

SYBASE®

Guide to Programming Interfaces

**Sybase Aleri Streaming Platform
3.1**

DOCUMENT ID: DC01291-01-0311-01

LAST REVISED: June, 2010

Copyright © 2010 Sybase, Inc.

All rights reserved.

This publication pertains to Sybase software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor. Upgrades are provided only at regularly scheduled software release dates. No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase trademarks can be viewed at <http://www.sybase.com/detail?id=1011207>. Sybase and the marks listed are trademarks of Sybase, Inc. ® indicates registration in the United States of America.

DB2, IBM and Websphere are registered trademarks of International Business Machines Corporation.

Eclipse is a trademark of Eclipse Foundation, Inc.

Excel, Internet Explorer, Microsoft, ODBC, SQL Server, Visual C++, and Windows are trademarks or registered trademarks of Microsoft Corp.

Intel is a registered trademark of Intel Corporation.

JDBC, Solaris, Sun and Sun Microsystems are trademarks or registered trademarks of Sun Microsystems or its subsidiaries in the U.S. and other countries.

Kerberos is a trademark of the Massachusetts Institute of Technology.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Netezza is a registered trademark of Netezza Corporation in the United States and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

SPARC is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc.

Teradata is a registered trademark of Teradata Corporation and/or its affiliates in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Group Ltd.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian

agencies.

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Table of Contents

About This Guide	viii
1. Purpose	viii
2. Organization	viii
3. Related Documents	ix
1. Publication/Subscription Interfaces	1
1.1. The Publication/Subscription Mechanism	1
1.1.1. Publication/Subscription Terminology	1
1.1.2. Initializing Pub/Sub Objects	2
1.1.3. Subscribing to the Sybase Aleri Streaming Platform	2
1.1.4. Publishing to the Sybase Aleri Streaming Platform	2
1.2. Record/Playback Mechanism	3
2. Publish/Subscribe API for Java	4
2.1. Overview of SP Java utilities	4
2.1.1. SP .jar files	4
2.1.2. Non-sp Utilities	4
2.1.3. Example Files	4
2.2. Design Decisions	5
2.3. Subscribing to the Sybase Aleri Streaming Platform Using Java	5
2.3.1. Set Up the Environment for Subscription Using Java	6
2.3.1.1. Configure the API Classpath	6
2.3.1.2. Set up Basic SP Objects	6
2.3.2. Set Up Java Objects for Subscription	7
2.3.2.1. Set/Get Methods For Maximum Buffer Size, Exit-On-Drop To SpSub- scription Using Java	8
2.3.2.2. SpSubscription Example	8
2.3.2.3. SpSubscriptionProjection Example	9
2.3.3. Receive/Process Subscription Updates in Java	10
2.3.3.1. Parse Sybase Aleri Streaming Platform Data	10
2.3.3.2. Inspect Parsing Errors	10
2.3.3.3. Detect Nulls/Stales	11
2.3.3.4. SHINE Flag Supports New Subscription Mode For Partial-Record Up- dates Using Java	11
2.3.3.5. SpSubscription/SpSubscriptionProjection Objects and Null Sybase Aleri Streaming Platform Field Data Values	11
2.3.4. Change/Stop Subscription in Java	12
2.3.4.1. Stop Subscription	12
2.4. Publishing to the Sybase Aleri Streaming Platform Using Java	13
2.4.1. Create Publication Objects Using Java	13
2.4.1.1. Create the SpPublication Object	13
2.4.1.2. Example: Setting Up Objects for Publication in Java	13
2.4.2. Start the Publication Connection Using Java	14
2.4.3. Publish a Collection Using Java	15
2.4.4. Set/Get Methods for Exit-on-Drop, Exit-on-Timeout Capability to SpPublica- tion Using Java	16
2.4.5. Handling Stale Data	16
2.4.6. Publication/Subscription in a High Availability (Hot Spare) Configuration ..	17
2.4.6.1. Subscription Mechanisms in a High Availability Configuration	17
2.4.6.2. Publication Mechanisms in a High Availability Configuration	18
2.5. Record/Playback using Java	18
3. Publish/Subscribe API for C++	20
3.1. Overview/General Information	20
3.1.1. Overview of SP C++ Utilities	20
3.1.2. Design Decisions for Publication/Subscription Using C++	21

3.1.3. Set/Get Methods For Maximum Buffer Size, Exit-On-Drop To SpSubscription	21
3.1.4. C++ Usage Restrictions	22
3.2. Subscribing to the Sybase Aleri Streaming Platform Using C++	22
3.2.1. Set Up Objects for SP Subscription in C++	22
3.2.1.1. Create an SpPlatform Object	22
3.2.2. Setup and Start a Subscription in C++	23
3.2.2.1. Initiate a Subscription Using SpSubscriptionProjection	24
3.2.2.2. Implement the SpObserver Interface	24
3.2.2.3. Start the Subscription Using SpSubscription	25
3.2.2.4. Start the Subscription Using SpSubscriptionProjection	26
3.2.3. Receive/Process Subscription Updates Using C++	27
3.2.3.1. Delivery to an SpObserver Notify(...) Method Implementation	28
3.2.3.2. Inspect the Subscription Parsing Errors within the SpObserver	28
3.2.3.3. SHINE Flag Supports Subscription Mode For Partial-Record Updates	28
3.2.4. Change/Stop Subscription Using C++	28
3.2.4.1. Stop Subscription	28
3.3. Publishing to the Sybase Aleri Streaming Platform Using C++	29
3.3.1. Create Objects for Publication Using C++	29
3.3.1.1. Create an SpPublication Object	29
3.3.1.2. Create SpStreamDataRecord Objects	30
3.3.2. Publish Data to the Sybase Aleri Streaming Platform Using C++	32
3.3.3. Handling Stale Data	33
3.3.4. Set/Get Methods for Exit-on-Drop, Exit-on-Timeout Capability to SpPublication Using C++	34
3.4. Record/Playback using C++	34
3.5. Special Topics for SP Publication/Subscription Using C++	36
3.5.1. Publication/Subscription In a High Availability (Hot Spare) Configuration	36
3.5.1.1. Subscription Mechanisms in a High Availability Configuration	36
3.5.1.2. Publication Mechanisms in a High Availability Configuration	37
4. Publish/Subscribe API for .NET 2.0	38
4.1. Overview/General Information	38
4.1.1. Overview of .NET Utilities for SP Publication/Subscription	38
4.1.1.1. API Library	38
4.1.1.2. Example Files	38
4.1.2. Design Decisions for SP Publication/Subscription Using .NET 2.0	39
4.1.3. Set/Get Methods For Maximum Buffer Size, Exit-On-Drop To SpSubscription Using .NET	40
4.2. Subscribing to the Sybase Aleri Streaming Platform Using .NET 2.0	40
4.2.1. Set Up the Environment for Subscription Using .NET 2.0	40
4.2.1.1. Configure the Pub/Sub API .NET 2.0 Pub/Subnet.dll	40
4.2.1.2. Initialize the SpFactory Object	41
4.2.1.3. Create the SpPlatform Object	41
4.2.2. Set Up/Start Subscription Using .NET 2.0	42
4.2.2.1. Initiate a Subscription Using SpSubscription in .NET 2.0	45
4.2.2.2. Initiate a Subscription Using SpSubscriptionProjection	47
4.2.2.3. The SpObserver Interface	49
4.2.2.4. Adding or Removing Streams from an Active Subscription	51
4.2.2.5. SHINE Flag Supports New Subscription Mode For Partial-Record Updates Using .NET	51
4.2.3. Receive/Process Subscription Updates Using .NET 2.0	51
4.2.3.1. Parse Sybase Aleri Streaming Platform Data	51
4.3. Publishing to the Sybase Aleri Streaming Platform Using .NET 2.0	52
4.3.1. Create Objects for SP Publication Using .NET 2.0	52
4.3.1.1. Create the SpPublication Object	52
4.3.1.2. Create a Data Object for Publication	52
4.3.1.3. Set/Get Methods for Exit-on-Drop, Exit-on-Timeout Capability to Sp-	

Publication Using .NET	54
4.3.2. Handling Stale Data	54
4.4. Record/Playback using .NET 2.0	55
4.5. Special Topics for SP Publication/Subscription Using .NET 2.0	56
4.5.1. Publication/Subscription in a High Availability (Hot Spare) Configuration ..	56
4.5.1.1. Subscription Mechanisms in a High Availability Configuration	57
4.5.1.2. Publication Mechanisms in a High Availability Configuration	57
5. The On-Demand SQL Interface	59
5.1. Aleri SQL Queries and Statements	59
5.2. ODBC Connectivity	59
5.3. JDBC Connectivity	60
6. The Command and Control Interface	61
6.1. Security for the On-Demand SQL Interface	61
6.1.1. Authentication Using the SQL On-Demand Interface	61
6.1.2. Encryption Using the SQL On-Demand Interface	61
7. Embeddable Aleri Streaming Platform	62
8. Plug-in Connector Framework	65
8.1. Introduction	65
8.2. Plug-in Connector Profile	65
8.3. System Parameters and Commands	68
8.4. Read Only System Parameters	69
8.5. Commands	69
8.6. User-Defined Parameters and Parameter Substitution	70
8.7. Notes on Auto Generated Parameter Files	72
8.8. A Parameter of Type configFilename	73
8.9. Other Parameter Types	73
A. Reference Guide to the Java Object Model	75
A.1. Objects for Subscription	75
A.1.1. SpFactory Object	75
A.1.2. SpPlatformParms Object	77
A.1.3. SpPlatformStatus Object	78
A.1.4. SpPlatform Object	79
A.1.5. SpStream Object	82
A.1.6. SpStreamDefinition Object	83
A.1.7. SpStreamProjection Object	84
A.1.8. Creating an SpSubscription or SpSubscriptionProjection Object	84
A.1.9. SpSubscriptionCommon Method Set	86
A.1.10. SpSubscription Method Set	88
A.1.11. SpSubscriptionProjection Method Set	89
A.1.12. SpSubscriptionEvent	90
A.1.13. SpParserReturnInfo	94
A.1.14. SpNullConstants	95
A.1.15. SpUtils	95
A.2. Objects for Publication	96
A.2.1. Stream Operation Codes	98
A.2.2. Stream Flag Values	99
A.2.3. SpStreamDataRecord Object	99
A.2.4. Create SpStreamDataRecord Objects	100
A.3. Objects for recording and playback	102
A.3.1. SpRecorder Object	102
A.3.2. SpPlayback Object	103
B. Reference Guide to the C++ Object Model	105
B.1. C++ Objects for Subscription	105
B.1.1. SpFactory Object	105
B.1.2. SpPlatformParms Object	107
B.1.3. SpPlatformStatus Object	108
B.1.4. SpPlatform Object	109
B.1.5. SpStream Object	112

B.1.6. SpStreamDefinition Object	113
B.1.7. SpStreamProjection Object	114
B.1.8. Creating an SpSubscription or SpSubscriptionProjection Object	114
B.1.9. SpSubscriptionCommon Method Set	116
B.1.10. SpSubscription Method Set	118
B.1.11. SpSubscriptionProjection Method Set	120
B.1.12. SpSubscriptionEvent	120
B.1.13. SpParserReturnInfo object	124
B.1.14. SpDataValue Object	125
B.1.15. SpBinaryData Object	126
B.2. C++ Objects for Publication	127
B.2.1. SpPublication Method Set	127
B.2.2. Stream Operation Codes	130
B.2.3. Stream Flag Values	130
B.2.4. SpStreamDataRecord Object	131
B.3. C++ Objects for Record and Playback	132
B.3.1. SpRecorder object	132
B.3.2. SpPlayback object	133
B.4. Other C++ API Classes/Methods	133
C. Reference Guide to the .NET Object Model	135
C.1. Common Service Objects for .NET	135
C.1.1. SpFactory Object	135
C.1.2. The SpPlatformParms Object	137
C.1.3. SpPlatformStatus Object	138
C.1.4. SpPlatform Object	138
C.1.5. SpStream Object	142
C.1.6. SpStreamDefinition Object	143
C.1.7. SpStreamProjection Object	143
C.2. Subscription Objects for .NET	144
C.2.1. SpSubscriptionCommon Method Set	144
C.2.2. SpSubscriptionEvent	146
C.3. Methods for Publication in .NET 2.0	150
C.3.1. SpPublication Method Set	150
C.3.2. Stream Operation Codes	153
C.3.3. Stream Flag Values	153
C.3.4. SpStreamDataRecord Object	154
C.3.5. Creating SpStreamDataRecord Objects	155
C.3.6. Other Pub/Sub API Classes	155
C.3.7. The aleri_PubSubconst namespace	156
C.4. Record and Playback objects for .NET	158
C.4.1. SpNetRecorder Object	158
C.4.2. SpNetPlayback object	159
D. Reference Guide to SQL Query Interface	161
D.1. Aleri SQL Connectivity C++ Library	161
E. Reference Guide to the Command and Control Interface	166
E.1. Command and Control Messages	166
F. Using Encryption with Java Client Applications	182
F.1. Ready To Run in Encrypted Mode	185

About This Guide

1. Purpose

This guide describes the three major functional interfaces used to develop applications for the Sybase Aleri Streaming Platform: Publish/Subscribe, Command & Control, and On-line SQL Queries.

In this context, "Subscribing" means creating a steady connection so that an outside application can receive data from the Sybase Aleri Streaming Platform. "Publishing" means creating the same type of connection to send data into the Sybase Aleri Streaming Platform. The Publish/Subscribe ("Pub/Sub") interface gives application programmers a set of reliable and versatile modules (in Java, C++ and .NET) that manage all the low-level maintenance tasks for a publication or subscription.

The Command & Control Interface can be used to control and monitor the Streaming Processor. Finally, the Sybase On-line SQL Query Interface enables applications or users to issue SQL queries to the streams.

This guide helps you write and integrate application or system components with the Sybase Aleri Streaming Platform. Higher-level utilities, which are described in the *Utilities Guide* are also available for each of these interfaces to be used without additional coding. The source code for Sybase Aleri Streaming Platform utilities such as **sp_convert**, **sp_upload**, and **sp_subscribe** is available in the release distributions; this code can also be used by developers writing components that publish or subscribe to the Sybase Aleri Streaming Platform.

2. Organization

[Chapter 1, *Publication/Subscription Interfaces*](#) Starts with an overview of the three major functional interfaces.

These interfaces are detailed in [Chapter 2, *Publish/Subscribe API for Java*](#), [Chapter 3, *Publish/Subscribe API for C++*](#), and [Chapter 4, *Publish/Subscribe API for .NET 2.0*](#) of this Guide.

These chapters cover the interface programming options relative to Publish and Subscribe, which is the most widely used interface to the Sybase Aleri Streaming Platform.

[Chapter 5, *The On-Demand SQL Interface*](#) details the interface options for on-demand queries of stream data. Currently, on-demand queries are restricted to single stream select statements, but this restriction will soon be removed. All on-demand queries are SQL-based ([Section 5.1, "Aleri SQL Queries and Statements"](#)), but the connection layer for SQL can be either ODBC ([Section 5.2, "ODBC Connectivity"](#)), JDBC ([Section 5.3, "JDBC Connectivity"](#)), or Sybase's proprietary C++ Library ([Section D.1, "Aleri SQL Connectivity C++ Library"](#)). Sybase's SQL Connectivity C++ Library is lighter and more efficient for C++ application component development than ODBC, but for third party software integration either ODBC or JDBC (depending on the client platform) may be required.

[Chapter 6, *The Command and Control Interface*](#), details the Command & Control Interface for monitoring and controlling the Streaming Processor. This interface is implemented over a collection of XMLRPC calls in the Sybase Aleri Streaming Platform. These calls may be made from a client application, or directly via the Command and Control tools (**sp_cli**, **sp_cnc**, **sp_monitor**) described in the *Utilities Guide*

The appendices provide in-depth reference information about these interfaces:

Java	Appendix A, <i>Reference Guide to the Java Object Model</i>
C++	Appendix B, <i>Reference Guide to the C++ Object Model</i>
.NET	Appendix C, <i>Reference Guide to the .NET Object Model</i>

SQL Query	Appendix D, Reference Guide to SQL Query Interface
C&C	Appendix E, Reference Guide to the Command and Control Interface
Encryption	Appendix F, Using Encryption with Java Client Applications , describes how to set up and use encryption for Java-based clients using any of the Sybase Aleri Streaming Platform interfaces.

3. Related Documents

This guide is part of a set. The following list briefly describes each document in the set.

<i>Product Overview</i>	Introduces the Aleri Streaming Platform and related Aleri products.
<i>Getting Started - the Aleri Studio</i>	Provides the necessary information to start using the Aleri Studio for defining data models.
<i>Release Bulletin</i>	Describes the features, known issues and limitations of the latest Aleri Streaming Platform release.
<i>Installation Guide</i>	Provides instructions for installing and configuring the Streaming Processor and Aleri Studio, which collectively are called the Aleri Streaming Platform.
<i>Authoring Guide</i>	Provides detailed information about creating a data model in the Aleri Studio. Since this is a comprehensive guide, you should read the <i>Introduction to Data Modeling and the Aleri Studio</i> first.
<i>Authoring Reference</i>	Provides detailed information about creating a data model for the Aleri Streaming Platform.
Guide to Programming Interfaces	<p>Provides instructions and reference information for developers who want to use Aleri programming interfaces to create their own applications to work with the Aleri Streaming Platform.</p> <p>These interfaces include:</p> <ul style="list-style-type: none">• the Publish/Subscribe (Pub/Sub) Application Programming Interface (API) for Java• the Pub/Sub API for C++• the Pub/Sub API for .NET• a proprietary Command & Control interface• an on-demand SQL query interface
<i>Utilities Guide</i>	Collects usage information (similar to UNIX® man pages) for all Aleri Streaming Platform command line tools.
<i>Administrators Guide</i>	Provides instructions for specific administrative tasks related to the Aleri Streaming Platform.
<i>Introduction to Data Modeling and the Aleri Studio</i>	Walks you through the process of building and testing an Aleri data model using the Aleri Studio.

