropriate references are defined to associate the script with the table, event and script version that you specify. If the version name is new, it is automatically inserted into the ml_version table.

See also

- "System procedures to add or delete scripts" on page 532
- "Adding and deleting scripts" on page 228
- "ml_add_connection_script" on page 533
- "ml_add_dnet_connection_script" on page 534
- "ml_add_dnet_table_script" on page 535
- "ml_add_java_connection_script" on page 536
- "ml_add_java_table_script" on page 537

Example

The following command adds a table script associated with the upload_insert event on the Customer table.

```
call ml_add_table_script( 'default', 'Customer', 'upload_insert',
   'INSERT INTO Customer( cust_id, name, rep_id, active )
      VALUES ( {ml r.cust_id}, {ml r.name}, {ml r.rep_id}, 1 )' )
```

ml_reset_sync_state

Use this procedure to reset synchronization state information in MobiLink system tables.

Parameters

Item	Description	Remarks
1	MobiLink user name	VARCHAR(128)
2	remote ID	VARCHAR(128)

The parameters can be NULL. If both parameters are null, this procedure does nothing.

This stored procedure sets the progress, last_upload_time, and last_download_time columns in the ml_subscription table to their default values for the given user_name and remote ID. The default value for the progress is 0. The default value for the last_upload_time and last_download_time columns is '1900/01/01 00:00:00'.

If the remote ID is NULL and the MobiLink user name is not NULL, this procedure sets those columns to the default values for the rows in the ml_subscription table referenced by the given MobiLink user name. If the MobiLink user name is NULL and the remote ID is not NULL, it sets them to the default values for the rows in the ml_subscription table with the given remote ID.

Use this stored procedure with extreme caution. The MobiLink server will not do any synchronization status checking for this remote ID the next time the MobiLink client requests synchronization for this remote ID. It may cause data inconsistency to reset a remote ID that did not have a successful synchronization in the last synchronization.

ml_set_sis_sync_state

This procedure is for internal use only.

CHAPTER 15

MobiLink Utilities

Contents

Introduction to MobiLink utilities

There are two MobiLink server utilities:

- "MobiLink stop utility [mlstop]" on page 547
- "MobiLink user authentication utility [mluser]" on page 548

In addition, see:

- MobiLink client utilities: "MobiLink Client Utilities" [*MobiLink - Client Administration*]
- UltraLite utilities: "UltraLite Utilities Reference" [*UltraLite - Database Management and Reference*]
- Utilities for using TLS certificates: "Certificate utilities" [*SQL Anywhere Server - Database Administration*]
- Other SQL Anywhere utilities: "Database Administration Utilities" [*SQL Anywhere Server - Database Administration*]

MobiLink stop utility [mlstop]

Stops the MobiLink server on the local machine.

Syntax

mlstop [*options*] [*server-name*]

Option	Description
@data	Use this option to read in options from the specified environment variable or configuration file. If both exist with the same name, the environment variable is used. See "Using configuration files" [*SQL Anywhere Server - Database Administration*]. If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file. See "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].
-f	Forced shutdown. Use if a hard shutdown does not work.
-h	Hard shutdown. MobiLink stops all synchronizations and exits. Some remotes may report an error.
-q	Quiet mode. Suppresses the banner.
-t *time*	Soft shutdown, with a hard shutdown done after the specified time. *time* is a number followed by D, H, M, or S (for days, hours, minutes and seconds). For example, `-t 10m` specifies that the server should be shut down in 10 minutes or when current synchronizations complete, whichever is sooner. D, H, M, and S are not case sensitive.
-w	Waits for the MobiLink server to shut down before returning from the command.

Parameters

Server-name If the MobiLink server is started using the -zs option, it must be shut down by specifying the same server name.

See "-zs option" on page 88.

Description

By default (if none of -f, -h or -t are specified), mlstop does a soft shutdown.

- **Soft shutdown** means that the MobiLink server stops accepting new connections and exits when the current synchronizations are complete.
- **Hard shutdown** means that the MobiLink server stops all synchronizations and exits. Some remotes may report an error.

MobiLink user authentication utility [mluser]

Registers MobiLink users at the consolidated database. For SQL Anywhere remotes, the users must have previously been created at the remote databases with the CREATE SYNCHRONIZATION USER statement.

Syntax

mluser [*options*] **-c** "*connection-string*"
{ **-f** *file* | **-u** *user* [**-p** *password*] }

Option	Description
@data	Use this option to read in options from the specified environment variable or configuration file. If both exist with the same name, the environment variable is used. See "Using configuration files" [*SQL Anywhere Server - Database Administration*]. If you want to protect passwords or other information in the configuration file, you can use the File Hiding utility to obfuscate the contents of the configuration file. See "File Hiding utility (dbfhide)" [*SQL Anywhere Server - Database Administration*].
-c "*keyword=value*;…"	Supply database connection parameters. The connection string must give the utility permission to connect to the consolidated database using an ODBC data source. This parameter is required.
-d	Deletes the user name(s) specified by -f or -u.
-dl	Display messages in the window or on the command line and also in the log file, if specified.
-f *filename*	Read the user names and passwords from the specified file. The file should be a text file containing one user name and password pair on each line, separated by white space. You must specify either -f or -u.
-fips	When set, mluser will fail if FIPS support is not installed.
-o *filename*	Log output messages to the specified file.
-os *size*	Limit the size of the output file. The *size* is the maximum file size for logging output messages, specified in bytes. Use the suffix k or m to specify units of kilobytes or megabytes, respectively. By default, there is no size limit. The minimum size limit is 10 KB.
-ot *filename*	Truncate the log file and then append output messages to it. The default is to send output to the screen.
-p *password*	Password to associate with the user. This option can only be used with -u.

Option	Description
-pc *collation-id*	Supply a database collation ID for character set conversion of the user name and password. This should be one of the SQL Anywhere collation labels such as those listed in "Supported and alternate collations" [*SQL Anywhere Server - Database Administration*]. This option is required when user names and passwords are read from a file that is encoded in a different character set than the default character set determined by locale.
-u *ml_username*	Specify the user name to add (or delete, if used with -d). Only one user can be specified on a single command line. This option is used with -p if passwords are being used. You must specify either -f or -u.

Remarks

Given a user/password pair, the mluser utility first attempts to add the user. If the user has already been added to the consolidated database, it attempts to update the password for that user.

There are alternative ways to register user names in the consolidated database:

- Use Sybase Central.
- Specify the -zu+ command line option with mlsrv10. In this case, any existing MobiLink users that have not been added to the consolidated database are added when they first synchronize.

The MobiLink user must already exist in a remote database. To add users at the remote, you have the following options:

- For SQL Anywhere remotes, set the name with CREATE SYNCHRONIZATION USER and synchronize with that user name.
- For UltraLite remotes, you can either use the user_name field of the ul_synch_info structure; or in Java, use the SetUserName() method of the ULSynchInfo class before synchronizing.

See also

- "MobiLink Users" [*MobiLink - Client Administration*]
- "Transport-Layer Security" [*SQL Anywhere Server - Database Administration*]

CHAPTER 16

MobiLink Server System Tables

Contents

Introduction to MobiLink system tables

MobiLink system tables store information about MobiLink users, subscriptions, tables, scripts, script versions, and other information. They are required for MobiLink synchronization. Unlike other system tables, you can modify the MobiLink system tables, although in most cases you will not need to.

MobiLink system tables are created when you run the MobiLink setup script for your consolidated database. They must be stored on your consolidated database. The database user who runs the setup script is the owner of the MobiLink system tables that are created by the script.

See "Setting up a consolidated database" on page 6.

Notes

- This chapter provides data types for the MobiLink system tables in SQL Anywhere consolidated databases. In some RDBMSs, the data types are slightly different.
- IBM DB2 UDB version 5.2 only supports column names and other identifiers of 18 characters or less. In a DB2 UDB 5.2 consolidated database, MobiLink system tables are truncated where necessary.

ml_column

Stores the names of columns for a specific table in a specific script version.

Column	Description
version_id	INTEGER.
table_id	INTEGER.
idx	INTEGER. The index, origin 1, of this column in the table. The column order must be the order the columns were created in the remote database.
name	VARCHAR(128).The column name.
type	VARCHAR(128).Not currently used.

This table has a composite primary key made up of the columns idx, version_id, and table_id.

This table is only required when SQL scripts contain named parameters for columns (for example, o.column-name and r.column-name). (The exception is the column index, which is available even without this MobiLink system table being populated; for example, o.column-index and r.column-index.)

This table is populated by the Create Synchronization Model wizard when you deploy a MobiLink model. If you did not use the Create Synchronization Model wizard, or if you did use it but afterward changed the schema of synchronized columns on the remote database outside of Sybase Central Model mode, you can use the ml_add_column stored procedure to populate the table.

Note: The dbmlsync extended option SendColumnNames and UltraLite synchronization parameter Send Column Names are used by direct row handling, but are not used for named row parameters.

See also

- “ml_add_column” on page 532
- “Script parameters” on page 221

ml_connection_script

For a given script version, this table associates a script with a given event.

Column	Description
version_id	INTEGER. Primary key. This column references the version_id column of the ml_script_version table.
event	VARCHAR(128). Primary key. This column stores the name of the event that triggers the connection script.
script_id	INTEGER. Foreign key. This column references the script_id column of the ml_script system table. The text of the connection script is stored in the ml_script system table.

Remarks

There is a view, ml_connection_scripts, that makes it easier to view the contents of the ml_connection_script MobiLink system table.

ml_database

This table stores a unique ID for each remote database that has synchronized.

Caution
Do not alter this table.

Column	Description
rid	INTEGER. Primary key. This column stores a unique integer identifying the remote ID. This value is used internally.
remote_id	VARCHAR(128). This column stores the MobiLink remote ID. The remote ID uniquely identifies each remote database.
description	VARCHAR(128). Reserved.

The remote ID is sent by the client in each synchronization. The MobiLink server tracks the state information for each remote database using this remote ID.

ml_device

This table is used only for server-initiated synchronization. It stores device names that are required by device tracking.

Column	Description
device_name	VARCHAR(255). Primary key. This column stores the name given to the device. This name is extracted from the operating system unless you specify a name using the dblsn -e option.
listener_version	VARCHAR(128). Not null. This column contains the SQL Anywhere version number for the installed software on the device. Changing this value does not affect the operation of the software, but may be useful for diagnostic purposes.
listener_protocol	INTEGER. Not null. This column is 0, 1, or 2: ♦ 0 - for Listeners from SQL Anywhere prior to version 9.0.1 ♦ 1 - for post-9.0.0 Palm Listeners ♦ 2 - for post-9.0.0 Windows Listeners
info	VARCHAR(255). Not null. This column stores operating system information about the listening device. This information can be overridden by providing information using the dblsn -f option.
ignore_tracking	VARCHAR(1). Not null. If this is **y**, tracking information is not written to the row. If it is **n**, tracking information is written to the row.
source	VARCHAR(255). Not null. This is **tracking** if the row was created by automatic device tracking. Otherwise, it is blank unless you change it using stored procedures to add information about where the data in this row came from. Unless it is set to **tracking**, the value in this column does not affect the operation of the software.

Remarks

The MobiLink system tables ml_device, ml_device_address, and ml_listening contain information about devices for server-initiated synchronization. DeviceTracker gateways use this information to address target devices by MobiLink user name.

In most cases, you should not need to alter these tables. However, if your device does not support device tracking or if you want to override device tracking for troubleshooting purposes, you can add or delete rows to this system table using pre-defined stored procedures. See "Using device tracking with Listeners that don't support it" [*MobiLink - Server-Initiated Synchronization*].

If you want to stop automatic tracking, set ignore_tracking to **y**. In this case, it is also recommended that you use a source name other than **tracking**.

See also

- "ml_set_device" [*MobiLink - Server-Initiated Synchronization*]
- "ml_delete_device" [*MobiLink - Server-Initiated Synchronization*]

ml_device_address

This table is used only for server-initiated synchronization. It stores addressing information that is required by device tracking.