SPLASH Tutorial

Introduces the SPLASH programming language and illustrates its capabilities through a series of examples.

Frequently Asked Questions

Answers some frequently asked questions about the Aleri Streaming Platform.

Chapter 1. Publication/Subscription Interfaces

This chapter introduces concepts about Java, C++, and .NET 2.0.

1.1. The Publication/Subscription Mechanism

This section provides general information about the Sybase Aleri Streaming Platform's Pub/Sub mechanism.

In the Sybase Aleri Streaming Platform, you publish or subscribe to data from a stream. A stream is comparable to a collection of keyed records in a database table. The publish interface allows a client application to augment or modify a stream by sending records tagged with insert, update or delete operations. The subscribe interface allows a client application to connect to a stream in the Sybase Aleri Streaming Platform and receive records tagged with insert, update or delete operations. The set of records that streams out of the Sybase Aleri Streaming Platform, forms a logical "log" for the stream. It provides a sequence of record operations when applied to the data for the stream and will keep it up with the content of the Sybase Aleri Streaming Platform.

1.1.1. Publication/Subscription Terminology

Here are key terms that will be used in this guide:

Source Stream	A basic input stream to which applications publish.
Derived Stream	A non-input stream which is computational in nature. Input to derived streams can come from other derived streams or source streams. A derived stream can be one of the following: Compute Stream, Filter Stream, Aggregate Stream, Join Stream, Pattern Stream, Union Stream, Extend Stream, Copy Stream or FlexStream.
Record	A keyed sequence of fields accompanied by an operation specification (insert, update, delete or safedelete). It can be considered similar to a row in a database table plus an operation.
Record Operations	INSERT, UPDATE, DELETE, SAFEDELETE - the basic operation on records within a stream. These are explained in detail in the appendices to the specific language API. For example, see Section B.1, "C++ Objects for Subscription" for information about the C++ objects for Publication.
Record Flags	NOACK, SHINE, - modifiers to the basic record operations. These are explained in detail in the appendices to the specific language API. For example, see Section B.1, "C++ Objects for Subscription" for information about these record flags.
Subscribe Flags	LOSSY, NOBASE, DROPPABLE, PRESERVE_BLOCKS - modifiers to stream subscriptions. These are explained in detail in the appendices to the specific language API. For example, see Section B.1.8, "Creating an SpSubscription or SpSubscriptionProjection Object"
Envelope	A collection of records packaged together to decrease network transmission overhead and increase efficiency.
Transaction Block	A collection of records grouped together to become an atomic unit when applied to a stream in the Sybase Aleri Streaming Platform. A transaction block has transactional semantics: if any record in the block cannot be applied successfully to the Sybase stream (perhaps because of an update issued against a non-existing record), no record in the transaction block will be ap-

plied.

Commit

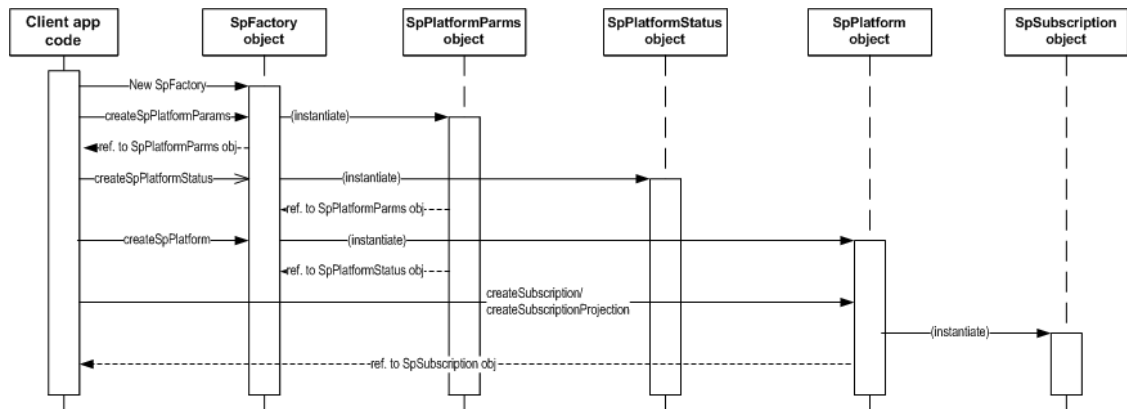
A publish API operation that upon successful return ensures all data queued in each source stream has been absorbed and written to disk if persistence via log stores is being used.

Note:

Both derived streams and source streams may be subscribed to, but applications may only publish to source streams.

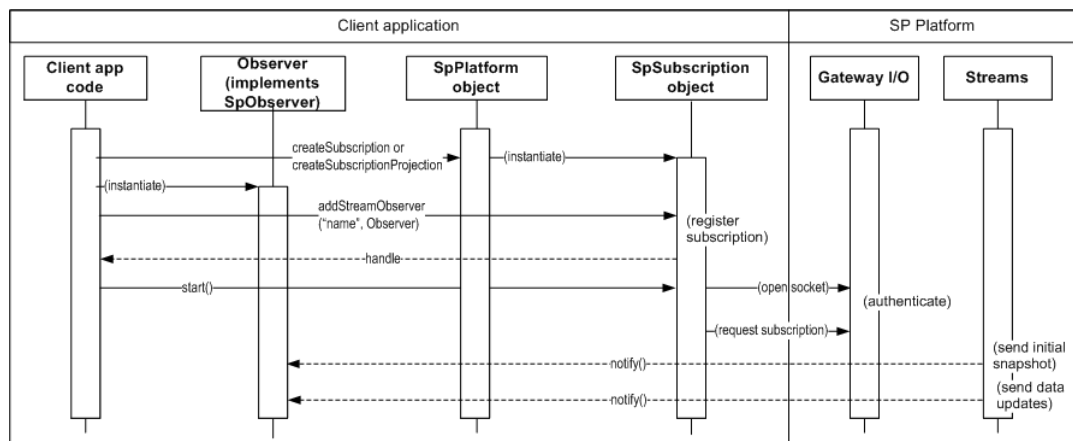
1.1.2. Initializing Pub/Sub Objects

The following Universal Modeling Language (UML) sequence diagram shows how objects defined in the API chapters of this guide interact to create the environment for a subscription or publication to the Sybase Aleri Streaming Platform.



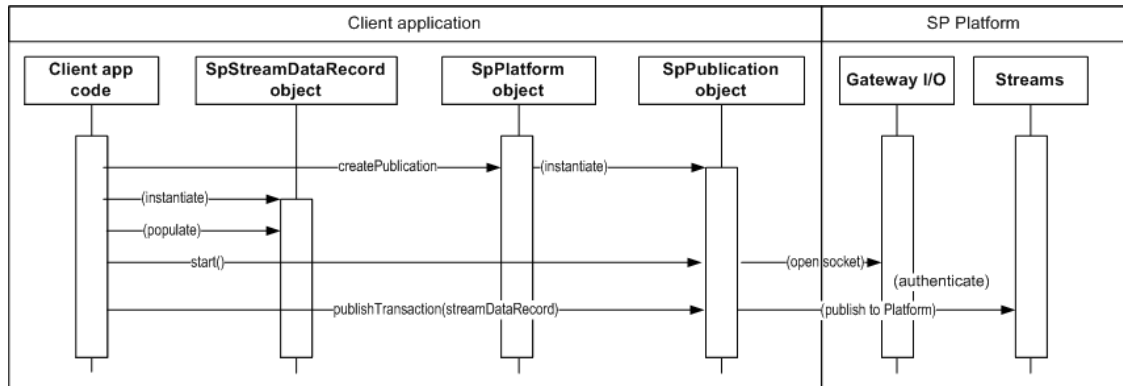
1.1.3. Subscribing to the Sybase Aleri Streaming Platform

The following diagram shows how objects defined in the API chapters of this guide interact to initialize and start up a subscription to a stream in the Sybase Aleri Streaming Platform. Returned SpStatus objects have been omitted in this diagram for clarity.



1.1.4. Publishing to the Sybase Aleri Streaming Platform

The following diagram shows how objects defined in the API chapters of this guide interact to initialize and start up a publication to a stream in the Sybase Aleri Streaming Platform. Returned SpStatus objects have been omitted in this diagram for clarity.



1.2. Record/Playback Mechanism

The Pub/Sub interfaces also include a mechanism for recording and playing back data from one or more streams. This makes it possible for an application to monitor the events occurring on one or more streams and record that information in a file. This log file includes timing information for the various events as well as the data. When played back, this recording can reproduce the exact sequence and times of events that occurred during recording. In addition, the Pub/Sub API allows client programs to control the rate of playback.

Similar to publish or subscribe, in order to record data or playback recorded data, client programs need to create the appropriate objects using the factory methods supplied by `SpPlatform`.

Chapter 2. Publish/Subscribe API for Java

This chapter describes how to create objects that use the Pub/Sub API to build Java applications that publish to and subscribe from the Sybase Aleri Streaming Platform.

Building client applications with Sybase's Java Pub/Sub API requires third party tools.

- Certified for use with Java version 1.5.0_06 or later.
- To build the included examples, you also need ant 1.6.3 or later.
- The scripts that start the Java Pub/Sub API examples use the Sybase-supplied Java version.

2.1. Overview of SP Java utilities

2.1.1. SP .jar files

<code>pubsub.jar</code>	This jar file contains the Pub/Sub API implementation.
<code>pubsub.properties</code>	This file is used as a Java “resource bundle”. It contains a list of internal error message strings, subscription event names, and so forth.

2.1.2. Non-sp Utilities

<code>xmlrpc-2.0.jar</code>	This jar file contains the Apache xmlrpc functionality.
<code>commons-codec-1.1.jar</code>	This jar file is used by the xmlrpc functionality.

2.1.3. Example Files

Inside the distribution directory, there is a directory called `$ALERI_PLATFORM_HOME/examples/clients/pubsub/java`. This directory contains the java examples which illustrate the various features of the Pub/Sub API. The example source code is in the package *com.aleri.pubsub.examples*. It contains the following source files:

<code>PubExample.java</code>	This file demonstrates publication to the Sybase Aleri Streaming Platform.
<code>SubExample.java</code>	This file demonstrates subscription to the Sybase Aleri Streaming Platform.
<code>SubExampleProjection.java</code>	This file demonstrates SQL subscription to the Sybase Aleri Streaming Platform.
<code>SubExampleSpObserver.java</code>	This file is part of the example that demonstrates subscription to the Sybase Aleri Streaming Platform.
<code>SubExampleSpObserverProjection.java</code>	This file is part of the example that demonstrates SQL subscription to the Sybase Aleri Streaming Platform.
<code>DebuggerExample.java</code>	This file demonstrates debugging support provided by the Pub/Sub API.

Compiled versions of these programs are in the `example.jar` file. There is also an ant build script that can be used to build the source files. In addition, the `pubexample.sh` shell script illustrates the required classpath and JVM command line arguments to run an example class file.

2.2. Design Decisions

The Pub/Sub API has a set of interfaces or object “types” that exposes all of the Sybase Aleri Streaming Platform publication and subscription functionality but hides the implementation details. You are strongly discouraged from using inheritance to extend the implementation classes found in the Pub/Sub API code base. This lets Sybase change underlying implementation in the future without breaking client code.

To achieve this encapsulation, most of the implementation classes found in the Pub/Sub API code base have private constructors. You are provided with the “Factory” methods for object instantiation. For example, the code for creating a new `SpPlatform` object should not be `new SpPlatform(...)`, but `SpFactory.createPlatform(...)`, and so forth.

In addition, most of the objects in the Pub/Sub API support only “get” methods, providing immutability. Complete immutability would require all “get” methods to return “copies” of internal vectors/arrays because vectors and arrays are mutable. However, this method is inefficient. Sybase chose a design that does not generate or return “copies” to the caller.

The client application programmer must maintain order and integrity for the state of these “read-only” data structures. For example, when requesting the list of column types for a stream definition, You must not modify elements within the list. If you choose to modify the list, the application encounters difficulties when it makes subsequent calls to retrieve the column types, which are now out of sync with those on the Sybase Aleri Streaming Platform.

The design of the Pub/Sub API also limits the use of exceptions generated by API routines. A method of the API usually returns a non-zero error code if the method fails. Otherwise, a method returns zero. This limitation keeps the API consistent across the set of different languages in which it is implemented.

2.3. Subscribing to the Sybase Aleri Streaming Platform Using Java

To use a Subscription to the Sybase Aleri Streaming Platform, the client application has to do the following:

1. Set up `Sp` objects using `SpFactory.init`.

2. Connect to the Sybase Aleri Streaming Platform.

```
status = SpFactory.createPlatformStatus()  
sp = SpFactory.createPlatform(params, status)
```

3. Create subscriptions.

```
sub = sp.createSubscription()  
observer = new ClientSpObserver()
```

4. Associate a stream by name with the observer.

```
handle = sub.addStreamObserver("stream1", observer);
```

5. Begin handling events.

```
sub.start()
```

After this, call `Yourclass.notify()` for each event on the associated stream, where `YourClass` is the class that implements the `SpObserver` method of your `Observer` object.

6. If necessary, stop event handling and clean up.

```
sub.stop()

delete sub

delete observer

SpFactory.destroy()
```

2.3.1. Set Up the Environment for Subscription Using Java

2.3.1.1. Configure the API Classpath

To enable your application to use the Pub/Sub API, list the `.jar` files mentioned above in the classpath. You should place the `pubsub.properties` file somewhere along the classpath so that Java can locate it when the Pub/Sub API attempts to load it from disk.

For example, if all the files mentioned above are present in the `./lib` directory, your Windows startup script may look like the following example.

```
Java -cp "./lib/xmlrpc-2.0.jar;./lib/commons-codec-1.1.jar;
./lib/PubSub.jar" ....
```

In this example, the `pubsub.properties` file is also located in the `./lib` directory.

2.3.1.2. Set up Basic SP Objects

Use the `SpFactory` class to instantiate objects that provide the Sybase Aleri Streaming Platform functionality.

The first object that you will “ask” the factory to instantiate is the `SpPlatform` object.

1. Using the `SpFactory`, create an `SpPlatformParms` object which contains all of the Sybase Aleri Streaming Platform connection information: host name, port number, username, password, and a boolean “Encrypted” flag indicating whether or not all connections will use encryption. See the set of overloaded `SpFactory.createPlatformParms(...)` methods for the set of available connection/authentication options. They include RSA authentication, as well as the High Availability configuration option.

```
SpPlatformParms parms = SpFactory.createPlatformParms(host,
port, user, password, isEncrypted);
```

2. Use `SpFactory` to make an `SpPlatformStatus` object to be used by subsequent `SpFactory` method calls to return error information.

```
SpPlatformStatus status = SpFactory.createPlatformStatus();
```


3. Using the `SpFactory`, create the `SpPlatform` object. Pass in the `SpPlatformParms` and `SpPlatformStatus` objects previously created.

If the call is successful, the `createPlatform()` method returns a fully initialized `SpPlatform` object. Otherwise, the factory method returns null, and the error code is stored in the `SpPlatformStatus` object that was passed into the `createPlatform(...)` method. The `SpPlatformStatus` object can be used to retrieve the error code and the corresponding error message (see the “else” condition in the following code fragment).

```
SpPlatform sp = SpFactory.createPlatform(parms, status);
if (sp != null)
{
    /* Use the new sp object to perform Sybase Aleri Streaming Platform related work
    */
    /* See "The SpPlatform object" */
} else {
    System.err.println("Could not create SpPlatform, error = "
        + status.getErrorCode() + ", error msg="
        + status.getErrorMessage());
}
```

2.3.2. Set Up Java Objects for Subscription

To get stream updates from the Sybase Aleri Streaming Platform, your client application must “ask” the Sybase Aleri Streaming Platform to deliver them. This process is called “subscribing” or creating a subscription. The Pub/Sub API offers two forms of subscription mechanisms (`SpSubscription` and `SpSubscriptionProjection`) that hide most of the low-level details associated with making a subscription request to the Gateway I/O process. If the Pub/Sub API is being used in a High Availability (Hot Spare) configuration, the switchover to the High Availability (Hot Spare) server is also transparently handled within the API.

The interface is simple:

```
/**
 * This interface must be implemented by all SpObserver
 * objects that are to be “notified” of events delivered by
 * their corresponding subscription objects.
 */
public interface SpObserver
{
    public String getName();

    /**
     * In the client's implementation of this interface, they would
     * simply “case” on the event types (and event ids) that they are
     * notified with and handle them appropriately.
     */
    public void notify(Collection theEvents);
}
```

There are two methods that must be implemented within the class.

1. The `getName()` method, which returns a string identifier for the `SpObserver`.
2. The `notify(Collection theEvents)` method is the link between the underlying Sybase Aleri Streaming Platform subscription, which the subscription object manages, and the client application object, which receives the subscription updates as coming in from the Sybase Aleri Streaming Platform.

As stream updates flow from the Sybase Aleri Streaming Platform to the subscription object, the subscription object forwards them to the appropriate `SpObserver` objects, where it is picked up through their `notify(Collection theEvents)` implementations.

The subscription's underlying mechanism for stream update acquisition and delivery runs in a separate thread used to manage the “read-only” Gateway I/O subscription socket. The `notify(Collection theEvents)` methods actually execute from within the context of this thread. The client application programmer must be conscious of this fact and program accordingly.

2.3.2.1. Set/Get Methods For Maximum Buffer Size, Exit-On-Drop To SpSubscription Using Java

Set methods should be called before `SpSubscription start()` method. You should check `SpPlatformStatus` after the `start()` for any possible problem. If you fail to send exit-on-close, the status is set to `SP_ERROR_SETTING_EXIT`. If you fail to send a maximum buffer size, the status is set to `SP_ERROR_SUB_SETTING_BUFFERSIZE`. You should note that `getQueueSize()` does NOT return the current queue size until after you've set it with `setQueueSize()`.