Column	Description
device_name	VARCHAR(255). Primary key. Foreign key references dbo.ml_device. Not null. This column stores the name of the device. This name is extracted from the operating system unless you specify a name using the dblsn -e option.
medium	VARCHAR(255). Primary key. Not null. For UDP, this is _**UDP**_. Otherwise, it is the network provider ID.
address	VARCHAR(255). Not null. For UDP, this is *ip*:*port-number*, where *ip* is an IP address or host name. For SMS, this is the phone number.
active	VARCHAR(1). Not null. This is **y** for active, and **n** otherwise. A DeviceTracker gateway may deactivate a UDP channel if it is unresponsive and there is a fallback SMS delivery path.
last_modified	Timestamp. Not null. Default timestamp. This stores the datetime when this row was last modified.
ignore_tracking	VARCHAR(1). Not null. If this is **y**, tracking information is not written to the row. If it is **n**, tracking information is written to the row.
source	VARCHAR(255). Not null. This is **tracking** if the row was created by automatic device tracking. Otherwise, it is blank unless you change it using stored procedures to add information about where the data in this row came from. Unless it is set to **tracking**, the value in this column does not affect the operation of the software.

Remarks

The MobiLink system tables ml_device, ml_device_address, and ml_listening contain information about devices for server-initiated synchronization. DeviceTracker gateways use this information to address target devices by MobiLink user name.

In most cases, you should not need to alter these tables. However, if your device does not support device tracking or if you want to override device tracking for troubleshooting purposes, you can add or delete rows to this system table using pre-defined stored procedures. See "Using device tracking with Listeners that don't support it" [*MobiLink - Server-Initiated Synchronization*].

If you want to stop automatic tracking, set ignore_tracking to **y**. In this case, it is also recommended that you use a source name other than **tracking**.

See also

- "ml_set_device_address" [*MobiLink - Server-Initiated Synchronization*]
- "ml_delete_device_address" [*MobiLink - Server-Initiated Synchronization*]

ml_listening

This table is used only for server-initiated synchronization. It maps a MobiLink user name to a device name for device tracking.

Column	Description
name	VARCHAR(128). Primary key. Not null. This is the name you use to address notification. This column can be populated in one of three ways: ♦ If you use the dblsn -t+ option, this is the alias that you have defined for the ml_user. ♦ If you used the dblsn -u option, this is an ml_user name. ♦ If you use neither -t+ nor -u, the default name is *device-name*-**dblsn**, where *device-name* is the name of your device. You can find the device name in your Listener console. Optionally, you can set the device name using the dblsn -e option. See "Listener options for device tracking" [*MobiLink - Server-Initiated Synchronization*].
device_name	VARCHAR(255). Foreign key references dbo.ml_device. Not null. This column stores the name given to the device. This name is extracted from the operating system unless you specify a name using the dblsn -e option.
listening	VARCHAR(1). Not null. This is **y** for an active Listener; otherwise it is **n**. This field is set when you use the dblsn option -t, or you can manually set it with stored procedures.
ignore_tracking	VARCHAR(1). Not null. If this is **y**, tracking information is not written to the row. If it is **n**, tracking information is written to the row.
source	VARCHAR(255). Not null. This is **tracking** if the row was created by automatic device tracking. Otherwise, it is blank unless you change it using stored procedures to add information about where the data in this row came from. The value in this column does not affect the operation of the software.

Remarks

The MobiLink system tables ml_device, ml_device_address, and ml_listening contain tracked information about devices for server-initiated synchronization. DeviceTracker gateways use this information to address target devices by MobiLink user name.

In most cases, you should not need to alter these tables. However, if your device does not support device tracking or if you want to override device tracking for troubleshooting purposes, you can add or delete rows in this table using pre-defined stored procedures. See "Using device tracking with Listeners that don't support it" [*MobiLink - Server-Initiated Synchronization*].

If you want to stop automatic tracking, set override_tracking to Yes. In this case, it is also recommended that you use a source name other than **tracking**.

See also

- "ml_set_listening" [*MobiLink - Server-Initiated Synchronization*]

- "ml_delete_listening" [*MobiLink - Server-Initiated Synchronization*]

ml_property

This table stores some MobiLink properties.

Column	Description
component_name	VARCHAR(128). First part of the composite primary key. For user-defined properties, this can be **ScriptVersion** or **SIS**.
property_set_name	VARCHAR(128). Second part of the composite primary key. If the component_name is **ScriptVersion**, this is the name of the script version. If the component_name is **SIS**, this is the name of the Notifier, gateway, or carrier that you are setting a property for.
property_name	VARCHAR(128). Third part of the composite primary key. This is the name of the property. If the component_name is **ScriptVersion**, this is a user-defined property. If the component_name is **SIS**, this is a property of the Notifier, gateway, or carrier. See "MobiLink Notification Properties" [*MobiLink - Server-Initiated Synchronization*].
property_value	TEXT. This is the value of the property.

Remarks

This table stores name-value pairs. Some of the properties in this table are used internally by MobiLink. In addition, you can use the stored procedure ml_add_property to add or delete rows in this table.

You can use the component_name **ScriptVersion** to store information on a per script version basis that can be accessed by Java or .NET scripting logic.

This table has a composite primary key made up of component_name, property_set_name, and property_name.

See also

- "ml_add_property" on page 539

ml_qa_clients

This table is used only for QAnywhere applications. It is a global temporary table that exists only on SQL Anywhere and Oracle consolidated databases.

Caution
Do not alter this table.

Column	Description
client	VARCHAR(128). Client targeted by uploaded messages.

ml_qa_delivery

This table is used only for QAnywhere applications.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Globally unique message identifier.
seqno	BIGINT. Used to give an ordering to the messages, which is necessary for true queuing.
address	VARCHAR(255). Address of the target recipient.
clientaddress	VARCHAR(128). Client part of the address.
client	VARCHAR(128). Client targeted by current client state.
originator	VARCHAR(128). The name of the originating client.
priority	INTEGER. A number from 0 to 9. Messages with a higher priority number are delivered before messages with a lower priority. The default is 4.
expires	TIMESTAMP . Expiry time after which the message might not be delivered.
kind	INTEGER. Indicates whether the message is binary (1) or text (2).
contentsize	BIGINT. The size of the message. For binary messages, this is the number of bytes. For text messages, this is the number of characters.
status	INTEGER. The status of the message. Can be 1 (pending), 10 (receiving), 30 (expired), 40 (cancelled), 50 (unreceivable), or 60 (received).
statustime	TIMESTAMP. The time this status was achieved. The time is local to the client achieving the state.
syncstatus	INTEGER. Indicates the state of the synchronization between the client and server with respect to this message. Can be 0 (not in sync), 1 (in sync), 2 (message should not be synchronized), or 3 (synchronizing).
receiverid	VARCHAR(128). An identifier set by the receiver that identifies the receiver of the message, if any.

ml_qa_delivery_client

This table is used only for QAnywhere applications.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Globally unique message identifier.
seqno	BIGINT. Used to give a total ordering to the messages, this is necessary for true queuing.
address	VARCHAR(255). Address of the target recipient.
target	VARCHAR(128).Client targeted by the current state.
originator	VARCHAR(255). The name of the originating MobiLink user.
priority	INTEGER. A number from 0 to 9. Messages with a higher number are delivered before messages with a lower number. The default is 4.
expires	TIMESTAMP. Expiry time after which the message might not be delivered.
kind	INTEGER. Indicates whether the message is binary (1) or text (2).
contentsize	BIGINT. The size of the message. For binary messages, this is the number of bytes. For text messages, this is the number of characters.
status	INTEGER. The status of the message. Can be 1 (pending), 10 (receiving), 30 (expired), 40 (cancelled), 50 (receivable), or 60 (received).
statustime	TIMESTAMP. The time this status was achieved. The time is local to the client achieving the state.
verbiage	VARCHAR (32767). Localized description of the status (if any).
syncstatus	INTEGER. Indicates the state of the synchronization between the client and server with respect to this message. Can be 0 (not in sync), 1 (in sync), or 2 (message should not be synchronized).
receiverid	VARCHAR(128). An identifier set by the receiver that identifies the receiver of the message, if any.

Remarks

The owner of this table is ml_qa_user_group.

ml_qa_global_props

This table is used only for QAnywhere applications. It contains global *name-value* pairs that are used in transmission rules.

Caution
Do not alter this table.

Column	Description
client	VARCHAR(128). Primary key. The client associated with the property. A client value of 'ianywhere.server.defaultClient' indicates a property that is global to all clients.
name	VARCHAR(255). Primary key. The name of the property.
modifiers	INTEGER. Bitfields used to further describe the property. Currently, only the first bit is used to indicate a property that should not be synchronized. All other bit fields are reserved for future use.
value	LONG VARCHAR. The value of the property.
last_modified	TIMESTAMP. Indicates the last time the value was changed. This is necessary to indicate when a property needs to be synchronized with the client.

Remarks

This table has a composite primary key made up of client and name.

ml_qa_global_props_client

This table is used only for QAnywhere applications. It is synchronized with ml_qa_global_props. This table is created on the remote database by the QAnywhere Agent as needed.

Caution
Do not alter this table.

Column	Description
client	VARCHAR(128). The owner of the property. Values can be **c** (the value is associated with the client owning the message store) or **d** (the value is a default global to all clients). Primary key.
name	VARCHAR(255). Primary key. The name of the property.
modifiers	INTEGER. Bitfields used to further describe the property. Currently, only the first bit is used to indicate a property that should not be synchronized. All other bit fields are reserved for future use.
value	VARCHAR(32767). The value of the property.

Remarks

The owner of this table is ml_qa_user_group.

See also

- "ml_qa_global_props" on page 565

ml_qa_notifications

This table is used only for QAnywhere applications. It is used by the Notifier to determine which QAnywhere clients to notify to initiate synchronization.

Caution
Do not alter this table.

Column	Description
user_id	INTEGER.
name	VARCHAR(128). The QAnywhere client name that uniquely identifies a client message store. Primary key.

ml_qa_repository

This table is used only for QAnywhere applications. It stores messages and their properties.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Primary key. Globally unique message identifier.
props	LONG BINARY. An encoding of the message properties.
content	LONG BINARY. The content of the message. Text messages are encoded as UTF-8.

ml_qa_repository_client

This table is used only for QAnywhere applications. This table is created on the remote database by the QAnywhere Agent as needed.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Primary key. Globally unique message identifier.
props	LONG BINARY. An encoding of the message properties.
content	LONG BINARY. The content of the message. Text messages are encoded as UTF-8.

Remarks

The owner of this table is ml_qa_user_group.

See also

- “ml_qa_repository” on page 568
- “ml_qa_repository_props_client” on page 572

ml_qa_repository_content_client

This table is used only for QAnywhere applications. This table is created on the remote database by the QAnywhere Agent as needed.

Caution
Do not alter this table.

The owner of this table is ml_qa_user_group.

ml_qa_repository_props

This table is used only for QAnywhere applications. This is an expansion of the props column in the ml_qa_repository table. Properties are only expanded as needed by the transmission rules engine. If there are no associated rules, a property is not expanded.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Primary key. Globally unique message identifier.
name	VARCHAR(128). Primary key. The name of the property. If the property name was provided in Unicode, it is translated to the native character set of the database.
value	LONG VARCHAR. The value of the property.

Remarks

This table has a composite primary key made up of msgid and name.

ml_qa_repository_props_client

This table is used only for QAnywhere applications. This table is created on the remote database by the QAnywhere Agent as needed.

Caution
Do not alter this table.

Column	Description
seqno	BIGINT. Used to give a total ordering to the messages, this is necessary for true queuing.
msgid	VARCHAR(128). Primary key. Globally unique message identifier.
name	VARCHAR(128). Primary key. The name of the property. If the property name was provided in Unicode, it is translated to the native character set of the database.
value	VARCHAR(32767). The value of the property.

Remarks

The owner of this table is ml_qa_user_group.

See also

- "ml_qa_repository_props" on page 571

ml_qa_repository_staging

This table is used only for QAnywhere applications. It contains messages that are to be sent to a QAnywhere client using SQL Anywhere version 9.0.1.

Caution
Do not alter this table.