Here is an example for Java.

```
public void setQueueSize(int queue, SpPlatformStatus status);

public int getQueueSize();

public void setExitOnClose(SpPlatformStatus status);

public boolean getExitOnClose();
```

2.3.2.2. SpSubscription Example

the `sp` object represents an `SpPlatform` object instantiated through the `SpFactory.createPlatform(...)` method in the following example:

```
>
    SpSubscription sub = sp.createSubscription("MySubscription_1",
        SpSubscriptionCommon.BASE, SpSubscriptionCommon.DELIVER_PARSED,
        status); // "status" is an SpPlatformStatus object.

if (sub != null)
{
```

You must create a concrete class implementing the `SpObserver` interface. This concrete `SpObserver` class will be registered with the new `SpSubscription` object, and “notified” with `SpSubscriptionEvent` objects when the `SpSubscription` is started.

```
SpObserver spObserver = new ClientSpObserver("myObserver");
String streamName = "input";
```

You must associate the concrete `SpObserver` object(s) with a stream (or set of streams), and register the `SpObserver` with the `SpSubscription` object. This can be done using either the `SpSubscription.addStreamObserver(...)` or the `SpSubscription.addStreamsObserver(...)` method. It should be done for each `SpObserver` that is to be notified with the `SpSubscriptions` events.

```
int cookie;
cookie = sub.addStreamObserver(streamName, spObserver);

if (cookie <= 0)
{
    // SpObserver registration failed.
    return cookie;
}
```

2.3.2.3. SpSubscriptionProjection Example

The following code sample shows how to create, configure, start and stop an `SpSubscriptionProjection` object. The `sp` object represents an `SpPlatform` object instantiated previously through the `SpFactory.createPlatform(...)` method.

```
String sqlQuery = "select intData, charData from inputstream where \
intData > 100";

SpSubscriptionProjection subProj = sp.createSubscriptionProjection(
    "MySubscriptionProj_1", SpSubscriptionCommon.BASE,
    SpSubscriptionCommon.DELIVER_PARSED, sqlQuery,
    status); // "status" is an SpPlatformStatus object.

if (subProj != null)
{
    /*
     * Get the stream projection produced when the Sybase Aleri Streaming Platform
     * parsed the sql query, during the createSubscriptionProjection()
     * method call.
     */
    SpStreamProjection spStreamProj = subProj.getStreamProjection();

    /*
     * The client programmer must create a concrete
     * class implementing the SpObserver interface.
     * This concrete SpObserver class will be
     * registered with the new SpSubscriptionProjection object,
     * and "notified" with SpSubscriptionEvent objects when
     * the SpSubscriptionProjection is started.
     *
     * The SpStreamProjection is passed into the SpObserver
     * constructor.
     */
    SpObserver spObserver = new ClientSpObserver("myObserver", spStreamProj);

    /*
     * Client programmer must register the SpObserver
     * with the SpSubscriptionProjection object. This is done
     * using the SpSubscriptionProjection.addObserver(...) method.
     * This should be done for each SpObserver the client
     * wishes to be notified with the subscription events.
     */
    int cookie = subProj.addObserver(spObserver);

    if (cookie <= 0)
    {
        // SpObserver registration failed.
        return cookie;
    }

    /*
     * Once the SpObserver(s) are registered with the
     * SpSubscriptionProjection, the SpSubscriptionProjection.start()
```

```
* method is called, which starts the subscription process.
* If start up is successful, the SpObservers will be
* notified with updates sent from the Sybase Aleri Streaming Platform.
*/
System.out.println("Starting the subscription.");

rc = subProj.start();

if (rc != 0)
{
    // The subscription could not be started.
    System.out.println("Subscription could not be started, rc="+rc);
    System.out.println("Error message = " +
        SpUtils.getErrorMessage(rc));

    return rc;
}
```

A subscription created using `SpSubscription` receives updates for all columns in the stream with each event. If the subscription was created using `SpSubscriptionProjection`, an update consists of only the subset of columns defined by the SQL query. These types of updates are issued only when there is a change in one of the columns specified in the SQL query.

See [Section 5.1, “Aleri SQL Queries and Statements”](#) for some limitations related to the Sybase Aleri Streaming Platform's handling of On-Demand SQL queries, which also apply in this situation.

2.3.3. Receive/Process Subscription Updates in Java

A collection (vector) of `SpSubscriptionEvent` objects is delivered to the `SpObserver`. Each `SpSubscriptionEvent` object represents “something” that has happened that the subscription object “thought” appropriate to “notify” its registered `SpObserver` objects. A few examples would include, an `SpSubscriptionEvent` sent to the `SpObserver` if a stream update has arrived, or if the Sybase Aleri Streaming Platform shuts down.

The `notify(...)` method that you implement on your `SpObserver` objects must iterate over the array of `SpSubscriptionEvents`. It should check through each `SpSubscriptionEvent`, uniquely identified by an `EventId`, to determine what action should be taken.

2.3.3.1. Parse Sybase Aleri Streaming Platform Data

See the methods described in [Section A.1.12, “SpSubscriptionEvent”](#).

2.3.3.2. Inspect Parsing Errors

There is a “special” event id defined within the `SpSubscriptionEvent` interface as `SpSubscriptionEvent.EVID_PARSING_ERROR`. This event is generated when the subscription object's message parser encounters an error while parsing the message delivered from the Sybase Aleri Streaming Platform.

In case of error, the partial results that are successfully parsed up to the point of error are stored for possible inspection by the `SpObserver`. The `SpObserver` is not obligated to look at them. However, you may want to use this information for debugging purposes. In the case where this event ID is encountered, and “partial” results that were parsed before the error was encountered, the `getData()` method of the `SpSubscriptionEvent` passed to the `SpObserver notify()` method returns a collection (vector) of two elements.

- The first element of the vector is an object of type `SpParserReturnInfo`, which has parsing error information stored in it.
- The second element of the vector is another `SpSubscriptionEvent` that has an event ID of `SpSubscriptionEvent.PARSED_PARTIAL_FIELD_DATA`. The `getData()` method of this event object returns a collection (vector) of the stream update information that was successfully

parsed up to the point of error.

A data row is passed as an `SpSubscriptionEvent` with the ID of `SpSubscriptionEvent::EVTYPE_PARSED_DATA` or `SpSubscriptionEvent::EVTYPE_BINARY_DATA`. The choice is based on the subscription type requested. If it's parsed data, client programs can retrieve data as a vector of `SpDataValue` objects whereas the type field in an `SpDataValue` object specifies the data type which the object contains.

2.3.3.3. Detect Nulls/Stales

There is a class called `SpNullConstants` that contains a set of special static objects used to represent null values that are either returned from the Sybase Aleri Streaming Platform (via `SpSubscription/SpSubscriptionProjection` objects or sent to the Sybase Aleri Streaming Platform (via `SpPublication` objects). There is a null object for each of the different data types supported by the Sybase Aleri Streaming Platform.

The Sybase Aleri Streaming Platform creates an `SpDataValue` object for fields that are null. In order to determine if a value is null, client programs need to examine the type field in the `SpDataValue` object which will be set to the constant `DataTypes::NULLVALUE`. You also need to check the 'null' flag in the contained `dataValue` object. For example:

```
SpDataValue * dv = data->at(i);
if (dv->type == DataTypes::NULLVALUE || dv->dataValue.null) {
}
```

2.3.3.4. SHINE Flag Supports New Subscription Mode For Partial-Record Updates Using Java

The SHINE flag can support a new subscription mode for partial-record updates in Java with `SpSubscriptionCommon.SHINE`.

The following is an example of how to use this mode:

```
int flags = SpSubscriptionCommon.BASE | SpSubscriptionCommon.SHINE;
SpSubscription sub = sp.createSubscription(name, flags, deliveryType, status);
```

2.3.3.5. SpSubscription/SpSubscriptionProjection Objects and Null Sybase Aleri Streaming Platform Field Data Values

When a subscription object parses an update message sent from the Sybase Aleri Streaming Platform, each null field data value returned from the Sybase Aleri Streaming Platform is mapped onto the appropriate `SpNullConstants` object mentioned above. This design decision creates client code that doesn't care about null field data values or does not explicitly check everywhere for the Java null keyword. Each null field value is represented by a reference to an `SpNullConstant` object of the appropriate type. Numeric values default to zero, date/time values to the epoch, string values to "NULL". The client code can simply let the null field values returned from the Sybase Aleri Streaming Platform fall through the client code without requiring special null logic.

But, a client application that requires special null handling logic can test for the appropriate `SpNullConstant` object reference.

SpPublication Objects and Null Platform Field Data Values

The client application program uses an `SpPublication` object to publish data to the Sybase Aleri Stream-

ing Platform. The field data sent to the Sybase Aleri Streaming Platform is contained within a collection/vector of field data objects. To indicate a null field data value for a particular field, the publishing program can simply set the reference to the field data object to the Java keyword as null —. Alternatively, set the field data reference to the appropriate null value representation within the `SpNullConstants` object.

Do not use `SpNullConstants` values when publishing data to the Sybase Aleri Streaming Platform. Instead, set the object values to java null and use the `==` operator rather than the `.equals` function when comparing to null during subscription.

2.3.4. Change/Stop Subscription in Java

2.3.4.1. Stop Subscription

The `stop()` method shuts down the subscription mechanism. The `stop()` method closes the socket connection and stops the thread that was used to read, parse, and deliver the Sybase Aleri Streaming Platform updates to the `SpObserver` objects.

Example:

```
/*
 * Pause waiting for input from the keyboard before
 * making the call to stop the subscription.
 */
System.out.println(
    "Hit any key on the keyboard to stop the subscription...");
try
{
    BufferedReader in =
        new BufferedReader( new InputStreamReader(System.in) );
    in.readLine();
} catch (IOException ex) {
    System.out.println("ERROR reading from standard input, ex = " + ex);
}
```

The client programmer can invoke the `stop()` method in order to terminate the running `SpSubscription`.

```
System.out.println("Stopping the subscription.");

rc = sub.stop();

if (rc != 0)
{
    // Problems stopping the subscription.
    System.out.println("Problems stopping the
subscription, rc="+rc);
    System.out.println("Error message =" +
        SpUtils.getErrorMessage(rc));

    return rc;
}
}
else
{
    // Could not create the subscription object.
    rc = status.getErrorCode();
    System.out.println("Could not create subscription object,
error="+rc);
    System.out.println("Error message =" +
        status.getErrorMessage());
}
```

```
}
```

2.4. Publishing to the Sybase Aleri Streaming Platform Using Java

The Pub/Sub API gives you the ability to "publish" stream information to the Sybase Aleri Streaming Platform, using an object of type `SpPublication`.

Whether your program is publishing static data, such as a reference table, or dynamic data, such as stock market data, to the Sybase Aleri Streaming Platform, the same mechanism is used.

2.4.1. Create Publication Objects Using Java

2.4.1.1. Create the `SpPublication` Object

The first step in using the Pub/Sub API to submit "publications" (stream data) to the Sybase Aleri Streaming Platform is to create an `SpPublication` object. This is done by using a factory method provided by the `SpPlatform` object that must have been instantiated previously, as with the Pub/Sub API subscription mechanism. The signature of the `SpPlatform` factory method used to create `SpPublication` objects is as follows:

```
SpPublication createPublication(String name, SpPlatformStatus status);
```

Where:

`name` is an identifier assigned by the client application to the `SpPublication` object.

`status` is an object that returns error code information back from the `createPublication(...)` factory method if the `SpPublication` object creation has failed.

The following example shows how to use the `SpPlatform` object called `sp` to create an `SpPublication` object:

```
SpPublication pub = sp.createPublication("MyPublication_1", status);
```

In the above example, `status` is an `SpPlatformStatus` object that was created previously with the `SpFactory.createPlatformStatus()` factory method.

2.4.1.2. Example: Setting Up Objects for Publication in Java

The following is a small example of how the client application programmer can use the `createStreamDataRecord(...)` factory method to create an `SpStreamDataRecord` object that can be published to the Sybase Aleri Streaming Platform:

1. The source stream is called "input", and has the following record layout:

```
/*
 *
 * int, string, double, date, int, string, double, date
 *
 */
```

```
Collection fieldData = new Vector(8);

fieldData.add(new Integer(104));
fieldData.add("do_mystring");
fieldData.add(new Double(5.7));
fieldData.add(new Date(01));
fieldData.add(new Integer(200));
fieldData.add("do_mystring2");
fieldData.add(new Double(8.9));
fieldData.add(new Date(01));

SpStream stream = sp.getStream("input");
```

2. Use the `createStreamDataRecord(...)` factory method to bundle up the stream, fieldData vector, stream op code, and stream flags into an `SpStreamDataRecord` object.

`SpStreamDataRecord` objects are the basic elements of publication. You can publish them one at a time, or as a group, with or without transaction blocks.

All the `SpStreamDataRecords` within a group must reference the same stream to publish a single transaction.

```
SpStreamDataRecord sdr = SpFactory.createStreamDataRecord(
    stream,
    fieldData,
    SpGatewayConstants.SO_UPSERT,
    SpGatewayConstants.SF_NULLFLAG,
    status);

if (sdr == null)
{
    System.out.println("Could not createStreamDataRecord, status=" +
        status.getErrorCode());
    System.out.println("Error Message:" +
        status.getErrorMessage());
    return status.getErrorCode();
}
```

The client application programmer can create a large number of these `SpStreamDataRecord` objects, and place them within a common vector. Next, the application programmer can use one of the `SpPublication`'s publishing methods to send all rows of stream data that are stored in the vector to the Sybase Aleri Streaming Platform, either individually or by using transactions.

2.4.2. Start the Publication Connection Using Java

When an `SpPublication` object is started, the following events take place:

1. The `SpPublication` object creates a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process.
2. The `SpPublication` object authenticates with the Sybase Aleri Streaming Platform.
3. The `start()` method returns a zero back to the caller indicating that the `SpPublication` object was successfully started. If there is an error, the `start()` method will return a non-zero error code.

The `SpUtils.getErrorMessage(errorCode)` method can be used to get the specific error message.

Unlike the `SpSubscription` mechanism, the `SpPublication` mechanism does not use a separate thread to manage the publication. Behind the scenes, a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process is used to transmit and read stream data to the Sybase Aleri Streaming Platform response associated with each individual request. Unless otherwise specified in the flag values used when publishing data, a publication request is synchronous. You can call one of the `publish` methods and wait for the Sybase Aleri Streaming Platform to respond with an “ack” or “nak”.

However, there is a special stream flag, `SpGatewayConstants.SF_NOACK`, that can be used to make an asynchronous publication request. When this flag is specified, the `publish` method sends the request out to the Gateway I/O process and returns control immediately back to the caller, without waiting for a response from the Sybase Aleri Streaming Platform.

2.4.3. Publish a Collection Using Java

The following example shows how to publish a collection/vector of `SpStreamDataRecord` objects as a single transaction, where `sp` is an `SpPlatform` object that was previously instantiated and `streamInputData` is a vector that contains a number of `SpStreamDataRecord` objects.

```
/*
 * Create the publication object associated with the
 * platform.
 */
String name = "testPub_1";
SpPublication pub = sp.createPublication(name, status);
if (pub == null)
{
    System.out.println("Couldn't create a publication object, status=" +
        status);
    System.out.println("Error message = " + status.getErrorMessage());
    return status.getErrorCode();
}

/*
 * Start the publication object (this opens up a GW I/O
 * socket connection). Don't forget to eventually close
 * down the SpSubscription object (via the "stop()" method,
 * later on when you are finished using it,
 */

rc = pub.start();

if (rc != 0)
{
    System.out.println("Couldn't start the publication object.");
    System.out.println("Error message = " +
        SpUtils.getErrorMessage(rc));
    return rc;
}

/*
 * Publish the collection/vector of SpStreamDataRecord
 * objects as one big transaction.
 */

rc = pub.publishTransaction(streamInputData,
    SpGatewayConstants.SO_INSERT,
    SpGatewayConstants.SF_NULLFLAG,
    0);
```

```
if (rc != 0)
{
    System.out.println("Couldn't publish the transaction.");
    System.out.println("Error message = " +
        SpUtils.getErrorMessage(rc));
    return rc;
}
```

2.4.4. Set/Get Methods for Exit-on-Drop, Exit-on-Timeout Capability to SpPublication Using Java

Set methods should be called before SpPublication start() method. SpPlatformStatus should be checked after the start() for any possible problem. If you fail to send exit-on-close, the status is set to SP_ERROR_SETTING_EXIT. If you fail to send exit-on-timeout, the status is set to SP_ERROR_PUB_ERROR_SETTING_EXIT.

If setFinalizer and setExitOnTimeout are called, the second call returns SP_ERROR_PUB_EXIT_ALREADYSET (setFinalizer) or SP_ERROR_PUB_ACTION_ALREADYSET (setExitOnTimeout).

The following is an example using Java:

```
public void setExitOnTimeout(int timeout, SpPlatformStatus status);
public int getExitOnTimeout();
public void setExitOnClose(SpPlatformStatus status);
public boolean getExitOnClose();
```

2.4.5. Handling Stale Data

When a publishing source stops sending data to the Sybase Aleri Streaming Platform, the previously published data is retained. Depending on how long it has been since the last update, you may not want this data to be used as if it were current. The publish/subscribe APIs include two functions to enable publishers to handle this data.

The setFinalizer() function sets a timeout value (in milliseconds) and an SQL statement action. If the Sybase Aleri Streaming Platform receives no data on this connection within the specified time, the SQL statement is run. This SQL statement can perform any of the following actions:

- Delete previously published data.
- Mark previously published data as stale (via a field for that purpose in the data).
- Perform some other determined action on the source streams (and, consequently, the derived streams from these source streams).

In the following example, if the data is not updated within 1000 milliseconds, it is deleted.

```
setFinalizer(1000, "delete from Positions", status)
```

The `sendHeartbeat()` function sends a keep-alive message to the Sybase Aleri Streaming Platform. This function can be used to keep the connection alive and prevent the SQL statement from running, if `setFinalizer()` has previously been called. As the following example shows, the `sendHeartbeat()` function takes no arguments.

```
sendHeartbeat()
```

2.4.6. Publication/Subscription in a High Availability (Hot Spare) Configuration

The Sybase Aleri Streaming Platform can be started with a dual server configuration, in which one server is the primary server and the other is considered the “Hot Spare” or secondary server. See the *Administrators Guide* for more information on how to start a High Availability/Hot Spare server configuration. If the Pub/Sub API is made aware of the High Availability Streaming Processor configuration, it will switch over to the secondary server if the primary server goes down. If this happens, any active `SpSubscription`, `SpSubscriptionProjection`, and `SpPublication` objects have to be re-established on the secondary server.

The Pub/Sub API is made aware of the High Availability configuration through the configuration contents of the `SpPlatformParms` object that was passed into `SpFactory.createPlatform()`. Refer to the set of overloaded `SpFactory.createPlatformParms(...)` methods, and the `SpPlatformParms` object for High Availability configuration detail.

2.4.6.1. Subscription Mechanisms in a High Availability Configuration

When the primary server goes down, the underlying subscription thread receives an exception on the Gateway I/O socket connection used to receive the stream updates being delivered from the primary server. When this occurs, the Pub/Sub API recognizes that there is a High Availability (Hot Spare) configuration and attempts to connect to the secondary server and re-establish the subscription. After connecting to the secondary server, the Pub/Sub API waits for the secondary server to internally change its state to that of a primary server. Afterward, the subscriptions are re-established. During the switch over to the secondary server, the subscription object delivers a succession of events to the `SpObserver` objects listening for `SpSubscriptionEvents`. Typically, these events are:

- `SpSubscriptionEvent.EVID_COMMUNICATOR_HALTED`

This event will be delivered to the `SpObserver` when the exception is received on the socket receiving the subscription messages from the primary Sybase Aleri Streaming Platform server.

- `SpSubscriptionEvent.EVID_HOT_SPARE_SWITCH_OVER_INITIATED`

This event will be delivered to the `SpObserver` when the Pub/Sub API recognizes that a connection attempt should be made to the High Availability server.

- `SpSubscriptionEvent.EVID_HOT_SPARE_SWITCH_OVER_SUCCEEDED`

This event is delivered to the `SpObserver` when the connection to the Hot Spare server succeeds.

- `SpSubscriptionEvent.EVID_HOT_SPARE_SWITCH_OVER_FAILED`

This event is delivered to the `SpObserver` when the connection to the Hot Spare server fails.

If the switch over to the High Availability (Hot Spare) server succeeds, the subscription(s) will be re-established using the same delivery flag values that were originally used when the subscription(s) were established against the primary server. This means that if the subscription originally requested the BASE

snapshot of the stream, the new subscription now going against the Hot Spare server, will request the BASE snapshot of the stream as well. It is up to you to determine what needs to be done with the contents of the snapshot received from the High Availability (Hot Spare) server.

When there is a successful switch over to the High Availability (Hot Spare) machine, the `SpPlatform` object performs some internal bookkeeping to ensure that the `SpPlatform.getHost()` and `SpPlatform.getPort()` methods will return the host name and port number of the new primary server.

2.4.6.2. Publication Mechanisms in a High Availability Configuration

When the primary server goes down during an attempt by the client application to send a publication request, the `SpPublication` object will detect this and attempt to perform a switch over to the Hot Spare machine. If the switchover is successful, the publication object will then attempt to re-send the data to the new primary server. If the publication can not take place, a non-zero error code will be returned to the caller indicating the problem.

You should treat the secondary server (Hot Spare) within a High Availability Sybase Aleri Streaming Platform configuration as a passive server. The program should never logon and send data to a secondary server while the primary server is alive and well. It is the responsibility of the running High Availability Sybase Aleri Streaming Platform configuration to manage both the primary and secondary servers appropriately. If the primary server goes down in a High Availability configuration, the secondary server will take over and become the new primary server. Once the secondary server becomes the primary server, data can then be published to the new primary server. Remember, the Pub/Sub API will not publish any data until the Hot Spare server switches its state to primary.

2.5. Record/Playback using Java

In order to record data from the Sybase Aleri Streaming Platform, a client program needs to create and configure an `SpPlatform` object in the same way you would for subscribing or publishing. Once an `SpPlatform` object has been created, you create an `SpRecorder` object using the factory method `createRecorder(...)`. If `SpRecorder` is successfully created, the program calls the `start` method. This spawns a background thread which subscribes to the configured streams and records all events for those streams. Recording will stop and the spawned thread terminate once the configured number of records have been processed or if the calling program calls `stop()`. Recording can be monitored by calling the `getRecordCount()` function which returns the number of records processed so far.

```
// Initialize SpFactory ...
// Create SpStatus and SpParms objects ...
// Create platform object ...

// Create recorder. It needs the following parameters to run
// recorder name (String) : a name to identify the instance of the recording object
// recorder file (String) : name of the file to store the recorded information
// streams                : a java.util.Collection, containing names of streams to record events for
// flags (int)            : recording options (encrypted/RSA/get base data)
// max records            : maximum number of data records to record
// status                 : returns error messages if any

// init recorder parameters - recorder name, filename, streams, etc

SpRecorder recorder = spPlatform.createRecorder(recName, recFile, streams, flags, maxRecords, status);

if (null == recorder) {
    System.out.println("Error starting recorder - " + status->getErrorMessage());
    // cleanup ... and exit
} else
    recorder.start();

// Wait, monitor, etc ...

// To stop recording
recorder.stop();
```

To playback recorded data, a client program creates an `SpPlayback` object using the factory method in `SpPlatform`. Among particular interest is the 'scale' parameter. This is a double that can be used to scale the rate of playback as a factor of the original recorded rate (for example, twice as fast or half as slow). Values -1 to 1 have no effect; data is played back at the rate it was recorded. A value greater than 1 speeds up playback by that factor (for example, a value of 2 doubles the playback speed). A value less than -1 slows down playback by that factor (for example, a value of -3 will slow down playback by a factor of 3). The scale can be changed dynamically while playback is in progress.

```
// Initialize SpFactory ...
// Create SpStatus and SpParms objects ...
// Create platform object ...

// Create playback. It needs the following parameters to run
// playback name (String) : a name to identify the instance of the playback object
// playback file (String) : name of the file containing previously recorded data
// scale (double)          : allows to scale the playback rate
// max records             : maximum number of data records to playback
// status                  : returns error messages if any

// init playback parameters - recorder name, filename, streams, etc
SpPlayback playback = spPlatform.createPlayback(playName, playFile, scale, maxRecords, status);

if (null == playback) {
    System.out.println("Error starting recorder - " + status->getErrorMessage());
    // cleanup ... and exit
} else {
    playback.setSendUpsert(true); // optionally enable converting opcodes to UPSERT
    playback.start();
}

// Wait, monitor, etc ...

// To stop recording
playback.stop();
```

Chapter 3. Publish/Subscribe API for C++

3.1. Overview/General Information

This chapter explains how to use the Publish and Subscribe (Pub/Sub) C++ API to create client applications that communicate with the Sybase Aleri Streaming Platform.

Building client applications with Sybase's C++ Pub/Sub API requires third party tools.

- Certified with GNU g++ compiler version 4.2.1 on Linux and Solaris.
- Certified with Microsoft Visual C++ compiler 2005 on Windows.
- The example makefiles for Linux and Solaris require GNU gmake version 3.80 in addition to the specified compiler.

3.1.1. Overview of SP C++ Utilities

Sybase provides files that support the Pub/Sub API for C++ in the following directories:

- `./include/PubSub/` - This directory (located in the distribution package) contains the set of C++ header interface files to be used by a client application developer for writing programs to either publish or subscribe from the Sybase Aleri Streaming Platform.
- `./include/PubSub/impl/` - This directory contains the implementation header files for the Sybase Aleri Streaming Platform C++ Pub/Sub API.

You should not modify these files.

The following support files are also provided:

- `./lib/libPubSub.a` - This is the library that you should link to when developing programs against the Sybase Aleri Streaming Platform. It contains the implementations of the Pub/Sub interfaces. The library is located within the "lib" directory provided in the distribution.
- `./examples/clients/pubsub/cpp` - This directory contains three examples:
 - `pubexample.cpp` demonstrates how to publish data to a specified stream to the Sybase Aleri Streaming Platform.
 - `subexample.cpp` shows how to subscribe for a specified stream to the Sybase Aleri Streaming Platform.
 - `subprojexample.cpp` explains how to subscribe for a specified stream with projection (using an SQL statement) to the Sybase Aleri Streaming Platform.

This directory also contains a Makefile that builds the three examples and demonstrates how to compile and link C++ programs that use the Pub/Sub C++ API.

The Pub/Sub library includes code for the various authentication mechanisms supported by Sybase Aleri Streaming Platform, including Kerberos. It requires that the SASL dynamic libraries shipped with the Sybase Aleri Streaming Platform be present at runtime. These libraries are located in the `$PLATFORM_HOME/lib` folder.

The `pubexample.cpp` file, which demonstrates how to publish source stream data to the Sybase Aleri Streaming Platform using transaction blocks. The files `subexample.cpp` and `SubExampleSpObserver.cpp` demonstrate how to subscribe to a stream running on the Sybase Aleri Streaming Platform. It is highly recommended that you look at these examples. Although small, it can be used as “boiler plate” code to create your own Pub/Sub client applications.

Visual Studio® 8.0 project and solution files have been included as part of the Windows package. You can build these examples using the Aleri Studio or typing **nmake** on the command line in the folder containing these examples. The **nmake** and **devenv.com** executables must be in the path as well.

3.1.2. Design Decisions for Publication/Subscription Using C++

The design of the Pub/Sub API provides a set of interfaces (or object “types”) that shows all of the Sybase Aleri Streaming Platform functionality while hiding implementation details. It is strongly recommended that you do not use inheritance to extend the implementation classes found in the Pub/Sub API code base. Sybase software has the ability to change the API’s underlying implementation in the future without breaking client code.

To achieve certain encapsulation goals, most of the implementation classes found in the Pub/Sub API code base have private constructors; you are provided with “Factory” methods for object instantiation. For example, the code for creating a new `SpPlatform` object would not be `new SpPlatform(...)`, but `SpFactory::createPlatform(...)`.

Most objects in the Pub/Sub API support only `get` methods, providing a degree of immutability. Complete immutability would require all `get` methods to return “copies” of internal vectors/arrays because vectors and arrays are mutable. However, this method is quite inefficient, so Sybase software is designed not to generate or return “copies” to the caller.

You must maintain order and integrity for the state of these “read-only” data structures. For example, when requesting the list of column types for a stream definition, the program must treat it as a “read-only” list and must not modify elements within the list. If the program does modify the list, the application encounters difficulties when it makes subsequent calls to retrieve the column types, because its out of sync with those on the Sybase Aleri Streaming Platform.

The design of the Pub/Sub API reflects the decision to avoid using exceptions generated by API routines. Each method of the API usually returns a non-zero error code if it fails. Otherwise, a method returns a zero to indicate success. This design choice keeps the API consistent across different languages.

When a Pub/Sub API method returns a non-zero return code, you can call the `SpUtils.getErrorMessage()` method (passing it the non-zero return code) to get the specific error message text. Those API methods which take an `SpStatus` parameter may use `SpStatus::getErrorCode()` and/or `SpStatus::getErrorMessage()` to retrieve respectively, the error code and error message.

Most classes within the Pub/Sub API have the prefix “Sp”, which stands for “Streaming Processor.” For example, `SpFactory`, `SpPlatform`, `SpSubscription`, `SpPublication` and `SpStatus`. The Pub/Sub API also makes use of a C++ namespace (`aleri_pubsub`) to further protect the class names from collision with application names.

3.1.3. Set/Get Methods For Maximum Buffer Size, Exit-On-Drop To SpSubscription

Set methods should be called before the `SpSubscription->start()` method. You should check `SpPlatform-Status` after the `start()` for any possible problem. If you fail to send exit-on-close, the status is set to

SP_ERROR_SETTING_EXIT. If you fail to send a maximum buffer size, the status is set to SP_ERROR_SUB_SETTING_BUFFERSIZE. You should note that `getQueueSize()` does NOT return the current queue size until after you've set it with `setQueueSize()`.

Here is an example for C++.

```
void setQueueSize(const uint32_t queue, SpPlatformStatus * status);
int getQueueSize();
void setExitOnClose(SpPlatformStatus * status);
bool getExitOnClose();
```

3.1.4. C++ Usage Restrictions

Some of the third party libraries used by the Pub/Sub API impose restrictions.

- The ptypes library is initialized to ignore the SIGPIPE signal.
- The xmlrpc library establishes its own signal handler for SIGCHLD.

Pub/Sub C++ clients should not attempt use these signals. Attempting to use these signals may interfere with the assumptions made by these libraries, resulting in incorrect behavior.

3.2. Subscribing to the Sybase Aleri Streaming Platform Using C++

3.2.1. Set Up Objects for SP Subscription in C++

The first thing the client application must do is initialize the `SpFactory` class and instantiate the `SpPlatform` object.

3.2.1.1. Create an SpPlatform Object

After the `SpFactory` class is initialized, the client application calls on this object to instantiate other objects that provide the Sybase Aleri Streaming Platform functionality.

The next object to instantiate is the `SpPlatform` object.

The following code example shows how to use the `SpFactory` class to initialize the factory and environment and then to create the `SpPlatform` object.

1. The `SpFactory` class must be initialized before the client application starts any threads. The static function `init()` does this as follows:

```
int rc = SpFactory::init();
```

The `SpFactory::init()` method sets up the XMLRPC environment. The XMLRPC implementation used by the Pub/Sub C++ API requires that the environment be initialized prior to starting up any threads. Therefore, make sure the `SpFactory::init()` method is called immediately when the client application first starts up. Later, usually upon exiting the program, a matching `SpFactory::dispose()` method should be called to “tear-down” the XMLRPC environment.

If the return from this call is a non-zero error code, the initialization has failed. If this happens, the client application can call `SpUtils::getErrorMessage(int errorCode)` to get the text describing the error.

2. Once the `SpFactory` has been initialized, the `SpFactory::createPlatformParms` method should be called. This factory method creates an object that encapsulates all of the Sybase Aleri Streaming Platform connection information, including the Sybase Aleri Streaming Platform host name, port number, username, password, and encryption usage.

See the set of overloaded `SpFactory::createPlatformParms(...)` methods for the set of available connection/authentication options. They include RSA authentication, as well as the High Availability configuration option.

Example:

```
SpPlatformParms *parms = SpFactory::createPlatformParms(  
    host, port, user, password, isEncrypted);
```

3. Use `SpFactory::createSpPlatformStatus()` to create an object for subsequent `SpFactory` method calls. The purpose of this object is communication of error status from the `SpFactory` class.

```
SpPlatformStatus *status = SpFactory::createPlatformStatus();
```

4. Call the `SpFactory` to create the `SpPlatform` object, passing in the `SpPlatformParms` and `SpPlatformStatus` objects previously created.

If the `SpFactory::createPlatform(parms, status)` call is successful, a fully initialized `SpPlatform` object is returned to the client programmer. Otherwise, the factory method will return `NULL`, and an error code will be stored in the `SpPlatformStatus` object that was passed into the `createPlatform(...)` method. The `SpPlatformStatus` object can be used to retrieve the error code, using `status->getErrorCode()` or the error text using `status->getErrorMessage()`. Observe the "else" condition in the following code fragment:

```
SpPlatform *sp = SpFactory::createPlatform(parms, status);  
if (0 != sp) {  
    /* Use the new sp object for Platform related work */  
    /* See "The SpPlatform object" */  
} else {  
    printf("Could not create SpPlatform, eNum = %d, eMsg= %s\n",  
        status->getErrorCode(),  
        (char *)status->getErrorMessage().c_str());  
}
```

3.2.2. Setup and Start a Subscription in C++

To get stream updates delivered from the Sybase Aleri Streaming Platform to your client application, the program must "ask" the Sybase Aleri Streaming Platform to deliver them. This process is called "subscribing" or creating a subscription. The Pub/Sub API offers two forms of subscription mechanisms (`SpSubscription` and `SpSubscriptionProjection`) that hide most of the low-level details associated with making a subscription request to the Sybase Aleri Streaming Platform's Gateway Server module. If the Pub/Sub API is used in a High Availability (Hot Spare) configuration, the switch over to

the Hot Spare server is handled transparently by the API.

The Pub/Sub API subscription mechanism is based on the “Observer” Design Pattern. When using the Pub/Sub API subscription mechanism, the client application program is responsible for creating a class that implements the `SpObserver` interface. This is a simple interface with a `notify(std::vector<SpSubscriptionEvent *> *events)` method in it. The subscription mechanism calls the `notify` method to deliver stream update and system event information to the client application program's `SpObserver` object.

3.2.2.1. Initiate a Subscription Using `SpSubscriptionProjection`

The client application program must create its own `SpObserver` objects, which are notified by the `SpSubscriptionProjection` with the updates arriving from the Sybase Aleri Streaming Platform. The client application program creates `SpObserver` objects by implementing the `SpObserver` interface. Refer to [Section 3.2.2.2, “Implement the `SpObserver` Interface”](#) for more information. The `addObserver(SpObserver theObserver)` method is used to register the `SpObserver` with the `SpSubscriptionProjection` object. As mentioned previously, you would construct the `SpObserver` using the `SpStreamProjection` that was returned by the `SpSubscriptionProjection.getStreamProjection()` method.