Column	Description
seqno	BIGINT. Used to give a total ordering to the messages, this is necessary for true queuing.
msgid	VARCHAR(255). Primary key. Globally unique message identifier.
destination	VARCHAR(128). The address of the message.
originator	VARCHAR(128). The name of the originating MobiLink user.
status	VARCHAR(128). The status of the message. Can be **pending**, **receiving**, **received**, **unreceivable**, **expired**, or **cancelled**. The default is **pending**.
statustime	TIMESTAMP. The last time the status was changed.
expires	TIMESTAMP. Expiry time after which the message will not be delivered.
priority	INTEGER. A number from 0 to 9. Messages with a higher number will always be delivered before messages with a lower number. The default is 4.
props	LONG BINARY. An encoding of the message properties.
kind	INTEGER. Indicates whether the message is binary (1) or text (2).
content	LONG BINARY. The content of the message. Text messages are encoded as UTF-8.
contentsize	BIGINT. The size of the message. For binary messages, this is the number of bytes. For text messages, this is the number of characters.
mluser	VARCHAR(128). The MobiLink user name that uniquely identifies a remote database.

ml_qa_status_history

This table is used only for QAnywhere applications. It contains a history of message status changes.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Primary key. Globally unique message identifier.
address	VARCHAR(255). Address of the target recipient.
status	INTEGER. The status of the message. Can be 1 (pending), 10 (receiving), 30 (expired), 40 (cancelled), 50 (unreceivable), or 60 (received).
statustime	TIMESTAMP. The time this status was achieved. The time is local to the client achieving the state.
servertime	TIMESTAMP. The time the status change was received by the server.
details	VARCHAR(1000). The details of the status change, if any.
syncstatus	INTEGER. Indicates the state of the synchronization between the client and server with respect to this message. Can be 0 (not in sync), 1 (in sync), 2 (message should not be synchronized), or 3 (synchronizing).

ml_qa_status_staging

This table is used only for QAnywhere applications. It is a staging table that is used when synchronizing status changes with the originating client, where the originating client was using SQL Anywhere version 9.0.1.

Caution
Do not alter this table.

Column	Description
msgid	VARCHAR(128). Primary key. Globally unique message identifier.
status	VARCHAR(255). The status of the message. Can be **pending**, **receiving**, **received**, **unreceivable**, **expired**, or **cancelled**. The default is **pending**.
statustime	TIMESTAMP. The last time the status was changed.
mluser	VARCHAR(128). The MobiLink user name that uniquely identifies a remote database.

ml_script

This table stores the content of all scripts.

Column	Description
script_id	INTEGER. Primary key. This column stores a unique integer that identifies the script.
script	TEXT. The script column stores the text of the script.
script_language	VARCHAR(128). This column stores the scripting language used for the script. The scripting language can be **sql**, **java**, or **dnet**.
checksum	VARCHAR(64). This column is used internally.

ml_script_version

This table stores the name and description of scripts associated with each script version.

Column	Description
version_id	INTEGER. Primary key. This column stores a unique integer that identifies the version.
name	VARCHAR(128). This column stores the name of the script version.
description	TEXT. This column stores the description given to the version. The description is not used by MobiLink, but is useful for application-specific comments. For example, you could describe the purpose of a given script version.

ml_scripts_modified

This table stores the last time script tables were changed. The MobiLink server checks this table to determine if it must load new scripts.

Column	Description
last_modified	DATETIME. Primary key. This column stores the last time when the ml_script, ml_table_script, or ml_connection_script system table was altered.

ml_sis_sync_state

This table is used by Sybase Central to generate request cursors for server-initiated synchronization.

Caution
Do not alter this table.

Column	Description
remote_id	VARCHAR(128). This is the first part of a composite primary key.
subscription_id	VARCHAR(128). This is the second part of a composite primary key.
publication_name	VARCHAR(128).
user_name	VARCHAR(128).
last_upload	TIMESTAMP.
last_download	TIMESTAMP.

ml_subscription

This table stores state information for each remote.

Column	Description
rid	INTEGER. Primary key. This column references the rid column of the ml_database table. This value is the remote ID and it uniquely identifies a database.
subscription_id	VARCHAR(128). Primary key. The subsciption_id is a number that is generated by a remote database. For SQL Anywhere clients, this value is the same as the sync_id in the SYS.ISYSSYNC system table. UltraLite clients do not use subscriptions, so for UltraLite clients, this value is the UltraLite publication ID for version 10.0.0 and up, and <unknown> for versions 8 and 9.
user_id	INTEGER. This column references the user_id column of the ml_user table. This value indicates the user who performed the last synchronization for the given rid and subscription_id. You can use the user_id column to find the MobiLink user who ran the last successful synchronization.
progress	NUMERIC(20,0). This column stores the synchronization progress, also called the offset, state, sequence number, or progress counter.
publication_name	VARCHAR(128). This column stores the user-defined name for the publication that is subscribed to by the subscription. In every synchronization, the client sends the publication name for each subscription_id. For UltraLite clients prior to version 10.0.0, this is always <unknown>; for UltraLite version 10 and higher, it is the publication or the string ul_no_pub if there is no publication.
last_upload_time	TIMESTAMP. This column indicates the last time an upload was applied to the consolidated database for a given remote ID and subscription_id. The default is January 1, 1900, 00:00:00.
last_download_time	TIMESTAMP. This column indicates the last time a download was applied to the consolidated for a given user and subscription_id. The default is January 1, 1900, 00:00:00.

Remarks

In SQL Anywhere clients, the progress refers to a position in the transaction log of the remote database. It indicates the point to which all committed operations for the subscription have been uploaded from the database. The dbmlsync utility uses the offset to decide what data to upload. On the SQL Anywhere remote database, the offset is stored in the progress column of the SYS.ISYSSYNC system table.

See:

- "SYSSYNC system view" [*SQL Anywhere Server - SQL Reference*]
- "Progress offsets" [*MobiLink - Client Administration*]

In UltraLite clients, the progress is the synchronization sequence number or progress counter for the given publication. This counter indicates what rows have been synchronized. It is incremented every time the publication synchronizes. This number is used internally in the UltraLite database and cannot be accessed.

See "The progress counter" [*MobiLink - Client Administration*].

See also

- "Remote IDs" [*MobiLink - Client Administration*]

ml_table

This table stores the names of remote tables. This list includes any table marked as a synchronized table in Sybase Central.

Column	Description
table_id	INTEGER. Primary key. This column stores a unique integer identifying the table.
name	VARCHAR(128). This column stores the name given to the table.

ml_table_script

For a given script version, this table associates a table script with a given table and event.

Column	Description
version_id	INTEGER. Primary key. This column references the version_id column of the ml_script_version table.
table_id	INTEGER. Primary key. This column references the table_id column of the ml_table system table.
event	VARCHAR(128). Primary key. This column stores the name of the event.
script_id	INTEGER. This column references the script_id column of the ml_script table. The script is stored in the ml_script table.

Remarks

There is a view, ml_table_scripts, that makes it easier to view the contents of the ml_table_script MobiLink system table.

ml_user

This table stores all registered MobiLink users and their hashed password.

Column	Description
user_id	INTEGER. Primary key. This column stores a unique integer identifying the user. This value is only used internally by MobiLink.
name	VARCHAR(128). This column stores a registered MobiLink user name.
hashed_password	BINARY(32). This column stores the MobiLink user's password in obfuscated form. If there is no password, this value is NULL. (This is not recommended.)

Remarks

This table stores all registered MobiLink users that are known by the MobiLink server. The user names are sent by MobiLink clients in every synchronization and the clients may optionally send up the password for the user for authentication.

The MobiLink server uses its own algorithm to hash the user password.

Do not directly insert any user names with non-NULL passwords into this table. The MobiLink user names can be added by using the MobiLink user utility, mluser.

See also

- "MobiLink user authentication utility [mluser]" on page 548

CHAPTER 17

MobiLink Data Mappings Between Remote and Consolidated Databases

Contents

Adaptive Server Enterprise data mapping

> **Note**
> Only supported data types are included here.

SQL Anywhere or UltraLite remote data types mapped to Adaptive Server Enterprise consolidated data types

The following table identifies how SQL Anywhere or UltraLite remote data types are mapped to Adaptive Server Enterprise consolidated data types. For example, a column of type FLOAT on the remote database should be type REAL on the consolidated database.

Maximum column length (MCL) depends on the Adaptive Server Enterprise page size. If the page size is 2K the MCL is 1954; if the page size is 4K the MCL is 4002. For information about MCL, see the Adaptive Server Enterprise documentation.

SQL Anywhere or UltraLite data type	Adaptive Server Enterprise data type	Notes
CHAR(n=<MCL)	VARCHAR(n)	
CHAR(n>MCL)	TEXT	On download, ensure the values are not too long.
LONG NVARCHAR	UNITEXT	
LONG VARCHAR	TEXT	
NCHAR(c=<MCL)	UNIVARCHAR(c/2)	
NCHAR(c>MCL)	UNITEXT	On download, ensure the values are not too long.
NTEXT	UNITEXT	
NVARCHAR(c=<MCL)	UNIVARCHAR(c/2)	
NVARCHAR(c>MCL)	UNITEXT	On download, ensure the values are not too long.
TEXT	TEXT	
UNIQUEIDENTIFIERSTR	CHAR(36)	Do not use UNIQUEIDENTIFIERSTR. Use UNIQUEIDENTIFIER instead.
VARCHAR(n=<MCL)	VARCHAR(n)	
VARCHAR(n>MCL)	TEXT	
XML	TEXT	

SQL Anywhere or UltraLite data type	Adaptive Server Enterprise data type	Notes
UNSIGNED BIGINT	NUMERIC(20)[1] or UNSIGNED BIGINT[2]	
BIGINT	NUMERIC(20)[1] or BIGINT[2]	
BIT	BIT	
DECIMAL(p<39, s)	DECIMAL(p,s)	The precision of the Adaptive Server Enterprise NUMERIC can be from 1 to 38 digits (p<39).
DECIMAL(p>=39,s)		There is no corresponding data type in SQL Anywhere or UltraLite.
DOUBLE	DOUBLE PRECISION	
FLOAT(p)	FLOAT(p)	
UNSIGNED INTEGER	UNSIGNED INT	
INTEGER	INTEGER	
NUMERIC(p<39,s)	NUMERIC(p,s)	The precision of the Adaptive Server Enterprise decimal can be from 1 to 38 digits (p<39).
NUMERIC(p>=39,s)		
REAL	REAL	
UNSIGNED SMALLINT	UNSIGNED SMALLINT	
SMALLINT	SMALLINT	
UNSIGNED TINYINT	TINYINT	
TINYINT	TINYINT	
MONEY	MONEY	
SMALLMONEY	SMALLMONEY	
LONG VARBIT	TEXT	
VARBIT(n=<MCL)	VARCHAR(n)	
VARBIT(n>MCL)	TEXT	

SQL Anywhere or UltraLite data type	Adaptive Server Enterprise data type	Notes
DATE	DATE[3] or DATETIME[4]	For Adaptive Server Enterprise DATETIME, the year must be in the range 1753-9999. For SQL Anywhere and UltraLite, the time value must in the format 00:00:00.
DATETIME	DATETIME	The Adaptive Server Enterprise DATETIME values are accurate to 1/300 second. The last digit of the fractional second is always 0, 3, or 6. Other digits will be rounded to one of these three digits, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 6; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from Adaptive Server Enterprise, but for upload, the values may not be exactly the original values. If DATETIME is used for a primary key, conflict resolution may fail. In order to successfully synchronize DATETIME, you should round the fractional second to 10 milliseconds. Also, the year must be in the range 1753-9999.
SMALLDATETIME	DATETIME[4]	SQL Anywhere and UltraLite SMALLDATETIME is implemented as TIMESTAMP. The Adaptive Server Enterprise DATETIME is accurate to the minute. 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute. SQL Anywhere or UltraLite SMALLDATETIME is accurate to the microsecond. In order to successfully synchronize, SQL Anywhere or UltraLite SMALLDATETIME must be rounded to the minute. Also, the year must be in the range 1753-9999.

SQL Anywhere or UltraLite data type	Adaptive Server Enterprise data type	Notes
TIME	TIME[3] or DATETIME[4]	The Adaptive Server Enterprise TIME values are accurate to 1/300 second. The last digit of the fractional second is always 0, 3, or 6. Other digits will be rounded to one of these three digits, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 6; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from Adaptive Server Enterprise, but for upload, the values may not be exactly the original values. If TIME is used for a primary key, conflict resolution may fail. In order to successfully synchronize TIME, you should round the fractional second to 10 milliseconds.
TIMESTAMP	DATETIME	The Adaptive Server Enterprise DATETIME values are accurate to 1/300 second. The last digit of the fractional second is always 0, 3, or 6. Other digits will be rounded to one of these three digits, so, 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 6; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from Adaptive Server Enterprise, but for upload, the values may not be exactly the original values. If DATETIME is used for a primary key, conflict resolution may fail. In order to successfully synchronize DATETIME, you should round the fractional second to 10 milliseconds. Also, the year must be in the range 1753-9999.
BINARY(n=<MCL)	BINARY(n)	
BINARY(n>MCL)	IMAGE	
IMAGE	IMAGE	
LONG BINARY	IMAGE	

SQL Anywhere or UltraLite data type	Adaptive Server Enterprise data type	Notes
UNIQUEIDENTIFIER	CHAR(36)	
VARBINARY(n=<MCL)	VARBINARY	
VARBINARY(n>MCL)	IMAGE	

[1] Only applies to Adaptive Server Enterprise before version 15.0.