The `addObserver(...)` call returns an integer value that represents a “handle” to the registered `SpObserver` object. Later on, the client application programmer can use the handle to remove the `SpObserver`.

The `removeObserver(int theHandle)` method is used to remove the `SpObserver` from the `SpSubscriptionProjection`'s delivery mechanism.

See [Section 5.1, “Aleri SQL Queries and Statements”](#) for some limitations related to the Sybase Aleri Streaming Platform's handling of SQL queries.

3.2.2.2. Implement the `SpObserver` Interface

The `SpObserver` interface must be implemented in the client application if it receive stream updates from the Sybase Aleri Streaming Platform through the Pub/Sub API subscription mechanism. The interface is simple:

```
std::string getName();  
  
void notify(std::vector<SpSubscriptionEvent *> *theEvents);
```

There are two methods that must be implemented within the class:

- The `getName()` method is placed in the interface to retrieve the *name* of the `SpObserver`, which is similar to the case of the `SpSubscription` objects `getName()` method.
- The `notify(std::vector<SpSubscriptionEvent *> *theEvents)` method is the link between the underlying Sybase Aleri Streaming Platform subscription, which the `SpSubscription/SpSubscriptionProjection` object manages, and the client applications object, which receives the subscription updates as they are delivered from the Sybase Aleri Streaming Platform.

When an update is delivered from the Sybase Aleri Streaming Platform to the `SpSubscription/SpSubscriptionProjection` object, that object determines which of the pre-registered `SpObserver` objects are interested in the data and forwards it to them. This data is then picked up by the `SpObserver` through the `notify(std::vector<SpSubscriptionEvent`

*> **theEvents*) implementation.

The subscription's stream update acquisition and delivery mechanism run in a separate thread which manages the “read-only” Gateway I/O subscription socket. The `notify(std::vector<SpSubscriptionEvent *> *theEvents)` methods actually execute from within the context of this thread. The client application programmer must be conscious of this fact and program accordingly.

3.2.2.3. Start the Subscription Using SpSubscription

The following is a portion of code that shows how to create, configure and start an `SpSubscription` object. In the following example, the `sp` object represents an `SpPlatform` object instantiated previously through the `SpFactory::createPlatform(...)` method.

```
SpSubscription *sub = sp->createSubscription("MySubscription_1",
    SpSubConst::BASE,
    SpSubConst::DELIVER_PARSED,
    status); // "status" is a pointer to an SpPlatformStatus object.
```

You must create a concrete class implementing the `SpObserver` interface. This concrete `SpObserver` class will be registered with the new `SpSubscription` object, and notified with `SpSubscriptionEvent` objects when the `SpSubscription` is started.

```
if (sub != NULL)
{
    SpObserver *spObserver=new ClientSpObserver("myObserver");
    constant char * streamName = "input";
    int handle;
```

You must associate the concrete `SpObserver` object(s) with a stream (or set of streams), and register the `SpObserver` with the `SpSubscription` object. This can be done using either of the following methods:

- `SpSubscription->addStreamObserver(...)`
- `SpSubscription->addStreamsObserver(...)`

This should be done for each `SpObserver` the client wishes to be notified with `SpSubscriptions` events.

```
handle = sub->addStreamObserver(streamName, spObserver);

if (handle <= 0)
{
    // SpObserver registration failed.
    return handle;
}
```

Once the `SpObserver(s)` are registered with the `SpSubscription`, the `SpSubscription->start()` method is called to start the subscription process. If it's successful, the appropriate `SpObservers` will be notified with updates sent from the Sybase Aleri Streaming Platform.

```
>
printf("Starting the subscription.\n");

rc = sub->start();

if (rc != 0)
{
    // The subscription could not be started.
    printf("Subscription could not be started, rc=%d\n",rc);

    printf("Error message =%s\n", SpUtils::getErrorMessage(rc));
    return rc;
}
```

At this point, if there is data for the specified stream located on the Sybase Aleri Streaming Platform, the registered `SpObserver` objects will start receiving updates within the running context of the subscription thread.

A subscription created using `SpSubscription` receives updates for all columns in the stream with each event. If the subscription was created using `SpSubscriptionProjection`, an update consists of only the subset of columns defined by the SQL query. These types of updates are issued only when there is a change in one of the columns specified in the SQL query.

3.2.2.4. Start the Subscription Using `SpSubscriptionProjection`

The following code example shows how to create, configure, start and stop an `SpSubscriptionProjection` object. In this example, the “sp” object represents an `SpPlatform` object instantiated previously through the `SpFactory::createPlatform(...)` method.

status is a pointer to an `SpPlatformStatus` object.

```
const char * sqlQuery
    "select intData, charData from inputstream where intData > 100";

SpSubscriptionProjection *subProj = sp->createSubscriptionProjection(
    "MySubscriptionProjection_1",
    SpSubConst::BASE,
    SpSubConst::DELIVER_PARSED,
    sqlQuery,
    status);

if (subProj != 0)
{
    SpStreamProjection *streamProj = subProj->getStreamProjection();

    SpObserver *spObserver = new ClientSpObserver("myObserver", streamProj);

    int handle = subProj->addObserver(spObserver);

    if (cookie <= 0)
    {
        // SpObserver registration failed.
        return handle;
    }

    printf("Starting the subscription.\n");

    rc = subProj->start();

    if (rc != 0)
    {
        // The subscription could not be started.
    }
}
```

```
    printf("Subscription could not be started, rc=%d\n",rc);

    println("Error message =%s\n",
        SpUtils::getErrorMessage(rc));
    return rc;
}
else
{
    // Could not create the subscription object.
    rc = status->getErrorCode();

    printf("Could not create subscription object, error=%d\n",rc);

    printf("Error message =%s\n",
        status->getErrorMessage());
}

printf("Stopping the subscription.\n");

rc = sub->stop();

if (rc != 0)
{
    // Problems stopping the subscription.
    printf("Problems stopping the subscription, rc=%d\n",rc);

    printf("Error message =%s\n",
        SpUtils::getErrorMessage(rc));
    return rc;
}
```

After a successful `createSubscriptionProjection()` call, the program gets back the schema information as a result of the projection.

The Client program must create a concrete class implementing the `SpObserver` interface. This concrete `SpObserver` class will be registered with the new `SpSubscriptionProjection` object, and “notified” with `SpSubscriptionEvent` objects when the `SpSubscriptionProjection` is started.

The stream projection schema information is passed into the observer so it will know how to process the update events.

See [Section 5.1, “Aleri SQL Queries and Statements”](#) for some limitations related to the Sybase Aleri Streaming Platform's handling of SQL queries.

The client application must register the concrete `SpObserver` object(s) with the `SpSubscriptionProjection` object, using the `addObserver(...)` method. This should be done for each `SpObserver` the client wants to receive subscription events.

Once the `SpObserver(s)` are registered with the `SpSubscriptionProjection`, the `SpSubscriptionProjection->start()` method is called, which starts the subscription process. If it's successful, the appropriate `SpObservers` will be notified with updates sent from the Sybase Aleri Streaming Platform.

At this point, if there is data for the specified stream located on the Sybase Aleri Streaming Platform, the registered `SpObserver` objects will start receiving updates within the running context of the subscription thread.

The client program can invoke the `stop()` method in order to terminate the running `SpSubscription`.

3.2.3. Receive/Process Subscription Updates Using C++

3.2.3.1. Delivery to an SpObserver Notify(...) Method Implementation

In a running subscription, a vector of SpSubscriptionEvent objects is delivered to the SpObserver. Each SpSubscriptionEvent object represents a state change to the SpSubscription/SpSubscriptionProjection object; the Subscription object distributes these SubscriptionEvents to the appropriate SpObserver objects.

For example, an SpSubscriptionEvent is sent to the SpObserver if a stream update has arrived, or if the Sybase Aleri Streaming Platform is shut down.

The notify(...) method that you implement in the SpObserver object must iterate over the vector of SpSubscriptionEvents (each one uniquely identified by an EventId) to determine the action to be taken.

3.2.3.2. Inspect the Subscription Parsing Errors within the SpObserver

There is a “special” event id defined within the SpSubscriptionEvent interface as *SpSubscriptionEvent::EVID_PARSING_ERROR*. This event is generated when the SpSubscription object's message parser encounters an error in the middle of parsing the message delivered from the Sybase Aleri Streaming Platform.

In this case, the partial results that are successfully parsed up to the point of error are stored for possible inspection by the SpObserver. The SpObserver is not obligated to look at them. However, if you want to use this information for debugging purposes, it is still stored in the event.

When this event ID is encountered, and there are partial results that were parsed before the error was encountered, the getData() method of the SpSubscriptionEvent passed to the SpObserver notify() method will return a vector of two elements:

- An object of type SpParserReturnInfo, which has parsing error information stored in it.
- Another SpSubscriptionEvent object whose event ID is *SpSubscriptionEvent.PARSED_PARTIAL_FIELD_DATA*. The getData() method of this event object returns a vector of the stream update information that was successfully parsed up to the point of error.

3.2.3.3. SHINE Flag Supports Subscription Mode For Partial-Record Updates

The SHINE flag can support a new subscription mode for partial-record updates in C++ with *SpSubConst::SHINE*

The following is an example of how to use this mode:

```
SpSubscription *sub = sp->createSubscription(subscriptionName,  
SpSubConst::BASE|SpSubConst::SHINE, SpSubConst::DELIVER_PARSED, spStatus);
```

3.2.4. Change/Stop Subscription Using C++

3.2.4.1. Stop Subscription

The client programmer may invoke the stop() method to terminate the running SpSubscription. Once the start() method has been invoked, the asynchronous arrival of events initiates a notify() method which runs on a separate thread. Be careful not to invoke the stop() method from the notify() callback.

```
printf("Stopping the subscription.\n");
rc = sub->stop();
if (rc != 0)
{
    // Problems stopping the subscription.
    printf ("Problems stopping the subscription, rc=%d\n",rc);

    printf("Error message =%s\n",
        SpUtils::getErrorMessage(rc));
    return rc;
}
```

Before stopping entirely, the client application should destroy all the Subscription objects it created, to avoid memory leaks.

3.3. Publishing to the Sybase Aleri Streaming Platform Using C++

The Pub/Sub API gives you the ability to publish stream information to the Sybase Aleri Streaming Platform. This is accomplished by using an object of type SpPublication.

Note:

Whether the client application is publishing static data (such as a reference table) or dynamic data (such as market feed data) to the Sybase Aleri Streaming Platform, the same mechanism is used.

3.3.1. Create Objects for Publication Using C++

3.3.1.1. Create an SpPublication Object

The first step in setting up a client application that publishes to the Sybase Aleri Streaming Platform using the Pub/Sub API is creating an SpPublication object. This is done by a factory method provided by the SpPlatform object that must have been instantiated previously as with the Pub/Sub API subscription mechanism. The signature of the SpPlatform factory method creates SpPublication objects as follows:

```
SpPublication *createPublication(const char * name,
    SpPlatformStatus *status);
```

Details:

- *const char *name* is an identifier that you intend to assign to the SpPublication object being created.
- *SpPlatformStatus *status* is an object that returns error information back from the `createPublication(...)` factory method if an error condition is detected during the creation of an SpPublication object.

The following example shows how to use an instance of SpPlatform named `sp` to create an SpPublication object:

```
SpPublication *pub =  
    sp->createPublication("MyPublication_1", status);
```

In the above example, *status* is a pointer to a *SpPlatformStatus* object that was created previously with the *SpFactory::createPlatformStatus()* factory method.

The *SpPublication* object is not re-entrant. If multiple threads are going to publish to the Streaming Processor, each thread should use a different *SpPublication* object. Each of these *SpPublication* objects should have its own socket connection to the Streaming Processor.

3.3.1.2. Create *SpStreamDataRecord* Objects

For consistency within the Pub/Sub API, an *SpStreamDataRecord* object is created using an *SpFactory* method with the following method signature:

```
SpStreamDataRecord *SpFactory::createStreamDataRecord  
(  
    SpStream *stream,  
    std::vector<SpDataValue> *fieldData,  
    opCode,  
    flags,  
    SpPlatformStatus *status  
);
```

Details:

- *SpStream *stream* refers to the *SpStream* object with which this new *SpStreamDataRecord* object will be associated. You can get this value through one of the *SpPlatform* methods, such as *getStream(const char *streamName)* or *getStream(int streamId)*.
- *std::vector<SpDataValue> *fieldData* is a pointer to a vector of pointers to *SpDataValue* objects. Each object entry in the vector matches the field data type indicated in the stream definition (specified in the *SpStream* parameter).

All of the *SpStreamDataRecord*'s key fields must be specified with non-null values within the *fieldData* vector. In addition, the types of the objects that are located in the *fieldData* vector must match those in the *SpStream* definition.

Your program can identify the key fields using the *getKeyColumns()* or *getKeyColumnVector()* functions, or by inspecting each column using *isKeyColumn()*.

- *int opCode* is the stream operation code that is associated with this *SpStreamDataRecord*.

The opcode tells the Sybase Aleri Streaming Platform how to apply this record to the source stream.

- *int flags* is the flag settings value that is associated with this *SpStreamDataRecord*. Refer to [Section B.2.3, "Stream Flag Values"](#) for more information about stream flag values.

Several of the publishing methods allow the client application program to override the stream opcode and/or stream flag settings.

- *SpPlatformStatus *status* is a pointer to an object that returns error information from the

`createStreamDataRecord(...)` factory method if the `SpStreamDataRecord` object cannot be created.

The following code example shows how the client application program can use the `createStreamDataRecord(...)` factory method to create an `SpStreamDataRecord` object that can be published to the Sybase Aleri Streaming Platform. Refer to the `pubexample.cpp` file to see how to build a record set that can be published to the Sybase Aleri Streaming Platform as a transaction.

```
/*
 * Build up a row, for a source stream called "input", that
 * has the following record layout:
 *
 * int, long, string, double, date, money, timestamp
 */

// Build the field data list for the stream row.
std::vector<SpDataValue *> *fieldData =
    new std::vector<SpDataValue *>;

/* build an int32 field */
SpDataValue *ptrDataValue = new SpDataValue();
ptrDataValue->dataValue.val.int32v = 100;
ptrDataValue->type = DataTypes::INT32;
ptrDataValue->dataValue.null = false;
fieldData->push_back(ptrDataValue);

/* build an int64 field */
ptrDataValue = new SpDataValue();
ptrDataValue->dataValue.val.int64v = 1001;
ptrDataValue->type = DataTypes::INT64;
ptrDataValue->dataValue.null = false;
fieldData->push_back(ptrDataValue);

/* build a string field */
ptrDataValue = new SpDataValue();
char *theString = new char [20];
strcpy(theString, "hello");
ptrDataValue->dataValue.val.stringv = theString;
ptrDataValue->type = DataTypes::STRING;
ptrDataValue->dataValue.null = false;
fieldData->push_back(ptrDataValue);

/* build a double field */
ptrDataValue = new SpDataValue();
ptrDataValue->dataValue.val.doublev = 3.14;
ptrDataValue->type = DataTypes::DOUBLE;
ptrDataValue->dataValue.null = false;
fieldData->push_back(ptrDataValue);

/* build a date field */
time_t timeData = time(0);
ptrDataValue = new SpDataValue();
ptrDataValue->dataValue.val.datev = timeData;
ptrDataValue->type = DataTypes::DATE;
ptrDataValue->dataValue.null = false;
fieldData->push_back(ptrDataValue);

/* build a money field
 * NOTE: Money has a scale value of
 * n = platform->getMoneyPrecision() decimal places
 * where 'platform' is a pointer to an SpPlatform object
 * Below, you "know" the scale factor is the default (10,000)
 * so you represent 1000 with the value 1000 * 10,000 = 10,000,000.
 */
DataTypes::money_t moneyData = 10000000;
ptrDataValue = new SpDataValue();
```

```

ptrDataValue->dataValue.val.moneyv = moneyData;
ptrDataValue->type = DataTypes::MONEY;
ptrDataValue->dataValue.null = false;
fieldData->push_back(ptrDataValue);

/* build a timestamp field
 * NOTE: The Timestamp datatype
 * is basically the same as a Date datatype except that it is capable of
 * holding milliseconds.
 */
DataTypes::timestampval_t timeStampData =
    (DataTypes::timestampval_t) time(0)* 1000;
ptrDataValue = new SpDataValue();
ptrDataValue->dataValue.val.timestampv = timeStampData;
ptrDataValue->type = DataTypes::TIMESTAMP;
dataValue.null = false;
fieldData->push_back(ptrDataValue);

// You need to have a pointer to the stream for the
// row that is building up.
SpStream *stream = sp->getStream("input");

/*
 * Use the createStreamDataRecord(...) factory method to
 * bundle up the stream, fieldData vector, stream opcode,
 * and stream flags into an SpStreamDataRecord object.
 *
 * The SpStreamDataRecord object is the basic
 * unit of publication. You can publish these one at a
 * time, or you can publish them as a group (with or
 * without transaction blocks).
 *
 * If you want to publish a group of SpStreamData-
 * Record objects as a transaction, then all of the
 * SpStreamDataRecords within the group must belong
 * to the same stream.
 */
SpStreamDataRecord *sdr = SpFactory::createStreamDataRecord(
    stream,
    fieldData,
    StreamInterface::UPSERT,
    StreamInterface::NULLFLAG,
    status
);

if (0 == sdr)
{
    printf("Could not createStreamDataRecord, error code=%d\n",
        status->getErrorCode());
    printf("Error Message = %s\n",
        status->getErrorMessage());
}

```

The client application program can create a large number of `SpStreamDataRecord` objects, placing each of them within a common vector. Next, the application programmer can use one of the `SpPublication`'s publishing methods to send all the vector's stream data objects to the Sybase Aleri Streaming Platform, either individually or by using transactions.

3.3.2. Publish Data to the Sybase Aleri Streaming Platform Using C++

The following example shows how to publish a vector of `SpStreamDataRecord` objects as a single transaction. In this example, `sp` is an `SpPlatform` object that was previously instantiated and `streamInputData` is a vector that contains a large number of `SpStreamDataRecord` objects.

```

>
/*

```

```
* Create the publication object associated with the
* platform.
*/
const char *name = "testPub_1";

SpPublication *pub = sp->createPublication(name, status);

if (0 == pub)
{
    printf("Couldn't create a publication object, error code =%d\n",
        status->getErrorCode());
    printf("Error message = %s\n", status->getErrorMessage());
    return status->getErrorCode();
}

/*
* Start the publication object (this opens up a GW I/O
* socket connection). Don't forget to eventually close
* down the SpSubscription object (via the stop() method,
* later on when you are finished using it,
*/
rc = pub->start();

if (0 != rc)
{
    printf("Couldn't start the publication object.\n");
    printf("Error message = %s\n",
        SpUtils::getErrorMessage(rc));
    return rc;
}

/*
* Publish the collection/vector of SpStreamDataRecord
* objects as one big transaction.
*/

rc = pub->publishTransaction(streamInputData,
    StreamInterface::INSERT,
    StreamInterface::NULLFLAG,
    0);

if (0 != rc)
{
    printf("Couldn't publish the transaction.\n");
    printf("Error message = %s\n",
        SpUtils::getErrorMessage(rc));
    return rc;
}
```

3.3.3. Handling Stale Data

When a publishing source stops sending data to the Sybase Aleri Streaming Platform, the previously published data is retained. Depending on how long it has been since the last update, you may not want this data to be used as if it were current. The publish/subscribe APIs include two functions to enable publishers to handle this data.

The “setFinalizer” function sets a timeout value (in milliseconds) and an SQL statement action. If the Sybase Aleri Streaming Platform receives no data on this connection within the specified time, the SQL statement is run. This SQL statement can perform any of the following actions:

- Delete previously published data.
- Mark previously published data as stale (via a field for that purpose in the data).
- Perform some other determined action on the source streams (and, consequently, the derived streams from these source streams).

Note:

When using `setFinalizer()`, you must ensure that the `SpStatus` object created by `SpFactory::createStatus()` is still in scope when the `SpPublication->start()` function is executed, since any errors are returned via the `SpStatus` object. Failure to do so can result in memory corruption and/or other undefined behavior.

In the following example, if the data is not updated within 1000 milliseconds, it is deleted.

```
setFinalizer(1000, "delete from positions where SharesHeld > 1", spStatus)
```

The “sendHeartbeat” function sends a keep-alive message to the Sybase Aleri Streaming Platform. This function can be used to keep the connection alive and prevent the SQL statement from running, if “setFinalizer” has previously been called. The “sendHeartbeat” function takes no arguments; its syntax is:

```
sendHeartbeat()
```

3.3.4. Set/Get Methods for Exit-on-Drop, Exit-on-Timeout Capability to SpPublication Using C++

Set methods should be called before `SpPublication start()` method. `SpPlatformStatus` should be checked after the `start()` for any possible problem. If you fail to send exit-on-close, the status is set to `SP_ERROR_SETTING_EXIT`. If you fail to send exit-on-timeout, the status is set to `SP_ERROR_PUB_ERROR_SETTING_EXIT`.

If `setFinalizer` and `setExitOnTimeout` are called, the second call returns `SP_ERROR_PUB_EXIT_ALREADYSET` (`setFinalizer`) or `SP_ERROR_PUB_ACTION_ALREADYSET` (`setExitOnTimeout`).

The following is an example using C++.

```
void setExitOnTimeout(const long timeout, SpPlatformStatus * status);  
int getExitOnTimeout();  
void setExitOnClose(SpPlatformStatus * status);  
bool getExitOnClose();
```

3.4. Record/Playback using C++

In order to record data from the Sybase Aleri Streaming Platform, a client program needs to create and configure an `SpPlatform` object in the same way you would for subscribing or publishing. Once an `SpPlatform` object has been created, the programmer should create an `SpRecorder` object using the fact-

ory method `createRecorder(...)`. If an `SpRecorder` is successfully created the program calls the `start` method. This spawns a background thread which subscribes to the configured streams and records all events for those streams. Recording will stop and the spawned thread terminate once the configured number of records have been processed or if the calling program calls `stop()`. Recording can be monitored by calling the `getRecordCount()` function which returns the number of records processed so far.

```
// Initialize SpFactory ...
// Create SpStatus and SpParms objects ...
// Create platform object ...

// Create recorder. It needs the following parameters to run
// recorder name (string) : a name to identify the instance of the recording object
// recorder file (string) : name of the file to store the recorded information
// streams                : vector of strings, containing names of streams to record events for
// flags (int)            : recording options (encrypted/RSA/get base data)
// max records (int64_t)  : maximum number of data records to record
// status                 : returns error messages if any

// init recorder parameters - recorder name, filename, streams, etc
SpRecorder * recorder = spPlatform->createRecorder(recName, recFile, streams, flags, maxRecords, status);

if (NULL == recorder) {
    std::cout << "Error starting recorder - " << status->getErrorMessage() << std::endl;
    // cleanup ... and exit
} else
    recorder->start();

// Wait, monitor, etc ...

// To stop recording
recorder->stop();

delete recorder;

// Cleanup
```

A client program creates an `SpPlayback` object using the factory method in `SpPlatform` to play back recorded data. The 'scale' parameter is of particular interest. This is a double that can be used to scale the rate of playback as a factor of the original recorded rate (for example, twice as fast or half as slow). Values -1 to 1 have no effect - data is played back at the rate it was recorded. A value greater than 1 speeds up playback by that factor (for example, a value of 2 doubles the playback speed). A value less than -1 slows down playback by that factor (for example, a value of -3 will slow down playback by a factor of 3). The scale can be changed dynamically while playback is in progress.

```
>
// Initialize SpFactory ...
// Create SpStatus and SpParms objects ...
// Create platform object ...

// Create playback. It needs the following parameters to run
// playback name (string) : a name to identify the instance of the playback object
// playback file (string) : name of the file containing previously recorded data
// scale (double)         : allows to scale the playback rate
// max records (int64_t)  : maximum number of data records to playback
// status                 : returns error messages if any

// init playback parameters - recorder name, filename, streams, etc.
SpPlayback * playback = spPlatform->createPlayback(playName, playFile, scale, maxRecords, status);

if (NULL == playback) {
    std::cout << "Error starting recorder - " << status->getErrorMessage();
    // cleanup ... and exit
} else {
    playback->setSendUpsert(TRUE); // optionally enable converting opcodes to UPSERT
    playback->start();
}

// Wait, monitor, etc ...

// To stop recording
```

```
playback->stop();  
delete playback;
```

3.5. Special Topics for SP Publication/Subscription Using C++

3.5.1. Publication/Subscription In a High Availability (Hot Spare) Configuration

The Sybase Aleri Streaming Platform can be started in a High Availability configuration, with one server as the primary server and the other considered the Hot Spare (secondary server). If the Pub/Sub API is made aware of the High Availability configuration, it will perform an automatic switchover to the secondary server if the primary server becomes unreachable. See [Section 2.4.6, “Publication/Subscription in a High Availability \(Hot Spare\) Configuration”](#) for more information about High Availability mode.

The Pub/Sub API is made aware of the High Availability configuration through the `SpPlatformParms` object that was passed into `SpFactory::createPlatform()`. Refer to the set of overloaded `SpFactory::createPlatformParms(...)` methods, and the `SpPlatformParms` object for details about the High Availability configuration.

3.5.1.1. Subscription Mechanisms in a High Availability Configuration

When the Sybase Aleri Streaming Platform is brought up in High Availability mode, and the Pub/Sub API makes use of this configuration, the following occurs when the primary server goes down:

First, the underlying subscription thread receives an exception on the Gateway I/O socket connection used to receive the stream updates delivered from the primary server. When this event occurs, the Pub/Sub API recognizes that there is a High Availability configuration and attempts to connect to the secondary server and re-establish the subscription. Before it does this, the Pub/Sub API has to wait for the secondary server to internally change its state to that of the primary server.

Once a successful connection is made to the secondary server and it has been promoted to the primary server, the subscriptions are re-established. During the switchover to the secondary server, the Subscription object delivers several events to the `SpObserver` objects listening for `SpSubscriptionEvents`. The following events are typically delivered between the time the socket is dropped and the time the connection is made to the secondary server:

- `SpSubscriptionEvent::EVID_COMMUNICATOR_HALTED`

It is delivered to the `SpObserver` when the exception is received on the socket, receiving the subscription messages from the primary Streaming Processor.

- `SpSubscriptionEvent::EVID_HOT_SPARE_SWITCH_OVER_INITIATED`

It is delivered to the `SpObserver` when the Pub/Sub API recognizes that a connection attempt should be made to the Hot Spare server. Note that the High Availability connection parameters were specified in the `SpPlatformParms` object passed to the `SpFactory::createPlatform()` method when the underlying `SpPlatform` was first created.

- `SpSubscriptionEvent::EVID_HOT_SPARE_SWITCH_OVER_SUCCEEDED`

It is delivered to the `SpObserver` when the connection to the Hot Spare server is made.

- `SpSubscriptionEvent::EVID_HOT_SPARE_SWITCH_OVER_FAILED`

It is delivered to the `SpObserver` when the attempted connection to the Hot Spare server fails.

If the switchover to the Hot Spare server is successful, the subscription(s) are re-established using the

same delivery flag values that were originally used when the subscription(s) against the primary server. It means that if the subscription originally requested the BASE snapshot of the stream, the new subscription (now going against the Hot Spare server) requests the BASE snapshot of the stream. You need to determine what should be done with the contents of the snapshot received from the Hot Spare server.

When there is a successful switchover to the Hot Spare server, the `SpPlatform` object takes note and performs some internal book keeping to ensure that the `SpPlatform::getHost()` and `SpPlatform::getPort()` methods will return the host name and port number of the new primary server.

3.5.1.2. Publication Mechanisms in a High Availability Configuration

When the primary server goes down during an attempt to send a publication request to the server, the `SpPublication` object detects it and attempts to perform a switch over to the Hot Spare server. If the switchover is successful, the publication object then attempts to re-send the data to the new primary server. If the publication cannot take place, a non-zero error code is returned to the caller indicating the problem.

The Pub/Sub API programmer should treat the secondary server within a High Availability configuration as a passive server. The client application should never log on and send data to a secondary server while the primary server is alive and well. It is the responsibility of the running High Availability configuration to manage both the primary and secondary servers appropriately. If the primary server goes down, the secondary server will take over and become the new primary server. Once the secondary server becomes the primary server, data can be published to the new primary server. The Pub/Sub API waits for the secondary server state to switch over to primary before publishing data.

The sync point between data sent to source streams of the primary server and this data being propagated to the Hot Spare server is manual. The client application can accomplish this synchronization by calling the `commit()` method of the `SpPublication` class.

Chapter 4. Publish/Subscribe API for .NET 2.0

4.1. Overview/General Information

This chapter describes how to use .NET 2.0 client applications that communicate with the Sybase Aleri Streaming Platform. The Pub/Sub API is delivered in the `pubsubnet.dll` file.

Building client applications with Sybase's .NET Pub/Sub API requires a third party tool: Microsoft Visual C++ compiler 2005.

4.1.1. Overview of .NET Utilities for SP Publication/Subscription

The .NET API provides a set of high-level interfaces for developers to write Microsoft .NET applications that interact directly with the Sybase Aleri Streaming Platform. The interfaces hide most of the underlying implementation details and provide a set of intuitive classes (`SpFactory`, `SpPlatform`, `SpSubscription`, `SpPublication`, and so on). Most of the classes within the Pub/Sub API start off with the prefix “Sp”, which stands for “Streaming Processor”. All the “Sp” classes reside in the namespace called `aleri_PubSubnet`. There are also some “constant” definitions that are used by various API method calls that are located in the `aleri.pubsubconst` namespace.

4.1.1.1. API Library

The `pubsubnet.dll` shared library contains the .NET 2.0 Pub/Sub API used to publish and subscribe against the Sybase Aleri Streaming Platform. It is located in the distribution directory under `.\lib`.

The Pub/Sub library includes code for the various authentication mechanisms supported by Sybase Aleri Streaming Platform, including Kerberos. It requires that the SASL dynamic libraries shipped with the Sybase Aleri Streaming Platform be present at runtime. These libraries are located in the `$PLATFORM_HOME/lib` folder.

4.1.1.2. Example Files

Here are three C# examples, with one each for publish, subscribe, and subscribe with projection.

- Publication Example

The publish example is located in the install directory under `.\examples\clients\pubsub\net\PubExample`. This folder contains the following files:

- `PubExample.cs` (which demonstrates publication to the Sybase Aleri Streaming Platform)
- `PubExample.csproj` (Visual Studio 2005 C# Project file)
- `PubExample.sln` (Visual Studio 2005 Solution File)
- `PubExample.exe` (A precompiled version of the code included for convenience.)

- Subscribe Examples

The Subscribe example is located in the install directory under `.\examples\clients\pubsub\net\SubExample`. This folder contains:

- `SubExample.cs` (demonstrates subscription to the Sybase Aleri Streaming Platform)
- `SubExampleSpObserver.cs` (demonstrates subscription to the Sybase Aleri Streaming Platform)

- `SubExample.csproj` (Visual Studio 2005 Project file)
- `SubExample.sln` (Visual Studio 2005 Solution File)
- `SubExample.exe` (A precompiled version of the code included for convenience.)

The Subscribe with projection example is located in the install directory under `.\examples\clients\pubsub\net\SubProjExample`. This folder contains:

- `SubProjExample.cs` (demonstrates subscription to the Sybase Aleri Streaming Platform)
- `SubProjExampleSpObserver.cs` (demonstrates subscription to the Sybase Aleri Streaming Platform)
- `SubProjExample.csproj` (Visual Studio 2005 Project file)
- `SubProjExample.sln` (Visual Studio 2005 Solution File)
- `SubProjExample.exe` (A precompiled version of the code included for convenience.)

Note:

In order to compile these examples, Visual Studio 8.0 has to be installed. To compile an example using the IDE open, the appropriate project file (`.csproj` file) in the IDE and build the example from there.

In addition there is Makefile provided in the `.\examples\clients\pubsub\net` folder of the install directory. This file can be used to build the three examples provided from the command line using the `nmake` command. Visual Studio 8.0 needs to be installed with the **nmake** and **devenv.com** executables in the path. To compile the examples, simply type the command **nmake** in the folder containing this Makefile.

4.1.2. Design Decisions for SP Publication/Subscription Using .NET 2.0

The Pub/Sub API provides a set of interfaces or object “types” that exposes all of the Sybase Aleri Streaming Platform functionality while hiding the implementation details. You are strongly encouraged to use the Pub/Sub API's implementation classes without using inheritance to extend it. This preserves Sybase's ability to change underlying implementation in the future without breaking client code.

To achieve these encapsulation goals, the “Factory” Design Pattern is used. Most of the implementation classes found in the Pub/Sub API have private constructors, and you are provided with “Factory” methods for object instantiation. For example, the code that instantiates a new `SpPlatform` object should be `SpFactory::createPlatform(...)`, not `new SpPlatform(...)`, and so forth.

In addition, most objects in the Pub/Sub API provide only “get” methods, to preserve some degree of immutability. Complete immutability would require all “get” methods to return “copies” of internal vectors/arrays because vectors and arrays are mutable. However, this method would be inefficient. Sybase chose a design that does not make and return “copies” to the caller.

You must maintain order and integrity for the state of these “read-only” data structures. For example, when requesting the list of column types for a stream definition, the program must not modify elements within the list. If it does, the application will encounter difficulties when it makes subsequent calls to retrieve the column types, which are now out of sync with those on the Sybase Aleri Streaming Platform.

The design of the Pub/Sub API also reflects the decision to avoid the use of exceptions generated by API routines. Each API method usually returns a non-zero error code if the method fails. Otherwise, the

method returns a zero to indicate that it did not detect an error. This limitation keeps the API consistent across the set of different languages in which it is implemented.

When a Pub/Sub API method returns a non-zero return code, the program can call the `theSpUtils.getErrorMessage(errorCode)` utility method to get the specific error message text.

Those API methods which take an `SpStatus` parameter may use `SpStatus::getErrorCode()` and/or `SpStatus::getErrorMessage()` to the same effect.

When one of the Pub/Sub API methods returns a non-zero error return code, the client application can call the `SpUtils.getErrorMessage(errorCode)` utility method to get the specific error message.

The `SpFactory` call that creates an object must pass in an `SpPlatformStatus` object. If the “create” method can not create the object that you requested, the method returns “null”, and sends the `SpPlatformStatus` object an error code identifying the problem. The `SpPlatformStatus` has a `getErrorMessage` method that will return the error text associated with the error message. All non-`SpFactory` method calls simply return a non-zero error code if an error occurs; as mentioned above, the `SpUtils.getErrorMessage(errorCode)` method can be called to get the associated error message text.

4.1.3. Set/Get Methods For Maximum Buffer Size, Exit-On-Drop To `SpSubscription` Using .NET

Set methods should be called before `SpSubscription` `start()` method. You should check `SpPlatformStatus` after the `start()` for any possible problem. If you fail to send exit-on-close, the status is set to `SP_ERROR_SETTING_EXIT`. If you fail to send a maximum buffer size, the status is set to `SP_ERROR_SUB_SETTING_BUFFERSIZE`. You should also note that `getQueueSize()` does NOT return the current queue size until after you've set it with `setQueueSize()`.