[2] Only applies to Adaptive Server Enterprise version 15.0 and above.

[3] Only applies to Adaptive Server Enterprise version 12.5.1 or above.

[4] Only applies to Adaptive Server Enterprise before version 12.5.1.

Adaptive Server Enterprise consolidated data types mapped to SQL Anywhere or UltraLite remote data types

The following table identifies how Adaptive Server Enterprise consolidated data types are mapped to SQL Anywhere or UltraLite remote data types. For example, a column of type DOUBLE PRECISION on the consolidated database should be type DOUBLE on the remote database.

Adaptive Server Enterprise data type	SQL Anywhere or UltraLite data type	Notes
BIGINT[1]	BIGINT	
INT	INT	
SMALLINT	SMALLINT	
TINYINT	TINYINT	
UNSIGNED BIGINT[1]	UNSIGNED BIGINT	
UNSIGNED INT[1]	UNSIGNED INT	
UNSIGNED SMALLINT[1]	UNSIGNED SMALLINT	
NUMERIC(p,s)	NUMERIC(p,s)	
DECIMAL(p,s)	DECIMAL(p,s)	
FLOAT(p)	FLOAT(p)	
DOUBLE PRECISION	DOUBLE	
REAL	REAL	
SMALLMONEY	SMALLMONEY	
MONEY	MONEY	

Adaptive Server Enterprise data type	SQL Anywhere or UltraLite data type	Notes
SMALLDATETIME	SMALLDATETIME	SQL Anywhere and UltraLite SMALLDATETIME is implemented as TIMESTAMP. The Adaptive Server Enterprise SMALLDATETIME is accurate to the minute. 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute. SQL Anywhere or UltraLite SMALLDATETIME is accurate to the microsecond. In order to successfully synchronize, SQL Anywhere or UltraLite SMALLDATETIME must be rounded to the minute. Also, the year must be in the range 1900-2078.
DATETIME	DATETIME	The Adaptive Server Enterprise DATETIME values are accurate to 1/300 second. The last digit of the fractional second is always one of 0, 3, or 6. Other digit numbers are rounded to one of these three digits, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 6; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from Adaptive Server Enterprise, but for upload, the values may not be exactly the original values. Conflict resolution may fail. In order to successfully synchronize DATETIME, you should round the fractional second to 10 milliseconds. Also, the year must be in the range 1753-9999.
DATE	DATE	For SQL Anywhere and UltraLite, the time value must in the format 00:00:00.
TIME	TIME	The Adaptive Server Enterprise TIME values are accurate to 1/300 second. The last digit of the fractional second is always one of 0, 3, or 6. Other digit numbers are rounded to one of these three digits, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 6; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from Adaptive Server Enterprise, but for upload, the values may not be exactly the original values. Conflict resolution may fail. In order to successfully synchronize TIME, it is recommended that you round the fractional second to 10 milliseconds.

Adaptive Server Enterprise data type	SQL Anywhere or UltraLite data type	Notes
CHAR(n)	VARCHAR(n)	There is no equivalence between SQL Anywhere CHAR/NCHAR and Adaptive Server Enterprise CHAR/NCHAR. SQL Anywhere CHAR/NCHAR is equivalent to VARCHAR/NVARCHAR. You should not use CHAR/NCHAR in a consolidated database column that is synchronized. If you must use non-SQL Anywhere CHAR/NCHAR, run the MobiLink server with the -b option.
VARCHAR(n)	VARCHAR(n)	
UNICHAR(n)	NVARCHAR(n)	Not available in UltraLite.
UNIVARCHAR(n)	NVARCHAR(n)	Not available in UltraLite.
NCHAR(n)	VARCHAR(n)	The Adaptive Server Enterprise NCHAR and NVARCHAR store multibyte national character strings, they are different from SQL Anywhere NCHAR and NVARCHAR. In a multibyte environment, use SQL Anywhere or UltraLite VARCHAR.
NVARCHAR(n)	VARCHAR(n)	The Adaptive Server Enterprise NCHAR and NVARCHAR store multibyte national character strings, they are different from SQL Anywhere NCHAR and NVARCHAR. In a multibyte environment, use SQL Anywhere or UltraLite VARCHAR.
TEXT	LONG VARCHAR	
UNITEXT[1]	LONG NVARCHAR	Not available in UltraLite.
BINARY(n)	BINARY(n)	
VARBINARY(n)	VARBINARY(n)	
IMAGE	LONG BINARY	
BIT	BIT	

[1] Only applies to Adaptive Server Enterprise before version 15.0.

IBM DB2 UDB data mapping

> **Note**
> Only supported data types are included here.

SQL Anywhere or UltraLite remote data types mapped to IBM DB2 UDB consolidated data types

The following table identifies how SQL Anywhere or UltraLite remote data types are mapped to IBM DB2 UDB consolidated data types. For example, a column of type BIT on the remote database should be type SMALLINT on the consolidated database.

When creating a DB2 table, you need to pay attention to the DB2 page size. DB2 has a maximum row length (MRL) based on the page size: the MRL is 4005 when the page size is 4K, 8101 when 8K, 16293 when 16K and 32677 when 32K. The length of all columns in a table can't exceed the above limitation. If a table has a BLOB or CLOB column, you count row length using the LOB locator, not BLOB or CLOB data directly. For details, see the DB2 documentation.

SQL Anywhere or UltraLite data type	IBM DB2 UDB data type	Notes
CHAR(n<MRL)	VARCHAR(n)	
CHAR(n>=MRL)	CLOB(n)	DB2 values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
LONG NVARCHAR	CLOB(n)	There is no corresponding data type in DB2. If the DB2 character set is Unicode, SQL Anywhere LONG NVARCHAR can synchronize to DB2 CLOB. UltraLite doesn't have LONG NVARCHAR.
LONG VARCHAR	CLOB(n)	

SQL Anywhere or UltraLite data type	IBM DB2 UDB data type	Notes
NCHAR(c)	VARCHAR(n) or CLOB(n)	There is no corresponding data type in DB2. If the DB2 character set is Unicode, NCHAR can synchronize to DB2 VARCHAR or CLOB. The size of SQL Anywhere NCHAR is characters and the size of DB2 VARCHAR is bytes. If you map to VARCHAR, the total bytes of NCHAR can not be bigger than MRL. Otherwise, NCHAR should map to CLOB. It is difficult to calculate the number of bytes in NCHAR(c), but it is approximately c=n/4. In general, if c is less than MRL/4, map to VARCHAR(n), but if c is greater than or equal to MRL/4, map to CLOB(n).
NTEXT	CLOB(n)	There is no corresponding data type in DB2. If the DB2 character set is Unicode, NTEXT can synchronize to DB2 CLOB.
NVARCHAR(c)	VARCHAR(n) or CLOB(n)	There is no corresponding data type in DB2. If the DB2 character set is Unicode, NVARCHAR can synchronize to DB2 VARCHAR or CLOB. The size of SQL Anywhere NVARCHAR is characters and the size of DB2 VARCHAR is bytes. If you map to VARCHAR, the total bytes of NVARCHAR can not be bigger than MRL. Otherwise, NVARCHAR should map to CLOB. It is difficult to calculate the number of bytes in NVARCHAR(c), but it is approximately c=n/4. In general, if c is less than MRL/4, map to VARCHAR(n), but if c is greater than or equal to MRL/4, map to CLOB(n).
TEXT	CLOB(n)	
UNIQUEIDENTIFIERSTR	CHAR(36)	UNIQUEIDENTIFIERSTR is not recommended for DB2. Use UNIQUEIDENTIFIER instead.
VARCHAR(n<MRL)	VARCHAR(n)	

SQL Anywhere or UltraLite data type	IBM DB2 UDB data type	Notes
VARCHAR(n>=MRL)	CLOB(n)	DB2 values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
XML	CLOB(n)	
UNSIGNED BIGINT	DECIMAL(20)	For download, DB2 values must be non-negative.
BIGINT	BIGINT	
BIT	SMALLINT	
DECIMAL(p<32,s)	DECIMAL(p,s)	The precision of SQL Anywhere DECIMAL is between 1 and 127. The maximum precision of DB2 DECIMAL is 31.
DECIMAL(p>=32,s)		Any data of SQL Anywhere DECIMAL precision greater than 31 cannot be synchronized to DB2.
DOUBLE	DOUBLE	DOUBLE is an imprecise numeric data type that is subject to rounding. When working with different types of computers, the underlying storage of DOUBLE is often different, resulting in different rounding. DOUBLE is a bad choice to use in a primary key because primary keys are looking for equality. This is especially true in a synchronization environment because the consolidated database often runs on different hardware from the remote database.
FLOAT(1-24)	REAL	FLOAT can cause problems if the consolidated and remote databases don't allow the exact same (imprecise) values. We do not test all possible values, so care must be taken. To avoid problems, do not use these types as part of a primary key.

SQL Anywhere or UltraLite data type	IBM DB2 UDB data type	Notes
FLOAT(25-53)	DOUBLE	FLOAT can cause problems if the consolidated and remote databases don't allow the exact same (imprecise) values. We do not test all possible values, so care must be taken. To avoid problems, do not use these types as part of a primary key.
UNSIGNED INTEGER	DECIMAL(11)	For download, DB2 values must be non-negative.
INTEGER	INTEGER	
NUMERIC(p<32,s)	NUMERIC(p,s)	
NUMERIC(p>=32,s)		There is no corresponding data type in DB2.
REAL	REAL	REAL can cause problems if the consolidated and remote databases don't allow the exact same (imprecise) values. We do not test all possible values, so care must be taken. To avoid problems, do not use these types as part of a primary key.
UNSIGNED SMALLINT	DECIMAL(5)	For download, DB2 values must be non-negative.
SMALLINT	SMALLINT	
UNSIGNED TINYINT	SMALLINT	For download, DB2 values must be non-negative.
TINYINT	SMALLINT	For download, DB2 values must be non-negative.
MONEY	DECIMAL(19,4)	
SMALLMONEY	DECIMAL(10,4)	
LONG VARBIT	CLOB(n)	
VARBIT(n<MRL)	VARCHAR(n)	
VARBIT(n>=MRL)	CLOB(n)	
DATE	DATE	For SQL Anywhere and UltraLite, the time value must in the format 00:00:00.

SQL Anywhere or UltraLite data type	IBM DB2 UDB data type	Notes
DATETIME	TIMESTAMP	
SMALLDATETIME	TIMESTAMP	
TIME	TIMESTAMP or TIME	SQL Anywhere and UltraLite TIME values with fractional seconds require DB2 TIMESTAMP. SQL Anywhere and UltraLite time values with fractional seconds that are always zero can use DB2 TIME.
TIMESTAMP	TIMESTAMP	
BINARY(n<MRL)	VARCHAR(n) FOR BIT DATA	
BINARY(n>=MRL)	BLOB(n)	
IMAGE	BLOB(n)	
LONG BINARY	BLOB(n)	
UNIQUEIDENTIFIER	CHAR(36)	
VARBINARY(n<MRL)	VARCHAR(n) FOR BIT DATA	
VARBINARY(n>=MRL)	BLOB(n)	

IBM DB2 UDB consolidated data types mapped to SQL Anywhere or UltraLite remote data types

The following table identifies how IBM DB2 UDB consolidated data types are mapped to SQL Anywhere or UltraLite remote data types. For example, a column of type INT on the consolidated database should be type INTEGER on the remote database.

When creating a DB2 table, you need to pay attention to the DB2 page size. DB2 has a maximum row length based on the page size: the MRL is 4005 when the page size is 4K, 8101 when 8K, 16293 when 16K and 32677 when 32K. The length of all columns in a table can't exceed the above limitation. If a table has a BLOB or CLOB column, you count row length using the LOB locator, not BLOB or CLOB data directly. For details, see the DB2 documentation.

IBM DB2 UDB data type	SQL Anywhere or UltraLite data type	Notes
SMALLINT	SMALLINT	
INT	INTEGER	
BIGINT	BIGINT	

IBM DB2 UDB data type	SQL Anywhere or Ultra-Lite data type	Notes
REAL	REAL	REAL can cause problems if the consolidated and remote databases don't allow the exact same (imprecise) values. We do not test all possible values, so care must be taken. To avoid problems, do not use these types as part of a primary key.
DOUBLE	DOUBLE	DOUBLE is an imprecise numeric data type that is subject to rounding. When working with different types of computers, the underlying storage of DOUBLE is often different, resulting in different rounding. DOUBLE is a bad choice to use in a primary key because primary keys are looking for equality. This is especially true in a synchronization environment because the consolidated database often runs on different hardware from the remote database.
FLOAT	DOUBLE	FLOAT can cause problems if the consolidated and remote databases don't allow the exact same (imprecise) values. We do not test all possible values, so care must be taken. To avoid problems, do not use these types as part of a primary key.
DECIMAL(p,s)	DECIMAL(p,s)	
NUMERIC(p,s)	NUMERIC(p,s)	
CHAR(n)	VARCHAR(n)	There is no equivalent to DB2 CHAR in SQL Anywhere. You should not use CHAR in a consolidated database column that is synchronized. If you must synchronize DB2 CHAR columns, run MobiLink server with the -b option.
VARCHAR(n)	VARCHAR(n)	
LONG VARCHAR	VARCHAR(32700)	
CLOB(n)	LONG VARCHAR	
CHAR(n) FOR BIT DATA	BINARY(n)	
VARCHAR(n) FOR BIT DATA	VARBINARY(n)	
LONG VARCHAR FOR BIT DATA	VARBINARY(32700)	

IBM DB2 UDB data type	SQL Anywhere or UltraLite data type	Notes
GRAPHIC(n)	VARCHAR(2n)	DB2 GRAPHIC will do blank-padding, but SQL Anywhere CHAR will not. We recommend that you do not use this data type. The data type GRAPHIC is only used for double-byte characters. SQL Anywhere does not have a corresponding data type. When the DB2 character set is Unicode, GRAPHIC is equivalent to CHAR.
VARGRAPHIC(n)	VARCHAR(2n)	The data type VARGRAPHIC is only used for double-byte characters. SQL Anywhere does not have a corresponding data type. When the DB2 character set is Unicode, VARGRAPHIC is equivalent to VARCHAR.
LONG VARGRAPHIC(n)	VARCHAR(32700)	The data type LONG VARGRAPHIC is only used for double-byte characters. SQL Anywhere does not have a corresponding data type. When the DB2 character set is Unicode, LONG VARGRAPHIC is equivalent LONG VARCHAR.
DBCLOB(n)	LONG VARCHAR	The data type DBCLOB(n) is only used for double-byte characters. SQL Anywhere does not have a corresponding data type. When the DB2 character set is Unicode, DBCLOB(n) is equivalent to CLOB.
BLOB	LONG BINARY	
DATE	DATE	For SQL Anywhere and UltraLite, the time value must in the format 00:00:00.
TIME	TIME	The fractional seconds values from SQL Anywhere TIME values will be truncated on download. To avoid problems, do not use fractional seconds.
TIMESTAMP	TIMESTAMP	

Oracle data mapping

Note
Only supported data types are ncluded here.

SQL Anywhere or UltraLite remote data types mapped to Oracle consolidated data types

The following table identifies how SQL Anywhere or UltraLite remote data types are mapped to Oracle consolidated data types. For example, a column of type BIT on the remote database should be type NUMBER on the consolidated database.

SQL Anywhere or UltraLite data type	Oracle data type	Notes
CHAR(n<=4000)	VARCHAR2(n byte)	Oracle VARCHAR2 allows you to specify the maximum number of bytes or characters. The maximum length of VARCHAR2 data is 4000 bytes. If you specify the character number, make sure the maximum data length is not over 4000 bytes.
CHAR(n>4000)	CLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
LONG NVARCHAR	NCLOB	Oracle CLOB and NCLOB can hold up to 4G of data. SQL Anywhere LONG VARCHAR and LONG NVARCHAR can only hold up to 2G. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
LONG VARCHAR	CLOB	Oracle CLOB and NCLOB can hold up to 4G of data. SQL Anywhere LONG VARCHAR and LONG NVARCHAR can only hold up to 2G. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.

SQL Anywhere or UltraLite data type	Oracle data type	Notes
NCHAR(c)	NVARCHAR2(c char) or NCLOB	The size of SQL Anywhere NCHAR and Oracle NVARCHAR2 indicates the maximum number of Unicode characters. The data length of Oracle NVARCHAR2 can't be over 4000 bytes. It is difficult to calculate the maximum byte length from character size. In general, if the size is over 1000, map to NCLOB, otherwise map to NVARCHAR2.
NTEXT	NCLOB	Oracle NCLOB can hold up to 4G of data. SQL Anywhere LONG VARCHAR and LONG NVARCHAR can only hold up to 2G. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
NVARCHAR	NVARCHAR2(c CHAR) or NCLOB	The size of SQL Anywhere NCHAR and Oracle NVARCHAR2 indicates the maximum number of Unicode characters. The data length of Oracle NVARCHAR2 can't be over 4000 bytes. It is difficult to calculate the maximum byte length from character size. In general, if the size is over 1000, map to NCLOB, otherwise map to NVARCHAR2.
TEXT	CLOB	Oracle CLOB can hold up to 4G of data. SQL Anywhere LONG VARCHAR and LONG NVARCHAR can only hold up to 2G. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
UNIQUEIDENTIFIERSTR	CHAR(36)	UNIQUEIDENTIFIERSTR is not recommended to use for Oracle. Use UNIQUEIDENTIFIER instead.

SQL Anywhere or UltraLite data type	Oracle data type	Notes
VARCHAR(n<=4000)	VARCHAR2(n byte)	Oracle VARCHAR2 allows you to specify the maximum number of bytes or characters. The maximum length of VARCHAR2 data is 4000 bytes. If you specify the character number, make sure the maximum data length is not over 4000 bytes.
VARCHAR(n>4000)	CLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
XML	CLOB	Oracle CLOB and NCLOB can hold up to 4G of data. SQL Anywhere LONG VARCHAR and LONG NVARCHAR can only hold up to 2G. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
UNSIGNED BIGINT	NUMBER(20)	For download, Oracle values must be non-negative.
BIGINT	NUMBER(20)	
BIT	NUMBER(1)	For download, Oracle values must be non-negative.
DECIMAL(p<=38,s)	NUMBER(p, 0<=s<=38)	In SQL Anywhere DECIMAL, p is between 1 and 127, and s is always less than or equal to p. In Oracle NUMBER, p ranges from 1 to 38, and s ranges from -84 to 127. In order to synchronize, the Oracle NUMBER scale must be restricted to between 0 and 38.
DECIMAL(p>38,s)		There is no corresponding data type in Oracle.
DOUBLE	DOUBLE PRECISION or BINARY_DOUBLE[1]	The special values INF, -INF and NAN of Oracle 10g BINARY_FLOAT and BINARY_DOUBLE cannot be synchronized with SQL Anywhere or UltraLite.
FLOAT(p)	FLOAT(p)	

SQL Anywhere or UltraLite data type	Oracle data type	Notes
UNSIGNED INTEGER	NUMBER(11)	For download, Oracle values must be non-negative.
INTEGER	INT	
NUMERIC(p<=38,s)	NUMBER(p, 0<=s<=38)	In SQL Anywhere NUMERIC, p is between 1 and 127, and s is always less than or equal to p. In Oracle NUMBER, p ranges from 1 to 38, and s ranges from -84 to 127. In order to synchronize, the Oracle NUMBER scale must be restricted to between 0 and 38.
NUMERIC(p>38,s)		There is no corresponding data type in Oracle.
REAL	REAL or BINARY_FLOAT[1]	The special values INF, -INF and NAN of Oracle 10g BINARY_FLOAT and BINARY_DOUBLE cannot be synchronized with SQL Anywhere or UltraLite.
UNSIGNED SMALLINT	NUMBER(5)	For download, Oracle values must be non-negative.
SMALLINT	NUMBER(5)	
UNSIGNED TINYINT	NUMBER(3)	For download, Oracle values must be non-negative.
TINYINT	NUMBER(3)	For download, Oracle values must be non-negative.
MONEY	NUMBER(19,4)	
SMALLMONEY	NUMBER(10,4)	
LONG VARBIT	CLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
VARBIT(n<=4000)	VARCHAR2(n byte)	
VARBIT(n>000)	CLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.

SQL Anywhere or UltraLite data type	Oracle data type	Notes
DATE	DATE[2] or TIMESTAMP	SQL Anywhere or UltraLite fractional seconds cannot be preserved when using an Oracle DATE data type which has no fractional seconds. To avoid problems, do not use fractional seconds. The year must be in the range 1-9999.
DATETIME	DATE[2] or TIMESTAMP	SQL Anywhere or UltraLite fractional seconds cannot be preserved when using an Oracle DATE data type which has no fractional seconds. To avoid problems, do not use fractional seconds. The year must be in the range 1-9999.
SMALLDATETIME	DATE[2] or TIMESTAMP	SQL Anywhere or UltraLite fractional seconds cannot be preserved when using an Oracle DATE data type which has no fractional seconds. To avoid problems, do not use fractional seconds. The year must be in the range 1-9999.
TIME	DATE[2] or TIMESTAMP	SQL Anywhere or UltraLite fractional seconds cannot be preserved when using an Oracle DATE data type which has no fractional seconds. To avoid problems, do not use fractional seconds.
TIMESTAMP	DATE[2] or TIMESTAMP	SQL Anywhere or UltraLite fractional seconds cannot be preserved when using an Oracle DATE data type which has no fractional seconds. To avoid problems, do not use fractional seconds. The year must be in the range 1-9999.
BINARY(n<=2000)	RAW(n)	
BINARY(n>2000)	BLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.

SQL Anywhere or UltraLite data type	Oracle data type	Notes
IMAGE	BLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
LONG BINARY	BLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
UNIQUEIDENTIFIER	CHAR(36)	
VARBINARY (n<=2000)	RAW(n)	
VARBINARY(n>2000)	BLOB	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.

[1] Only applies to Oracle version 10g or above.

[2] Only applies to Oracle version 8i.

Notes

The LONG data types are deprecated in Oracle 8, 8i and 9i.

For Oracle LONG data types to synchronize properly, you must check the Oracle **Force Retrieval of Long Columns** option in the ODBC data source configuration dialog of the iAnywhere Solutions Oracle ODBC driver.

Oracle consolidated data types mapped to SQL Anywhere or UltraLite remote data types

The following table identifies how Oracle consolidated data types are mapped to SQL Anywhere or UltraLite remote data types. For example, a column of type LONG on the consolidated database should be type LONG VARCHAR on the remote database.

Oracle data type	SQL Anywhere or UltraLite data type	Notes
VARCHAR2(n byte)	VARCHAR(n)	SQL Anywhere or UltraLite values can be longer than Oracle values, so make sure values are not too big when uploading.
NVARCHAR2(c char)	NVARCHAR(c)	Not available in UltraLite. SQL Anywhere or UltraLite values can be longer than Oracle values, so make sure values are not too big when uploading.

Oracle data type	SQL Anywhere or UltraLite data type	Notes
NUMBER(p,s)	NUMBER(p,s)	In SQL Anywhere NUMBER, p is between 1 and 127, and s is always less than or equal to p. In Oracle NUMBER, p ranges from 1 to 38, and s ranges from -84 to 127. To synchronize, the Oracle NUMBER scale must be 0 and 38.
LONG	LONG VARCHAR	
DATE	TIMESTAMP	The year must be in the range 1-9999.
BINARY_FLOAT	REAL	The special values INF, -INF and NAN of BINARY_FLOAT cannot be synchronized with SQL Anywhere or UltraLite. The precision of FLOAT and DOUBLE in Oracle is different from SQL Anywhere and UltraLite. The value of the data may change depending on the precision.
BINARY_DOUBLE	DOUBLE	The special values INF, -INF and NAN of BINARY_FLOAT cannot be synchronized with SQL Anywhere or UltraLite. The precision of FLOAT and DOUBLE in Oracle is different from SQL Anywhere and UltraLite. The value of the data may change depending on the precision.
TIMESTAMP(p<=6)	TIMESTAMP	When p<6, you may need to ensure SQL Anywhere or UltraLite values have the same precision. Otherwise, conflict detection may fail and/or duplicate rows may result. The year must be in the range 1-9999.
TIMESTAMP(p>6)		There is no corresponding data type in SQL Anywhere or UltraLite.
TIMESTAMP(p) WITH TIME ZONE		There is no corresponding data type in SQL Anywhere or UltraLite.
TIMESTAMP(p) WITH LOCAL TIME ZONE		There is no corresponding data type in SQL Anywhere or UltraLite.
INTERVAL YEAR (year_precision) TO MONTH		There is no corresponding data type in SQL Anywhere or UltraLite.
INTERVAL DAY (day_precision) TO SECOND(p)		There is no corresponding data type in SQL Anywhere or UltraLite.