Here is an example for .NET 2.0.

```
void SetQueueSize(int queue, SpPlatformStatus ^status);  
  
    void setExitOnClose(SpPlatformStatus ^status);  
  
        int getQueueSize();  
  
        bool getExitOnClose();
```

4.2. Subscribing to the Sybase Aleri Streaming Platform Using .NET 2.0

4.2.1. Set Up the Environment for Subscription Using .NET 2.0

The first step in this process is to create objects that handle the low-level details of the subscription.

4.2.1.1. Configure the Pub/Sub API .NET 2.0 Pub/Subnet.dll

To use the Pub/Sub API, the .NET 2.0 application must be able to reference the `pubsubnet.dll`. Once this is done, all of the “Sp” related interface/class definitions can be seen within the IDE.

The installation does not add the Pub/Sub library to the GAC automatically. It must be added to the GAC using the `gacutil.exe` utility provided as part of the .NET framework.

All exposed classes are sealed and cannot be extended through inheritance. Always use the `SpFactory` to create the “Sp” Sybase Aleri Streaming Platform related objects.

4.2.1.2. Initialize the SpFactory Object

The first object to use in setting up a subscription is the static `SpFactory` object. Because it is “static”, it needs no instantiation. The `SpFactory` is used to instantiate the other objects that offer the Sybase Aleri Streaming Platform functionality. The first call made to the `SpFactory` is an initialization method:

```
int SpFactory.init();
```

This call initializes the underlying xmlrpc mechanism that is required to communicate with the Sybase Aleri Streaming Platform.

The xmlrpc documentation indicates that the xmlrpc mechanism must be initialized while the client application is still “single” threaded. Therefore, make sure the

```
int SpFactory.init();
```

call is made from within the application's main thread before starting subsequent threads.

When the application is finished and is about to exit, call the

```
SpFactory.dispose();
```

method. This cleans up the underlying xmlrpc mechanism.

The

```
int SpFactory.init();
```

method returns a non-zero error code if it encounters a problem, otherwise it returns zero.

4.2.1.3. Create the SpPlatform Object

Once the static `SpFactory` object is initialized successfully, the client application must instantiate the `SpPlatform` object.

The following example shows you how to use the `SpFactory` to create the `SpPlatform` object.

```
/*
 * First, using the SpFactory, create an SpPlatformParms
 * object that contains all of the Sybase Aleri Streaming Platform connection
 * information. This information consists of the Sybase Aleri Streaming Platform
 * host name, port number, username, password, and a boolean
 * flag indicating whether or not all connections to the
 * Sybase Aleri Streaming Platform will use encryption.
 * NOTE: See the set of overloaded SpFactory.createPlatformParms(...)
 * methods for the set of available connection/authentication options.
 * They include RSA authentication, as well as the Sybase Aleri Streaming Platform's High
 * Availability configuration option.
 */

SpPlatformParms parms = SpFactory.createPlatformParms(host,
    port, user, password, isEncrypted);

/*
 * Second, using the SpFactory, create an SpPlatformStatus
 * object. This object is used by the SpFactory to
```

```
* return error information if the SpFactory cannot create
* the SpPlatform object.
*/

SpPlatformStatus status=SpFactory.createPlatformStatus();

/*
* Third, using the SpFactory, create the SpPlatform object,
* passing in the SpPlatformParms and SpPlatformStatus
* objects created previously.
* NOTE: If the call is successful, the client programmer
* will be returned as a fully initialized SpPlatform object.
*
* Otherwise, the factory method returns null, and the
* error code will be stored in the SpPlatformStatus object
* that was passed into the createPlatform(...) method. You
* can use the SpPlatformStatus object to retrieve the error
* code, and the corresponding error message (see the "else"
* condition in the following code fragment).
*/

SpPlatform sp = SpFactory.createPlatform(parms, status);

if (sp != null)
{
    /*Use the new sp object to perform Sybase Aleri Streaming Platform related work*/
    /* See "The SpPlatform object" */
} else {
    Console.WriteLine("Could not create SpPlatform, error = " +
        status.getErrorCode() + ", error msg = " +
        status.getErrorMessage());
}
```

It is very important to ensure that the three objects created above -- SpPlatform, SpPlatformStatus and SpPlatformParms -- are always in scope. The following three statements must be added to the end of the code (before the return statement), which is always in scope during the lifetime of the publication process:

```
GC.KeepAlive(parms);
GC.KeepAlive(status);
GC.KeepAlive(sp);
```

Without these three statements, there is a risk that the garbage collector will clean up these objects before the publication is complete, resulting in unpredictable results and program crashes.

4.2.2. Set Up/Start Subscription Using .NET 2.0

To get stream updates delivered from the Sybase Aleri Streaming Platform, the client application must "ask" the Sybase Aleri Streaming Platform to deliver them. This process is called "subscribing" or creating a subscription. The Pub/Sub API offers two forms of subscription mechanisms, (SpSubscription and SpSubscriptionProjection), that hide most of the low-level details associated with making a subscription request to the Gateway I/O process. If the Pub/Sub API is used within the context of an Sybase Aleri Streaming Platform High Availability (Hot Spare) configuration, the switchover to the Hot Spare server is transparently handled within the API.

The Pub/Sub API subscription mechanism is based on the "Observer" Design Pattern. To use the Pub/Sub API subscription mechanism, you must create a class that implements the aleri_PubSubnet::SpObserver interface. This interface requires a notify(SpSubscriptionEvent[] events) method, which the subscription mechanism calls to deliver stream update information and system event information to the client application's SpObserver object.

The client application must also create a SpSubscription or SpSubscriptionProjection object using the appropriate factory method provided by the SpPlatform object instantiated previ-

ously. The `SpPlatform` factory methods used to create `SpSubscription` and `SpSubscriptionProjection` objects have the following signatures:

```
alери_PubSubnet::SpSubscription ^createSubscription(  
    System::String ^name, int flags, int deliveryType,  
    alери_PubSubnet::SpPlatformStatus ^status);  
  
alери_PubSubnet::SpSubscriptionProjection ^createSubscriptionProjection(  
    System::String ^name,  
    int flags, int deliveryType,  
    System::String ^sqlQuery,  
    alери_PubSubnet::SpPlatformStatus ^status);
```

Details:

String name: This is the name that the client application program intends to assign to the `SpSubscription` or `SpSubscriptionProjection` object being created.

int flags: This integer encapsulates the “flag bits” sent to the Sybase Aleri Streaming Platform Gateway I/O process when the low-level subscription request is made. The flag settings control delivery from the Sybase Aleri Streaming Platform to the client application, on the Gateway I/O socket connection where the subscription request was made. The “flag bits” are defined as constants in the `alери_PubSubconst::SpSubFlags` enumeration as follows:

- `alери_PubSubconst.SpSubFlags.BASE = 0x0;`

The `BASE` flag bit tells the Sybase Aleri Streaming Platform to send a complete “snapshot” of each stream of the subscription request before sending deltas. The complete “snapshot” or “state” of the stream is a set of “insert” records sent from the Sybase Aleri Streaming Platform between the `EVID_GATEWAY_SYNC_START` and `EVID_GATEWAY_SYNC_END` subscription events.

- `alери_PubSubconst.SpSubFlags.LOSSY = 0x1;`

The `LOSSY` flag bit puts the Sybase Aleri Streaming Platform in “data shedding mode”, in which the Sybase Aleri Streaming Platform drops the oldest data if the client cannot keep pace with the data it is receiving.

- `alери_PubSubconst.SpSubFlags.NO_BASE = 0x2;`

The `NOBASE` flag bit tells the Sybase Aleri Streaming Platform that it should NOT send a complete “snapshot” of the streams in the subscription request. The Sybase Aleri Streaming Platform that receives this flag will only send the deltas for each of the streams being subscribed to.

- `alери_PubSubconst.SpSubFlags.DROPPABLE = 0x8;`

The `NOBASE` flag tells the Sybase Aleri Streaming Platform that it should NOT send a complete “snapshot” of the streams in the subscription request. The Sybase Aleri Streaming Platform that receives this flag will only send the deltas for each of the subscribed to streams.

- `alери_PubSubconst.SpSubFlags.PRESERVE_BLOCKS = 0x20;`

The `PRESERVE_BLOCKS` flag bit tells the Sybase Aleri Streaming Platform that it should preserve blocks while sending data to the client application.

These flag bits can be ORed together using the “|” operator. For example:

```
flags = alери_PubSubconst.SpSubFlags.NO_BASE | alери_PubSubconst.SpSubFlags.LOSSY.
```

int deliveryType: This integer value specifies how the client application program's SpObserver object receives the stream update events. The delivery types are defined in the `aleri_pubsubconst.DeliveryType` as follows:

- `aleri_PubSubconst.DeliveryType.DELIVER_PARSED = 1;`

This delivery type setting tells the SpSubscription object to deliver "parsed" field data objects representing the stream update to your SpObserver object.

- `aleri_PubSubconst.DeliveryType.DELIVER_BINARY = 3;`

This delivery type setting tells the SpSubscription object to deliver the "binary" representation of the stream update record to your SpObserver object.

- `aleri_PubSubconst.DeliveryType.DELIVER_STREAM_OPCODES = 5;`

This delivery type setting tells the SpSubscription object not to use field level data, but to simply deliver the stream update operation code (*INSERT*, *UPDATE*, *DELETE*, or *UPSERT*).

System::String ^sqlQuery: specifies the SQL query projection on which the SpSubscriptionProjection will be based. The *sqlQuery* parameter can only be used to create an SpSubscriptionProjection object.

SpPlatformStatus status: can only be used to return error code information back from the `createSubscription(...)` and `createSubscriptionProjection(...)` factory methods, in the case where the SpSubscription or SpSubscriptionProjection object could not be created.

The following example shows how to use the SpPlatform object called *sp* to create both an SpSubscription and an SpSubscriptionProjection object:

```
SpSubscription sub = sp.createSubscription(
    "MySubscription_1",
    SpSubFlags.BASE,
    SpDeliveryType.DELIVER_PARSED,
    status);

SpSubscriptionProjection subProj = sp.createSubscription(
    "MySubscriptionProjection_2",
    SpSubFlags.BASE,
    SpDeliveryType.DELIVER_PARSED,
    "select intData, charData from inputstream where intData > 100",
    status);
```

In the above example, *status* is an SpPlatformStatus object that was created previously with the `SpFactory.createPlatformStatus()` factory method.

It is important that the SpSubscription or SpSubscriptionProjection object created above is always in scope. To ensure this, add the following line of code to the end of the code (before the return statement), that is always in scope during the lifetime of the subscription process:

```
GC.KeepAlive(sub);
```

>

Without this statement, you run the risk that the garbage collector will prematurely clean up the subscription object before the subscription is complete, leading to unpredictable results and program crashes.

4.2.2.1. Initiate a Subscription Using SpSubscription in .NET 2.0

If the `sp.createSubscription(...)` call is successful, the client application program gets back a `SpSubscription` object. The `SpSubscription` object can be used to subscribe to one or more streams, while an `SpSubscriptionProjection` object can only be used to subscribe to the projection defined by the `sqlQuery` passed into the `createSubscriptionProjection(...)` factory method. For each stream that is being observed, the `SpSubscription` object will deliver to the `SpObserver` stream events that contain all the stream's fields. `SpSubscription` extends the method set defined in the `SpSubscriptionCommon` interface as follows:

```
int /*Cookie*/ addStreamObserver(System::String ^streamName,
    aleri_PubSubnet::SpObserver ^theObserver);

int /*Cookie*/ addStreamsObserver(
    cli::array<System::String ^> ^theStreamNames,
    aleri_PubSubnet::SpObserver ^theObserver);

int subscribe(System::String ^streamName);

int unsubscribe(System::String ^streamName);
```

The following code example shows how to create, configure, start, and stop an `SpSubscription` object. In the following example, the `Sp` object represents a `SpPlatform` object instantiated previously through the `SpFactory.createPlatform(...)` method.

```
SpSubscription sub =
    sp.createSubscription("MySubscription_1",
        SpSubFlags.BASE,
        SpDeliveryType.DELIVER_PARSED,
        status); // "status" is an SpPlatformStatus object.

if (sub != null)
{
    /*
     * Client programmer must create a concrete class implementing
     * the aleri_PubSubnet::SpObserver interface. This concrete
     * SpObserver class will be
     * registered with the new SpSubscription object, and "notified" with
     * SpSubscriptionEvent objects when the SpSubscription is started.
     */
    SpObserver spObserver = new ClientSpObserver("myObserver");
    String streamName = "input";
    int cookie;

    /*
     * Client programmer must associate the concrete SpObserver
     * object(s) with a stream (or set of streams), and
     * register the SpObserver with the SpSubscription object.
     * This can be done using either the
     * SpSubscription.addStreamObserver * (...),
     * or SpSubscription.addStreamsObserver(...) methods.
     * This should be done for each SpObserver the client wishes
     * to be notified with the SpSubscriptions events.
     */
    cookie = sub.addStreamObserver(streamName, spObserver);

    if (cookie <= 0)
    {
        // SpObserver registration failed.
        return cookie;
    }

    /*
     * Once the SpObserver(s) are registered with the SpSubscription,
```

```
/* the SpSubscription.start() method is called,
 * which starts the * subscription process. If the startup
 * is successful, the appropriate SpObservers will be
 * notified with updates sent from the Sybase Aleri Streaming Platform.
 */
Console.WriteLine("Starting the subscription.");

rc = sub.start();

if (rc != 0)
{
    // The subscription could not be started.
    Console.WriteLine("Subscription could not be started, rc="+rc);
    Console.WriteLine("Error message ="
        SpUtils.getErrorMessage(rc));

    return rc;
}

/*
 * Block the main thread on input from the keyboard
 * while the subscription thread runs in the background.
 */
Console.ReadLine();

/*
 * The client programmer can invoke the "stop()"
 * method in order to terminate the running
 * SpSubscription.
 */
Console.WriteLine("Stopping the subscription.");

rc = sub.stop();

if (rc != 0)
{
    // Problems stopping the subscription.
    Console.WriteLine ("Problems stopping the
subscription, rc="+rc);
    Console.WriteLine ("Error message =" +
        SpUtils.getErrorMessage(rc));

    return rc;
}
}
else
{
    // Could not create the subscription object.
    rc = status.getErrorCode();
    Console.WriteLine ("Could not create subscription object, error="+rc);
    Console.WriteLine ("Error message =" +
        status.getErrorMessage());
}
```

It is very important that the `SpObserver` or `SpSubscriptionProjection` object created above is always in scope. To ensure this, add the following line to the end of the code before the return statement, that is always in scope during the lifetime of the subscription process:

```
GC.KeepAlive(spObserver);
```

Without this statement, you run the risk that the garbage collector will prematurely clean up the `SpObserver` object before the subscription is complete, leading to unpredictable results and program crashes.

A subscription created using `SpSubscription` receives updates for all columns in the stream with each event. If the subscription was created using `SpSubscriptionProjection`, however, an update consists of only the subset of columns defined by the SQL query. An update of this type is issued only when there is a change in one of the columns specified in the SQL query.

4.2.2.2. Initiate a Subscription Using SpSubscriptionProjection

If the `sp.createSubscriptionProjection(...)` call is successful, the client application program gets back an `SpSubscriptionProjection` object, which is used to instantiate the subscription. If the call fails for any reason (such as an invalid SQL Query), a null is returned, and the corresponding error information is set in the `SpPlatformStatus` object that was passed to the `createSubscriptionProjection(...)` method call.

The contents of the data returned from the Sybase Aleri Streaming Platform to the `SpSubscriptionProjection` object is determined by the SQL query passed into the `createSubscriptionProjection(...)` factory method. An `SpSubscriptionProjection` can only receive updates for the underlying stream specified in the SQL query, while the `SpSubscription` can get updates for more than one stream. The `SpSubscriptionProjection` interface extends the method set defined in the `SpSubscriptionCommon` interface, as follows:

```
SpStreamProjection ^getStreamProjection();  
int /*Cookie*/ addObserver(aleri_PubSubnet::SpObserver ^theObserver);
```

The `getStreamProjection()` method returns the `SpStreamProjection` object produced when the SQL query was sent to the Sybase Aleri Streaming Platform for parsing.

The `SpStreamProjection` should not be modified by the client application in any way. Typically, the `SpStreamProjection` object is specified into the `SpObserver`'s constructor, giving the `SpObserver` the list of fields and their corresponding data types. This information can be used by the `SpObserver` to process the updates that come back from the server.

See [Section 5.1, “Aleri SQL Queries and Statements”](#) for some limitations related to the Sybase Aleri Streaming Platform's handling of SQL queries.

You must create an `SpObserver` object to receive from the `SpSubscriptionProjection` the updates that arrive from the Sybase Aleri Streaming Platform. This object must implement the `SpObserver` interface. Refer to [Section 4.2.2.3, “The SpObserver Interface”](#) for more information.

Call the `addObserver(SpObserver theObserver)` method to register the `SpObserver` with the `SpSubscriptionProjection` object.

The `addObserver(...)` call returns an integer value that represents a “handle” to the registered `SpObserver` object. Later on, the client application programmer can use the cookie to remove the `SpObserver`.