Oracle data type	SQL Anywhere or UltraLite data type	Notes
RAW	BINARY	SQL Anywhere or UltraLite values can be longer than Oracle values, so make sure values are not too big when uploading.
LONG RAW	LONG BINARY	
ROWID	VARCHAR(64)	UROWID and ROWID are read-only and so are unlikely to be synchronized.
UROWID	VARCHAR(64)	UROWID and ROWID are read-only and so are unlikely to be synchronized.
CHAR(n byte)	VARCHAR(n)	There is no equivalence between SQL Anywhere CHAR and Oracle CHAR. SQL Anywhere CHAR is equivalent to VARCHAR. You should not use CHAR/NCHAR in a consolidated database column that is synchronized. If you must use non-SQL Anywhere CHAR, run the MobiLink server with the -b option. SQL Anywhere or UltraLite values can be longer than Oracle values, so make sure values are not too big when uploading.
NCHAR(c char)	NVARCHAR(c)	There is no equivalence between SQL Anywhere NCHAR and Oracle NCHAR. SQL Anywhere NCHAR is equivalent to NVARCHAR. You should not use NCHAR in a consolidated database column that is synchronized. If you must use non-SQL Anywhere NCHAR, run the MobiLink server with the -b option. SQL Anywhere or UltraLite values can be longer than Oracle values, so make sure values are not too big when uploading.
CLOB	LONG VARCHAR	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
NCLOB	LONG NVARCHAR	Not available in UltraLite. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.
BLOB	LONG BINARY	Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.

Oracle data type	SQL Anywhere or Ultra-Lite data type	Notes
BFILE	LONG BINARY	Download only. Oracle values can be longer than SQL Anywhere or UltraLite values, so make sure values are not too big when downloading.

Microsoft SQL Server data mapping

Note
Only supported data types are included here.

SQL Anywhere or UltraLite remote date types mapped to Microsoft SQL Server consolidated data types

The following table identifies how SQL Anywhere or UltraLite remote data types are mapped to Microsoft SQL Server consolidated data types. For example, a column of type DATE on the remote database should be type DATETIME on the consolidated database.

SQL Anywhere or UltraLite data type	Microsoft SQL Server data type	Notes
CHAR(n<=8000)	VARCHAR(n)	
CHAR(n>8000)	TEXT	
LONG NVARCHAR	NTEXT	
LONG VARCHAR	TEXT	
NCHAR(n<=4000)	NVARCHAR(c)	
NCHAR(n>4000)	NTEXT	
NTEXT	NTEXT	
NVARCHAR(n<=4000)	NVARCHAR(c)	
NVARCHAR(n>4000)	NTEXT	
TEXT	TEXT	
UNIQUEIDENTIFIERSTR	UNIQUEIDENTIFIER	
VARCHAR(n<=8000)	VARCHAR(c)	
VARCHAR(n>8000)	TEXT	
XML	XML or TEXT	For SQL Server 2005, use XML. For other versions, use TEXT.
UNSIGNED BIGINT	NUMERIC(20)	For download, values must be non-negative.
BIGINT	BIGINT	
BIT	BIT	

SQL Anywhere or UltraLite data type	Microsoft SQL Server data type	Notes
DECIMAL(p=<38,s)	DECIMAL(p,s)	SQL Server DECIMAL/NUMERIC precision ranges from 1 to 38, so p must be less than 39.
DECIMAL(p>38,s)		There is no corresponding data type in SQL Server.
DOUBLE	FLOAT(53)	
FLOAT(p)	FLOAT(p)	
UNSIGNED INTEGER	NUMERIC(11)	For download, values must be non-negative.
INTEGER	INT	
NUMERIC(p=<38,s)	NUMERIC(p,s)	SQL Server DECIMAL/NUMERIC precision ranges from 1 to 38, so p must be less than 39.
NUMERIC(p>38,s)		There is no corresponding data type in SQL Server.
REAL	REAL	
UNSIGNED SMALLINT	INT	For download, values must be non-negative.
SMALLINT	SMALLINT	
UNSIGNED TINYINT	TINYINT	For download, values must be non-negative.
TINYINT	TINYINT	For download, values must be non-negative.
MONEY	MONEY	
SMALLMONEY	SMALLMONEY	
LONG VARBIT	TEXT	
VARBIT(n<=8000)	VARCHAR(n)	
VARBIT(n>8000)	TEXT	
DATE	DATETIME	The year must be in the range 1753-9999.

SQL Anywhere or UltraLite data type	Microsoft SQL Server data type	Notes
DATETIME	DATETIME	SQL Server DATETIME values are accurate to 333.33 microseconds. The last digit of the fractional second is always rounded to one of 0, 3, or 7, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 7; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from SQL Server, but for upload, the values may not be exactly the original values. Conflict resolution may fail. To successfully synchronize DATETIME, you can round the fractional second to 10 milliseconds. The year must be in the range 1753-9999.
SMALLDATETIME	SMALLDATETIME	SQL Anywhere and UltraLite SMALLDATETIME is implemented as TIMESTAMP. The Adaptive Server Enterprise SMALLDATETIME is accurate to the minute. 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute. SQL Anywhere or UltraLite SMALLDATETIME is accurate to the microsecond. In order to successfully synchronize, SQL Anywhere or UltraLite SMALLDATETIME must be rounded to the minute. The year must be in the range 1900-2078.

SQL Anywhere or UltraLite data type	Microsoft SQL Server data type	Notes
TIME	DATETIME	SQL Server TIME values are accurate to 333.33 microseconds. The last digit of the fractional second is always rounded to one of 0, 3, or 7, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 7; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from SQL Server, but for upload, the values may not be exactly the original values. Conflict resolution may fail. To successfully synchronize TIME, you can round the fractional second to 10 milliseconds. The year must be in the range 1753-9999.
TIMESTAMP	DATETIME	SQL Server DATETIME values are accurate to 333.33 microseconds. The last digit of the fractional second is always rounded to one of 0, 3, or 7, so 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 7; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from SQL Server, but for upload, the values may not be exactly the original values. Conflict resolution may fail. To successfully synchronize DATETIME, you can round the fractional second to 10 milliseconds. The year must be in the range 1753-9999.
BINARY(n<=8000)	VARBINARY(n)	
BINARY(n>8000)	IMAGE	
IMAGE	IMAGE	
LONG BINARY	IMAGE	
UNIQUEIDENTIFIER	UNIQUEIDENTIFIER	
VARBINARY(n<=8000)	VARBINARY(n)	
VARBINARY(n>8000)	IMAGE	

Microsoft SQL Server consolidated data types mapped to SQL Anywhere or UltraLite remote data types

The following table identifies how Microsoft SQL Server consolidated data types are mapped to SQL Anywhere or UltraLite remote data types. For example, a column of type TEXT on the remote database should be type LONG VARCHAR on the consolidated database.

Microsoft SQL Server data type	SQL Anywhere or UltraLite data type	Notes
BIGINT	BIGINT	
INT	INT	
SMALLINT	SMALLINT	
BIT	BIT	
TINYINT	TINYINT	
DECIMAL(p,s)	DECIMAL(p,s)	
NUMERIC(p,s)	NUMERIC(p,s)	
MONEY	MONEY	
SMALLMONEY	SMALLMONEY	
FLOAT(p)	FLOAT(p)	
REAL	REAL	REAL can cause problems if the consolidated and remote databases don't allow the exact same (imprecise) values. We do not test all possible values, so care must be taken. To avoid problems, do not use these types as part of a primary key.
DATETIME	TIMESTAMP or DATETIME	The Adaptive Server Enterprise DATETIME values are accurate to 1/300 second. The last digit of the fractional second is always 0, 3, or 6. Other digits will be rounded to one of these three digits, so, 0 and 1 round to 0; 2, 3, and 4 round to 3; 5, 6, 7, and 8 round to 6; and 9 rounds to 10. For download, SQL Anywhere keeps the original values from Adaptive Server Enterprise, but for upload, the values may not be exactly the original values. If DATETIME is used for a primary key, conflict resolution may fail. In order to successfully synchronize DATETIME, you should round the fractional second to 10 milliseconds. The year must be in the range 1753-9999.

Microsoft SQL Server data type	SQL Anywhere or UltraLite data type	Notes
SMALLDATETIME	SMALLDATETIME	SQL Anywhere and UltraLite SMALLDATETIME is implemented as TIMESTAMP. The Adaptive Server Enterprise SMALLDATETIME is accurate to the minute. 29.998 seconds or lower are rounded down to the nearest minute; values with 29.999 seconds or higher are rounded up to the nearest minute. SQL Anywhere or UltraLite SMALLDATETIME is accurate to the microsecond. In order to successfully synchronize, SQL Anywhere or UltraLite SMALLDATETIME must be rounded to the minute. The year must be in the range 1900-2078.
CHAR(n)	VARCHAR(n)	There is no equivalence between SQL Anywhere CHAR and non-SQL Anywhere CHAR. SQL Anywhere CHAR is equivalent to VARCHAR. You should not use CHAR in a consolidated database column that is synchronized. If you must use non-SQL Anywhere CHAR, run the MobiLink server with the -b option.
VARCHAR(n)	VARCHAR(n)	
TEXT	LONG VARCHAR	
NCHAR(n)	NVARCHAR(c)	Not available in UltraLite. There is no equivalence between SQL Anywhere NCHAR and non-SQL Anywhere NCHAR. SQL Anywhere NCHAR is equivalent to NVARCHAR. You should not use NCHAR in a consolidated database column that is synchronized. If you must use non-SQL Anywhere NCHAR, run the MobiLink server with the -b option.
NVARCHAR(c)	NVARCHAR(c)	Not available in UltraLite.
NTEXT	LONG NVARCHAR	Not available in UltraLite.
BINARY(n)	BINARY(n)	
VARBINARY(n)	VARBINARY(n)	
IMAGE	LONG BINARY	
UNIQUEIDENTIFIER	UNIQUEIDENTIFIER	
XML	XML	

CHAPTER 18

Character Set Considerations

Contents

Character set considerations

Each character of text is represented in one or more bytes. The mapping from characters to binary codes is called the **character set encoding**. Some character sets used for languages with small alphabets, such as European languages, use a single-byte representation. Others, such as Unicode, use a double-byte representation. Because they use twice the storage space for each character, double-byte character sets can represent a much larger number of characters.

Conversion errors can occur or data can be lost when text using one character set must be converted to another character set. Not all characters can be represented in all character sets. In particular, single-byte character sets can represent a much smaller number of characters than multibyte systems because of the limited number of codes available.

When the character set of your MobiLink remote database is the same as your consolidated database, character conversion issues are avoided.

Text often needs to be sorted to build indexes and to prepare ordered result sets, such as directory listings. The **sort order** identifies the order of the characters. For example, a sort order typically states that the letter "a" comes before the letter "b", which comes before the letter "c".

Each database has a **collation sequence**. You set the collation sequence when you create the database, although how you do so can differ between database systems. The collation sequence defines both the character set and the sort order for that database.

Tip
Whenever possible, define the collation sequence of your remote database to be the same as that of your consolidated database. This arrangement reduces the chance of erroneous conversions.

See also

- SQL Anywhere clients: "International Languages and Character Sets" [*SQL Anywhere Server - Database Administration*]
- UltraLite clients: "UltraLite character considerations" [*UltraLite - Database Management and Reference*]
- Information specific to your RDBMS: "MobiLink Consolidated Databases" on page 3

Character set conversion during synchronization

During synchronization, characters may need to be converted from one character set to another. The following conversions occur as characters are passed between the remote application and the consolidated database.

Character set conversion during upload

The MobiLink client sends data to the MobiLink server using the character set of the remote database.

1. The MobiLink server communicates with the consolidated database using the Unicode ODBC API. To do so, the MobiLink server converts all characters received from the remote database into Unicode and sends the Unicode to the ODBC driver.
2. If necessary, the ODBC driver for the consolidated database server converts the characters from Unicode into the character set of your consolidated database. This conversion is controlled solely by the ODBC driver for your consolidated database system. Hence, behavior can differ between two different database systems, particularly systems made by different manufacturers. MobiLink synchronization works with a number of database systems. Check the documentation of your particular consolidated server and ODBC driver for details.

Character set conversion during download

1. The ODBC driver for the consolidated database system receives characters in the coding of the consolidated database. It converts these characters into Unicode to pass them through the Unicode API to the MobiLink server. This conversion is controlled solely by the ODBC driver for your consolidated database system. Check the documentation of your particular consolidated server and ODBC driver for details.
2. The MobiLink server receives characters through the Unicode ODBC API. If the remote database uses a different character set, the MobiLink server converts the characters before downloading them.

Examples

- UltraLite applications on Windows CE devices use the Unicode character set.

  When you synchronize a Windows CE application, no character conversion occurs within the MobiLink server. The server finds that data arriving from the application is already in Unicode and passes it directly to the ODBC driver. Similarly, no character set conversion is necessary when downloading data.

- All SQL Anywhere databases and all UltraLite applications on platforms other than Windows CE use the character set determined by the collating sequence of the remote database.

  When you synchronize a remote database, the MobiLink server performs character set conversions between the character set of the remote database and Unicode.

Controlling ODBC driver character set conversion

Because most consolidated databases are unlikely to use Unicode, it is important to understand how the ODBC driver for your consolidated database system converts data to and from Unicode. Some ODBC drivers use the language settings of the machine running MobiLink to determine what character set to use. In these cases, it is best if the language and code-page settings of the machine running the MobiLink server match those of the consolidated database.

Other ODBC drivers, such as the driver for Sybase Adaptive Server Enterprise, allow each connection to use a specific character set. To avoid conversion errors, the character set used by MobiLink should be set to match that of the consolidated database.

For a detailed description of how character set conversions take place in your consolidated database server's ODBC driver, consult that product's ODBC driver documentation.

CHAPTER 19

iAnywhere Solutions ODBC Drivers for MobiLink

Contents

ODBC drivers supported by MobiLink

The MobiLink server can work with a variety of consolidated databases and ODBC drivers, as shown in the table below. Some drivers, though compatible for use with MobiLink, may have functional restrictions associated with their use.

For more information about supported versions, see MobiLink Consolidated Databases.

Database	ODBC Driver
SQL Anywhere 10	SQL Anywhere 10[1]
Oracle 8i, 9i, or 10g	iAnywhere Solutions 10 - Oracle[2]
Microsoft SQL Server	Microsoft SQL Server ODBC driver[3]
Sybase Adaptive Server Enterprise 12.5.1 or later	Sybase Adaptive Server Enterprise driver[3]
IBM DB2 UDB 8.1 or 8.2 for Windows, Linux and Unix	IBM DB2 8.2 CLI driver[3]
IBM DB2 UDB 9.1 or 8.2 for Windows, Linux and Unix	IBM DB2 9.1 CLI driver[3]

[1] - Provided with SQL Anywhere version 10. See http://www.ianywhere.com/developer/technotes/odbc_mobilink.html.

[2] - Provided with SQL Anywhere version 10.0.1 and up. See "iAnywhere Solutions Oracle driver" on page 621.

[3] - Not provided with SQL Anywhere version 10. For installation and configuration instructions, see http://www.ianywhere.com/developer/technotes/odbc_mobilink.html.

iAnywhere Solutions Oracle driver

The iAnywhere Solutions 10 - Oracle ODBC driver is custom-tailored for use with iAnywhere software. This driver will not work with third-party software.

If you use Oracle with MobiLink or OMNI, you must install an Oracle client on the same machine as this Oracle driver.

The Oracle driver can be configured using the ODBC Administrator, the .odbc.ini file (in Unix), or the dbdsn utility.

The following table provides the configuration options for the Oracle driver.

Windows ODBC Administrator	dbdsn command line or .odbc.ini file	Description
Data source name	-w option in dbdsn; not applicable to *.odbc.ini*	A name to identify your data source.
User ID	UserID	The default logon ID that the application uses to connect to your Oracle database. A logon ID is required only if security is enabled on your database. If so, contact your system administrator to get your logon ID. If you leave this field blank, you are prompted for the information when you connect.
Password	Password	The password that the application uses to connect to your Oracle database. If you leave this field blank, you are prompted for the information when you connect.
SID	SID	The Oracle System Identifier that refers to the instance of Oracle running on the server. This item is required when connecting to servers that support more than one instance of an Oracle database. The SID is stored in *network/admin/tnsnames.ora* under your Oracle installation directory.
Procedures Return Results	ProcResults	You should select this field if your stored procedures can return results. The default is that procedures do not return results (not selected). If your download_cursor or download_delete_cursor scripts are stored procedure invocations, set this to yes.

Windows ODBC Administrator	dbdsn command line or .odbc.ini file	Description
Array Size	ArraySize	The size, in bytes, of the byte array used to pre-fetch rows, on a per-statement basis. The default is 60000. Increasing this value can significantly improve fetch performance (such as during MobiLink server downloads) at the cost of extra memory allocation.

Windows configuration

♦ **To create a DSN for the Oracle driver in Windows**

1. Open the ODBC Administrator:

   ♦ Choose Start ▶ Programs ▶ SQL Anywhere 10 ▶ SQL Anywhere ▶ ODBC Administrator.

   The ODBC Data Source Administrator appears.

2. Click Add.

   The Create New Data Source dialog appears.

3. Choose iAnywhere Solutions 10 - Oracle.

   The Driver Setup dialog appears.

4. Specify the configuration options you need. The fields are explained above.

5. Click Test Connection, and then click OK.

Unix configuration

On Unix, if you are setting up the driver in an ODBC system information file (typically called *.odbc.ini*), the section for this driver should appear as follows (with appropriate values entered for each field):

```
[sample_dsn_using_the_ias_odbc_driver_for_oracle]
Driver=full-path/libdboraodbc10_r.so
UserID=user-id
Password=password
SID=oracle-system-identifier
ProcResults=[yes|no]
ArraySize=bytes
```

For an explanation of each field, see above.

DBDSN configuration

To create an Oracle DSN with the dbdsn utility, use the following syntax:

dbdsn -w *data-source-name* **-or -c** *configuration-options*

The *configuration-options* are described above.

For example:


```
dbdsn -w MyOracleDSN -or -c
Userid=dba;Password=sql;SID=abcd;ArraySize=500;ProcResults=y
```

See "Data Source utility (dbdsn)" [*SQL Anywhere Server - Database Administration*].

See also

- http://www.ianywhere.com/developer/technotes/odbc_mobilink.html
- "Data Source utility (dbdsn)" [*SQL Anywhere Server - Database Administration*]

CHAPTER 20

Deploying MobiLink Applications

Contents

Introduction to MobiLink deployment

Deploying MobiLink applications involves the following activities:

- Deploy the MobiLink server into a production setting.
- If required, deploy the Redirector.
- Deploy any SQL Anywhere MobiLink clients.
- Deploy any UltraLite MobiLink clients.

This chapter describes the files you need to include in your application's install program for each of these items.

There is a Deployment wizard that can help with your deployment on Windows. See "Using the Deployment wizard" [*SQL Anywhere Server - Programming*].

Check your license agreement
Redistribution of files is subject to your license agreement. No statements in this document override anything in your license agreement. Please check your license agreement before considering deployment.

Deploying the MobiLink server

The simplest way to deploy a MobiLink server into a production environment is to install a licensed copy of SQL Anywhere onto the production machine.

However, if you are redistributing a MobiLink server in a separate install program (subject to your license agreement), you may want to include only a subset of the files. In this case, you need to include the following files in your installation.

Notes

- Test on a clean machine before redistributing.
- Files must be installed within the SQL Anywhere installation directory, with the exception of samples.
- The files should be in the same directory unless otherwise noted.
- When a location is given, the files must be copied into a directory of the same name.
- On Unix, environment variables must be set for the system to be able to locate SQL Anywhere applications and libraries. It is recommended that you use the appropriate file for your shell, either *sa_config.sh* or *sa_config.csh* (located in the directory *install-dir/bin*) as a template for setting the required environment variables. Some of the environment variables set by the sa_config files include PATH, LD_LIBRARY_PATH, SQLANY10, SQLANYSH10, and SQLANYSAMP10.
- To use Java synchronization logic, and to use the graphical administration tools (Sybase Central and the MobiLink Monitor), you must have JRE 1.5.0 installed.
- To deploy Sybase Central, see "Deploying administration tools" [*SQL Anywhere Server - Programming*].
- There is a Deployment wizard for Windows. See "Using the Deployment wizard" [*SQL Anywhere Server - Programming*].

Windows applications

All directories are relative to your SQL Anywhere installation directory.

Description	Windows files
MobiLink server	♦ *win32\mlodbc10.dll* ♦ *win32\mlsrv10.exe* ♦ *win32\mlsrv10.lic* ♦ *win32\mlsql10.dll* ♦ *win32\dbicu10.dll* ♦ *win32\dbicudt10.dll*
Language library	♦ *win32\dblgen10.dll*[1]
Synchronization stream libraries (to support version 8 and 9 clients)	♦ *win32\mlhttp10.dll* ♦ *win32\mlsock10.dll*

Description	Windows files
Java synchronization logic	♦ *java\activation.jar*[2] ♦ *java\imap.jar*[2] ♦ *java\jodbc.jar* ♦ *java\log4j.jar*[2] ♦ *java\mailapi.jar*[2] ♦ *java\mlscript.jar* ♦ *java\mlsupport.jar* ♦ *java\pop3.jar*[2] ♦ *java\smtp.jar*[2] ♦ *win32\mljava10.dll* ♦ *win32\dbjodbc10.dll* ♦ *win32\mljodbc10.dll*
.NET synchronization logic	♦ *MobiLink\setup\dnet\mlDomConfig.xml* ♦ *win32\mldnet10.dll* ♦ *win32\dnetodbc10.dll* ♦ *Assembly\v1\\iAnywhere.MobiLink.dll* ♦ *Assembly\v1\iAnywhere.MobiLink.Script.dll* ♦ *Assembly\v1\iAnywhere.MobiLink.Script.xml* ♦ *win32\mlDomConfig.xsd*
Security option for version 10 clients (mlsrv10 -x)[3]	♦ *win32\mlecc_tls10.dll* ♦ *win32\mlrsa_tls10.dll* ♦ *win32\mlrsa_tls_fips10.dll* ♦ *win32\sbgse2.dll*
Security option[3] for version 8 and 9 clients (mlsrv10 -xo)[5]	♦ *win32\mlhttps10.dll* ♦ *win32\mlhttpsfips10.dll* ♦ *win32\mlrsafips10.dll* ♦ *win32\mljrsa10.dll* ♦ *win32\mljtls10.dll* ♦ *win32\mlrsa10.dll* ♦ *win32\mltls10.dll* ♦ *win32\defaultmem.dll* ♦ *win32\libsb.dll*
Setup scripts (deploy the ones for your consolidated database)	♦ *MobiLink\setup* ♦ *MobiLink\upgrade*
mluser utility	♦ *win32\mluser.exe* ♦ *win32\mlodbc10.dll* ♦ *win32\dbicu10.dll* ♦ *win32\dbicudt10.dll*
mlstop utility	♦ *win32\mlstop.exe* ♦ *win32\dbicu10.dll*

Description	Windows files
MobiLink Monitor	♦ *java\mlmon.jar* ♦ *java\JComponents1001.jar* ♦ *java\mlstream.jar* ♦ *java\jsyblib500.jar* ♦ *Sun\JavaHelp-1_1\jh.jar* ♦ *Sun\JAXB1.0* ♦ *win32\jsyblib500.dll* ♦ *win32\mlmon.exe* For security with the Monitor:[3] ♦ *win32\mlcecc10.dll* ♦ *win32\mlcrsa10.dll* ♦ *win32\mlcrsafips10.dll* ♦ *win32\mlczlib10.dll*
Online help for the MobiLink plug-in and Monitor	♦ *java\dbmaen10.jar*[1]
MobiLink Redirector	♦ *MobiLink\redirector*
Notifier	♦ *java\activation.jar*[2] ♦ *java\jodbc.jar* ♦ *java\log4j.jar*[4] ♦ *java\mailapi.jar*[2] ♦ *java\mlnotif.jar* ♦ *java\mlscript.jar* ♦ *java\smtp.jar*[2] ♦ *win32\mljodbc10.dll*
MobiLink server files required by QAnywhere	♦ Notifier files ♦ *java\commons-logging.jar* ♦ *java\commons-codec-1.3.jar* ♦ *java\commons-httpclient-3.0.jar* ♦ *java\jsyblib500.jar* ♦ *java\log4j.jar*[4] ♦ *java\mlscript.jar* ♦ *java\mlstream.jar* ♦ *java\qaconnector.jar* ♦ *win32\jsyblib500.dll*

[1] For French, German, Japanese, and Chinese editions, substitute **en** with **fr**, **de**, **ja**, and **zh**, respectively.

[2] If you are redistributing an application, you must obtain these files directly from Sun.

[3] Transport-layer security requires that you obtain the separately-licensed SQL Anywhere security option and is subject to export regulations. To order this component, see "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

[4] If you are redistributing an application, you must obtain this file directly from Apache.

[5] You must also create a registry key called *HKEY_LOCAL_MACHINE\SOFTWARE\Certicom\libsb* and add a REG_BINARY value named **expectedtag** with the data 5B0F4FA6E24AEF3B4407052EB04902711FD991B6.

Unix applications (UNIX, Linux, and Macintosh)

All directories are relative to your SQL Anywhere installation directory.

Description	Unix files
MobiLink server	♦ *bin32/mlsrv10* ♦ *bin32/mlsrv10.lic* ♦ *lib32/libdbodm10_r.so*[3] ♦ *lib32/libmlodbc10_r.so*[3] ♦ *lib32/libmlsql10_r.so*[3] ♦ *lib32/libdbtasks10_r.so*[3] ♦ *lib32/libdbicu10_r.so*[3] ♦ *lib32/libdbicudt10_r.so*[3]
Language library	♦ *res/dblgen10.res*[1]
Synchronization stream libraries for version 8 and 9 clients (deploy the ones you use)	♦ *lib32/libmlhttp10_r.so*[3] ♦ *lib32/libmlsock10_r.so*[3]
Java synchronization logic	♦ *java/activation.jar*[2] ♦ *java/imap.jar*[2] ♦ *java/jodbc.jar* ♦ *java/log4j.jar*[5] ♦ *java/mailapi.jar*[2] ♦ *java/mlscript.jar* ♦ *java/mlsupport.jar* ♦ *java/pop3.jar*[2] ♦ *java/smtp.jar*[2] ♦ *lib32/libmljava10_r.so*[3] ♦ *lib32/libmljodbc10.so*[3]
.NET synchronization logic	♦ Not applicable
Security option for version 10 clients (mlsrv10 -x)[4]	♦ *lib32/libmlecc_tls10_r.so*[3] ♦ *lib32/libmlrsa_tls10_r.so*[3]
Security option for version 8 and 9 clients (mlsrv10 -xo)[4]	♦ *lib32/libmlhttps10_r.so*[3] ♦ *lib32/libmljrsa10_r.so*[3] ♦ *lib32/libmljtls10_r.so*[3] ♦ *lib32/libmlrsa10_r.so*[3] ♦ *lib32/libmltls10_r.so*[3]
Setup scripts (deploy the ones for your consolidated database)	♦ *MobiLink/setup/* ♦ *MobiLink/upgrade/*
mluser utility	♦ *bin32/mluser* ♦ *lib32/libmlodbc10_r.so*[3] ♦ *lib32/libdbicu10.so*[3] ♦ *lib32/libdbicudt10.so*[3]

Description	Unix files
mlstop utility	♦ *bin32/mlstop* ♦ *lib32/libdbicu10.so*[3]
MobiLink Monitor	♦ *bin32/mlmon* ♦ *java/mlmon.jar* ♦ *java/mlstream.jar* ♦ *lib32/libjsyblib500_r.so*[3] ♦ *sun/JavaHelp-1_1/jh.jar* ♦ *sun/JAXB1.0/* ♦ *java/JComponents1001.jar* ♦ *java/jsyblib500.jar*
MobiLink Redirector	♦ *Mobilink/redirector/redirector.config* ♦ *MobiLink/redirector/apache/* ♦ *MobiLink/redirector/java/* ♦ *MobiLink/redirector/MBusinessAnywhere/* ♦ *MobiLink/redirector/nsapi/*
Online help for the MobiLink plug-in and MobiLink Monitor	♦ *java/dbmaen10.jar*[1]
Notifier	♦ *java/activation.jar*[2] ♦ *java/jodbc.jar* ♦ *java/log4j.jar*[2] ♦ *java/mailapi.jar*[2] ♦ *java/mlnotif.jar* ♦ *java/mlscript.jar* ♦ *java/smtp.jar*[2]
MobiLink server files required by QAnywhere	♦ Notifier files ♦ *java/commons-codec-1.3.jar* ♦ *java/commons-httpclient-3.0.jar* ♦ *java/commons-logging.jar* ♦ *java/jsyblib500.jar* ♦ *java/log4j.jar*[5] ♦ *java/mlscript.jar* ♦ *java/mlstream.jar* ♦ *java/qaconnector.jar* ♦ *lib32/libjsyblib500_r.so*[3]

[1] For German, Japanese, and Chinese editions, substitute **en** with **de**, **ja**, and **zh**, respectively.

[2] If you are redistributing an application, you must obtain these files directly from Sun.

[3] For Solaris and Linux, the file extension is *.so*. For AIX, the file extension is *.a*. For HP, the file extension is *.sl*. For Macintosh, the file extension is *.dylib*.

[4] Transport-layer security requires that you obtain the separately-licensed SQL Anywhere security option and is subject to export regulations. To order this component, see "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

[5] If you are redistributing an application, you must obtain these files directly from Apache.

Deploying SQL Anywhere MobiLink clients

Notes

- For SQL Anywhere clients, you need to deploy a SQL Anywhere database server and the MobiLink client.

  See "Deploying Databases and Applications" [*SQL Anywhere Server - Programming*].
- If you are redistributing MobiLink synchronization clients (subject to your license agreement) you need to include the following files in your installation, in addition to those required for the SQL Anywhere database.
- The files should be in the same directory, unless otherwise noted.
- To deploy Sybase Central, see "Deploying administration tools" [*SQL Anywhere Server - Programming*].
- There is a Deployment wizard for Windows. See "Using the Deployment wizard" [*SQL Anywhere Server - Programming*].
- For CE deployment, files that are listed below in the *win32* directory are located in the *arm.30* or *arm.50* directories.

Windows applications

All directories are relative to your SQL Anywhere installation directory.

Description	Windows files
MobiLink synchronization client (dbmlsync)	♦ *win32\dbcon10.dll* ♦ *win32\dbicu10.dll* ♦ *win32\dblgen10.dll*[1] ♦ *win32\dblib10.dll* ♦ *win32\dbmlsync.exe* ♦ *win32\dbtool10.dll*
Dbmlsync integration component	♦ MobiLink synchronization client files ♦ Visual component: *win32\dbmlsynccomg.dll* ♦ Non-visual component: *win32\dbmlsynccom.dll*
Security option[2]	♦ *win32\mlcecc10.dll* ♦ *win32\mlcrsa10.dll* ♦ *win32\mlcrsafips10.dll* ♦ *win32\sbgse2.dll*
ActiveSync and HotSync utilities	♦ *win32\mlasinst.exe* ♦ *win32\mlasdesk.dll* ♦ *win32\dbcon10.exe* ♦ *ce\chip\mlasdev.dll* (where *chip* can be arm, mips, x86, .50, and so on)

Description	Windows files
Listener	♦ *win32\dblgen10.dll*[1] ♦ *win32\dblsn.exe* ♦ *win32\lsn_udp.dll* ♦ *win32\lsn_swi510.dll* ♦ *win32\maac555.dll* ♦ *win32\maac750.dll* ♦ *win32\maac750r3.dll* ♦ *win32\mabridge.dll*

[1] For French, German, Japanese, and Chinese editions, substitute **en** with **fr**, **de**, **ja**, and **zh**, respectively.

[2] Transport-layer security requires that you obtain the separately-licensed SQL Anywhere security option and is subject to export regulations. To order this component, see "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

Unix applications (UNIX, Linux, and Macintosh)

All directories are relative to your SQL Anywhere installation directory.

Description	Unix files
MobiLink synchronization client	♦ *bin32/dbmlsync* ♦ *res/dblgen10.res* ♦ *lib32/libdbcon10_r.so*[1] ♦ *lib32/libdbicu10_r.so*[1] ♦ *lib32/libdblib10_r.so*[1] ♦ *lib32/libdbtool10_r.so*[1]
Security option[2]	♦ *lib32/libmlcecc10_r.so*[1] ♦ *lib32/libmlcrsa10_r.so*[1]

[1]For Solaris and Linux, the file extension is *.so*. For AIX, the file extension is *.a*. For HP, the file extension is *.sl*. For the Macintosh, the file extension is *.dylib*.

[2] Transport-layer security requires that you obtain the separately-licensed SQL Anywhere security option and is subject to export regulations. To order this component, see "Separately licensed components" [*SQL Anywhere 10 - Introduction*].

Deploying UltraLite MobiLink clients

For UltraLite clients, the UltraLite runtime library or the UltraLite component includes the required synchronization stream functions. The UltraLite runtime library is compiled into your application. Deployment is subject to your license agreement.

See also

- Appforge: "Deploying UltraLite applications" [*UltraLite - AppForge Programming*]
- C/C++: "Deploying Palm applications" [*UltraLite - C and C++ Programming*] and "Deploying Windows CE applications" [*UltraLite - C and C++ Programming*]
- M-Business Anywhere: "Deploying UltraLite for M-Business Anywhere applications" [*UltraLite - M-Business Anywhere Programming*]
- .NET: "Lesson 5: Build and deploy application" [*UltraLite - .NET Programming*]

Deploying QAnywhere applications

QAnywhere applications require client and server files.

In addition to the files listed below, a QAnywhere application requires:

- Files from the MobiLink synchronization client, Listener, and optionally the Security sections in "Deploying SQL Anywhere MobiLink clients" on page 632. (The Listener files are required only if you are using push notifications, which is the default.)
- dbeng10 or dbsrv10 files from "Deploying database servers" [*SQL Anywhere Server - Programming*].

To deploy Sybase Central, see "Deploying administration tools" [*SQL Anywhere Server - Programming*].

Windows applications

All directories are relative to your SQL Anywhere installation directory.

Description	Windows files
QAnywhere client	♦ *qanywhere.db* ♦ .NET Framework 1.1: *Assembly\v1\iAnywhere.QAnywhere.Client.dll* ♦ .NET Framework 2.0: *Assembly\v2\iAnywhere.QAnywhere.Client.dll* ♦ .NET Compact Framework 1.0: *ce\Assembly\v1\iAnywhere.QAnywhere.Client.dll* ♦ .NET Compact Framework 2.0: *ce\Assembly\v2\iAnywhere.QAnywhere.Client.dll* ♦ *win32\qaagent.exe* or *ce\arm.30\qaagent.exe* or *ce\arm.50\qaagent.exe* or *ce\x86.30\qaagent.exe* ♦ *win32\qastop.exe* or *ce\arm.30\qastop.exe* or *ce\arm.50\qastop.exe* ♦ *win32\qany10.dll* or *ce\arm.30\qany10.dll* or *ce\arm.50\qany10.dll* or *ce\x86.30\qany10.dll* ♦ *ce\x86.30\qastop.exe* For Java clients on Windows: ♦ *win32\qany10jni.dll* ♦ *java\qaclient.jar* For Java clients on CE with J9 VM Personal Profile: ♦ *ce\arm.30\qany10jni.dll* or *ce\arm.50\qany10jni.dll* ♦ *java\qaclient.jar*

Description	Windows files
Mobile web services client	.NET clients: ♦ QAnywhere .NET client files ♦ .NET Framework 1.1: *Assembly\v1\iAnywhere.QAnywhere.WS.dll* ♦ .NET Framework 2.0: *Assembly\v2\iAnywhere.QAnywhere.WS.dll* ♦ .NET Compact Framework 1.0: *ce\Assembly\v1\iAnywhere.QAnywhere.WS.dll* ♦ .NET Compact Framework 2.0: *ce\Assembly\v2\iAnywhere.QAnywhere.WS.dll* Java clients: ♦ QAnywhere Java client files ♦ *java\iawsrt.jar* ♦ *java\jaxrpc.jar*

Registering the QAnywhere .NET API DLL

The QAnywhere .NET API dll (*Assembly\v1\iAnywhere.QAnywhere.Client.dll* or *Assembly\v2\iAnywhere.QAnywhere.Client.dll*) needs to be registered in the Global Assembly Cache on Windows (except on Windows CE). The Global Assembly Cache lists all the registered programs on your machine. When you install SQL Anywhere, the installation program registers it. On Windows CE you do not need to register the DLL.

If you are deploying QAnywhere, you must register the QAnywhere .NET API DLL (*Assembly\v1\iAnywhere.QAnywhere.Client.dll* or *Assembly\v2\iAnywhere.QAnywhere.Client.dll*) using the gacutil utility that is included with the .NET Framework.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