The following code example shows how to create, configure, start, and stop an `SpSubscriptionProjection` object. In the following example, the “sp” object represents an `SpPlatform` object instantiated previously (through the `SpFactory::createPlatform(...)` method).

```
string sqlQuery =  
    "select intData, charData from inputstream where intData > 100";  
  
SpSubscriptionProjection subProj = sp.createSubscriptionProjection(  
    "MySubscriptionProjection_1",  
    SpSubFlags.BASE,  
    SpDeliveryType.DELIVER_PARSED,  
    sqlQuery,  
    status); // "status" is an SpPlatformStatus object.  
  
if (subProj != null)  
{
```

```
/*
 * Upon creation of an SpSubscriptionProjection object,
 * you can get the schema information that the
 * parsed sql query produced on the server as follows:
 */
SpStreamProjection streamProj = subProj.getStreamProjection();

/*
 * Client programmer must create a concrete class implementing
 * the aleri_PubSubnet::SpObserver interface. This concrete
 * SpObserver class will be registered with the new
 * SpSubscriptionProjection object, and "notified" with
 * SpSubscriptionEvent objects when the SpSubscriptionProjection is started.
 * NOTE: The SpStreamProjection object passes into the SpObserver
 * constructor. This gives the SpObserver the projections schema
 * information that is typically used for processing the update
 * events sent to the observer.
 */
SpObserver spObserver = new ClientSpObserver("myObserver", streamProj);
int cookie;

/*
 * Client programmer must associate the concrete SpObserver
 * object(s) with a stream (or set of streams), and
 * register the SpObserver with the SpSubscriptionProjection object.
 * This can be done using the
 * SpSubscriptionProjection.addObserver(...) method.
 * This should be done for each SpObserver the client
 * wishes to be notified with the SpSubscriptions events.
 */
cookie = subProj.addObserver(spObserver);

if (cookie <= 0)
{
    // SpObserver registration failed.
    return cookie;
}

/*
 * Once the SpObserver(s) are registered with the
 * SpSubscriptionProjection,
 * the SpSubscriptionProjection.start() method is called,
 * which starts the subscription process. If the startup
 * is successful, the appropriate SpObservers will be
 * notified with updates sent from the Platform.
 */
Console.WriteLine("Starting the subscription.");

rc = subProj.start();

if (rc != 0)
{
    // The subscription could not be started.
    Console.WriteLine("Subscription could not be started, rc="+rc);
    Console.WriteLine("Error message = " +
        SpUtils.getErrorMessage(rc));

    return rc;
}

/*
 * Block the main thread on input from the keyboard
 * while the subscription thread runs in the background.
 */
```

```
Console.ReadLine();

/*
 * The client programmer can invoke the "stop()"
 * method in order to terminate the running
 * SpSubscription.
 */
Console.WriteLine("Stopping the subscription.");

rc = subProj.stop();

if (rc != 0)
{
    // Problems stopping the subscription.
    Console.WriteLine ("Problems stopping the subscription, rc="+rc);
    Console.WriteLine ("Error message =" +
        SpUtils.getErrorMessage(rc));

    return rc;
}
}
else
{
    // Could not create the subscription object.
    rc = status.getErrorCode();
    Console.WriteLine ("Could not create subscription object, error="+rc);
    Console.WriteLine ("Error message =" +
        status.getErrorMessage());
}
}
```

It is important that the `SpObserver` and `SpSubscriptionProjection` objects created above are always in scope. To ensure this, add the following line to the end of the code (before the return statement), that is always in scope during the lifetime of the subscription process:

```
GC.KeepAlive(spObserver);
```

Without this statement, you run the risk that the garbage collector will prematurely clean up the `SpObserver` object before the subscription is complete, leading to unpredictable results and program crashes.

4.2.2.3. The `SpObserver` Interface

To receive stream updates from the Sybase Aleri Streaming Platform, the client application must implement the `SpObserver` interface. The interface is simple:

```
public interface SpObserver
{
    public System::String ^getName();

    /**
     * In the client's implementation of this interface, they
     * would simply "case" on the event types (and event ids)
     * that they are
     * notified with and handle them appropriately.
     */
}
```

```
public void notify(  
cli::array<SpSubscriptionEvent ^> ^theEvents);  
}
```

There are the two methods must be implemented within the class:

- The `getName()` method retrieves the “name” of the `SpObserver`. It is similar to the subscription object's `getName()` method.
- The `notify(cli::array<SpSubscriptionEvent ^> ^theEvents)` method is the “link” between the underlying Sybase Aleri Streaming Platform subscription, which the *subscription* object manages, and the client application object. The `SpSubscription` calls this method to send updates to the `SpObserver` as they come in from the Sybase Aleri Streaming Platform.

As updates flow from the Sybase Aleri Streaming Platform to the *subscription* object, the *subscription* object forwards them to the appropriate `SpObserver` objects (that is, those that have registered with the `SpSubscription` object), by calling each one's `notify(cli::array<SpSubscriptionEvent ^> ^theEvents)` implementation.

The subscription's underlying stream update acquisition and delivery mechanism runs in a separate thread used to manage the “read-only” Gateway I/O subscription socket, the `notify(cli::array<SpSubscriptionEvent ^> ^theEvents)` methods actually execute from within the context of this thread. The client application programmer must be conscious of this fact and program accordingly.

After the `SpObserver` class has been implemented, one or more instances of this class must be registered with the `SpSubscription` or `SpSubscriptionProjection` object created previously. To register the observer, call the `addStreamObserver` or `addStreamsObserver` method of the `SpSubscription` object, or the `addObserver` method of the `SpSubscriptionProjection` object, depending on the kind of Subscription.

Brief descriptions of these methods follow.

- The `addStreamObserver(System::String ^streamName, aleri_PubSubnet::SpObserver ^theObserver)` method instructs the `SpSubscription` object to send all updates for the *streamName* to the `SpObserver` object specified by the *theObserver* parameter. The client application can register as many observers as necessary.
- The `addStreamsObserver(cli::array<System::String ^> ^theStreamNames, aleri_PubSubnet::SpObserver ^theObserver)` method can be used to associate two or more streams with a particular `SpObserver`. The same thing can be accomplished by making multiple calls to the `addStreamObserver(...)` method described previously. Again, multiple `addStreamsObserver(...)` calls can be made to set up the streams and their corresponding observers.
- The `addObserver(pubsubnet::SpObserver ^theObserver)` method is defined in the `SpSubscriptionProjection` interface. It registers the specified observer with the `SpSubscriptionProjection` object. Although more than one observer can be added to the `SpSubscriptionProjection` object by calling this method multiple times, it is recommended that only one observer be added.

Note that this version of `addObserver` does not take a stream name. Any observer added to a `SpSubscriptionProjection` object will only get data specified in the SQL statement provided when the `SpSubscriptionProjection` object was created.

All three varieties of the `addObserver` method return an integer value that represents a “handle” to the registered `SpObserver` object. Later on, the client application programmer can use the cookie in a call to the `removeObserver(int theCookie)` method. The signature for this method is defined in the `SpSubscriptionCommon` interface.

4.2.2.4. Adding or Removing Streams from an Active Subscription

The `SpSubscription` object provides two methods that can modify an `SpSubscription` object's stream set after its `start()` method has been called: `subscribe(const char *streamName)` and `unsubscribe(const char *streamName)`.

Each of these methods takes a single `streamName` parameter. Before calling `subscribe(System::String ^streamName)`, the client application program must first ensure that there is an `SpObserver` associated with the about to be subscribed stream. The client application program can accomplish this by first calling `addStreamObserver(System::String ^streamName, aleri_PubSubnet::SpObserver ^theObserver)` to register the observer for the stream, then `subscribe(System::String ^streamName)`.

If successful, the `subscribe(String streamName)` and `unsubscribe(String streamName)` methods return zeroes. Otherwise, non-zero error codes are sent back to the caller. The `SpUtils.getErrorMessage(rc)` method can be called to see the error text associated with the error code.

Note that the `SpSubscriptionProjection` object does not have these two methods. There is no notion of streams for this object, only a notion of SQL Statements.

4.2.2.5. SHINE Flag Supports New Subscription Mode For Partial-Record Updates Using .NET

The SHINE flag can support a new subscription mode for partial-record updates in .NET with `SpSubFlags.SHINE`

The following is an example of how to use this mode:

```
sub = spPlatform.createSubscription("test_subname",  
                                   SpSubFlags.BASE|SpSubFlags.SHINE,  
                                   SpDeliveryType.DELIVER_PARSED,  
                                   spStatus);
```

4.2.3. Receive/Process Subscription Updates Using .NET 2.0

When the subscription is active, an array of `SpSubscriptionEvent` objects is delivered to the `SpObserver`. Each `SpSubscriptionEvent` object represents “something” that has happened to the subscription object, whereby the subscription object “thought” that it was appropriate to “notify” its registered `SpObserver` objects.

For example, an `SpSubscriptionEvent` is sent to the `SpObserver` if a stream update has arrived, or if the Sybase Aleri Streaming Platform is shut down, and so forth.

The `notify(...)` method that the programmer implements in the `SpObserver` object must iterate over the vector of `SpSubscriptionEvents` (each one uniquely identified by an `EventId`) to determine the action to be taken.

4.2.3.1. Parse Sybase Aleri Streaming Platform Data

The facility for inspecting parsing errors within the `SpObserver` is not supported by this release of the .NET version of the API.

4.3. Publishing to the Sybase Aleri Streaming Platform Using .NET 2.0

The Pub/Sub API defines an object of type `SpPublication`, which can enable a client application to publish information to the Sybase Aleri Streaming Platform.

Whether your program is publishing static data (such as a reference table) or dynamic data (such as stock market data) to the Sybase Aleri Streaming Platform, the same mechanism is used.

4.3.1. Create Objects for SP Publication Using .NET 2.0

As with the Subscription mechanism, the Pub/Sub API for .NET 2.0 defines objects that must be instantiated to make the publish process work.

4.3.1.1. Create the `SpPublication` Object

An `SpPublication` object is used by the Pub/Sub API to submit “publications” (stream data) to the Sybase Aleri Streaming Platform. This object is instantiated in a call to a factory method provided by the `SpPlatform` object which has been instantiated previously (as with the Pub/Sub API subscription mechanism). The signature of this factory method is:

```
SpPublication ^createPublication(System::String ^name,  
    aleri_PubSubnet::SpPlatformStatus ^status);
```

Details:

- *String name* is the name that the client application program intends to assign to the `SpPublication` object being created. It is not necessary for this name to be unique although it is a good idea for error reporting purposes.
- *SpPlatformStatus status* is an object that returns error code information from the `createPublication(...)` factory method if the `SpPublication` object cannot be created.

The following example shows how to use the `SpPlatform` object called “sp” to create an `SpPublication` object:

```
SpPublication pub = sp.createPublication("MyPublication_1", status);
```

In the above example, `status` is an `SpPlatformStatus` object that was created previously with the `SpFactory.createPlatformStatus()` factory method.

The `SpPublication` object is not re-entrant. If multiple threads are going to publish to the Sybase Aleri Streaming Platform, each thread should use a different `SpPublication` object. Each of these `SpPublication` objects should have its own socket connection to the Streaming Processor.

4.3.1.2. Create a Data Object for Publication

The following code example shows how to use the `createStreamDataRecord(...)` factory method to create an `SpStreamDataRecord` object that can be published to the Sybase Aleri Streaming Platform:

```
/*  
* Source Stream is called "input", and has the following  
* record layout:
```

```
/*
 * int, string, double, date, int, string, double, date
 */

Object[] fieldData = new Object[8];

fieldData[0] = intCounter++;
fieldData[1] = "do_mystring_" + intCounter;
fieldData[2] = doubleData++;
fieldData[3] = DateTime.Now;
fieldData[4] = intCounter;
fieldData[5] = "do_mystring2_" + intCounter;
fieldData[6] = doubleData++;
fieldData[7] = DateTime.Now;

SpStream stream = sp.getStream("input");

/*
 * Use the createStreamDataRecord(...) factory method to
 * bundle up the stream, fieldData vector, stream op code,
 * and stream flags into an SpStreamDataRecord object.
 *
 * At the moment, the SpStreamDataRecord object is the
 * basic unit of publication. You can publish these one at
 * a time, or you can publish them as a group (with or
 * without transaction blocks).
 *
 * NOTE: If you wish to publish a group of
 * SpStreamDataRecord objects
 * as a transaction, then all of the SpStreamDataRecords
 * within the group must belong to the same stream.
 */
SpStreamDataRecord sdr = SpFactory.createStreamDataRecord(
stream,fieldData,SpOpCodes. UPSERT,SpStreamFlags. NULLFLAG,
status);

if (sdr == null)
{
    System.Console.WriteLine("Could not createStreamDataRecord, status=" +
        status.getErrorCode());

    System.Console.WriteLine("Error Message:" +
        status.getErrorMessage());
    return status.getErrorCode();
}
```

The client application program can create a large number of these `SpStreamDataRecord` objects, placing each of them in a common array. Next, you can use one of the `SpPublication`'s publishing methods to send all rows of the stream data that are stored in the vector, to the Sybase Aleri Streaming Platform, either individually, using transactions, or envelopes.

The following code example shows how to publish an array of `SpStreamDataRecord` objects as a single transaction. In this example, `sp` is an `SpPlatform` object that was previously instantiated and `streamInputData` is an array that contains a large number of `SpStreamDataRecord` objects.

```
/*
 * Create the publication object associated with the platform.
 */
String name = "testPub_1";

SpPublication pub = sp.createPublication(name, status);

if (pub == null)
{
    System.Console.WriteLine("Couldn't create a publication object, status=" +
```

```
        status);
        System.Console.WriteLine ("Error message = " +
            status.getErrorMessage());
        return status.getErrorCode();
    }

    /*
    * Start the publication object (this opens up a GW I/O
    * socket connection). Don't forget to eventually close
    * down the SpPublication object(via the "stop()" method,
    * later on when you have finished using it,
    */
    rc = pub.start();

    if (rc != 0)
    {
        System.Console.WriteLine("Couldn't start the publication object.");
        System.Console.WriteLine ("Error message = " +
            SpUtils.getErrorMessage(rc));
        return rc;
    }

    /*
    * Publish the array of SpStreamDataRecord objects as one
    * big transaction.
    */

    rc = pub.publishTransaction(streamInputData,
        SpOpCodes.INSERT,
        SpStreamFlags. NULLFLAG,
        0);

    if (rc != 0)
    {
        System.Console.WriteLine ("Couldn't publish the transaction.");
        System.Console.WriteLine ("Error message = " +
            SpUtils.getErrorMessage(rc));
        return rc;
    }
}
```

4.3.1.3. Set/Get Methods for Exit-on-Drop, Exit-on-Timeout Capability to SpPublication Using .NET

Set methods should be called before SpPublication start() method. SpPlatformStatus should be checked after the start() for any possible problem. If you fail to send exit-on-close, the status is set to SP_ERROR_SETTING_EXIT. If you fail to send exit-on-timeout, the status is set to SP_ERROR_PUB_ERROR_SETTING_EXIT.

If setFinalizer and setExitOnTimeout are called, the second call returns SP_ERROR_PUB_EXIT_ALREADYSET (setFinalizer) or SP_ERROR_PUB_ACTION_ALREADYSET (setExitOnTimeout).

The following is an example using .NET 2.0.

```
void setExitOnTimeout(int timeout, SpPlatformStatus ^status);

    void setExitOnClose(SpPlatformStatus ^status);

        int getExitOnTimeout();

        bool getExitOnClose();
```

4.3.2. Handling Stale Data

When a publishing source stops sending data to the Sybase Aleri Streaming Platform, the previously published data is retained. Depending on how long it has been since the last update, you may not want this data to be used as if it were current. The publish/subscribe APIs include two functions to enable publishers to handle this data.

The “setFinalizer” function sets a timeout value (in milliseconds) and an SQL statement action. If the Sybase Aleri Streaming Platform receives no data on this connection within the specified time, the SQL statement is run. This SQL statement can perform any of the following actions:

- Delete previously published data.
- Mark previously published data as stale (via a field for that purpose in the data).
- Perform some other determined action on the source streams (and, consequently, the derived streams from these source streams).

In the following example, if the data is not updated within 2000 milliseconds, it is deleted.

```
setFinalizer(2000, "delete from Positions where SharesHeld > 1", spStatus)
```

The “sendHeartbeat” function sends a keep-alive message to the Sybase Aleri Streaming Platform. This function can be used to keep the connection alive and prevent the SQL statement from running, if “setFinalizer” has previously been called. As the following example shows, the “sendHeartbeat” function takes no arguments.

```
sendHeartbeat()
```

4.4. Record/Playback using .NET 2.0

In order to record data from the Sybase Aleri Streaming Platform, a client program needs to create and configure an SpNetPlatform object in the same way you would for subscribing or publishing. Once an SpNetPlatform object has been created, you should create an SpNetRecorder object using the factory method createRecorder(...). If an SpNetRecorder is created successfully, the program calls the start method. This spawns a background thread which subscribes to the configured streams and records all events for those streams. Recording will stop and the spawned thread terminate once the configured number of records have been processed or if the calling program calls stop(). Recording can be monitored by calling the getRecordCount() function which returns the number of records processed so far.

```
// Initialize SpFactory ...
// Create SpStatus and SpPlatformParms objects ...
// Create SpPlatform object ...

// Create recorder. It needs the following parameters to run
// recorder name (String) : a name to identify the instance of the recording object
// recorder file (String) : name of the file to store the recorded information
// streams                : array of strings, containing names of streams to record events for
// flags (int)            : recording options (encrypted/RSA/get base data)
// max records (int)      : maximum number of data records to record
// status                 : returns error messages if any

// init recorder parameters - recorder name, filename, streams, etc
SpRecorder recorder = spPlatform.createRecorder(recName, recFile, streams, flags, maxRecords, status);

if (null == recorder) {
    Console.WriteLine("Error starting recorder - " + status.getErrorMessage());
}
```

```
// cleanup ... and exit
} else
    recorder.start();

// Wait, monitor, etc ...

// To stop recording
recorder.stop();

// Cleanup
```

To play back recorded data, a client program creates an `SpPlayback` object using the factory method in `SpPlatform`. The 'scale' parameter is of particular interest. This is a double that can be used to scale the rate of playback as a factor of the original recorded rate (for example, twice as fast or half as slow). Values -1 to 1 have no effect - data is played back at the rate it was recorded. A value greater than 1 speeds up playback by that factor, for example, a value of 2 doubles the playback speed. A value less than -1 slows down playback by that factor, for example, a value of -3 will slow down playback by a factor of 3. The scale can be changed dynamically while playback is in progress.

```
// Initialize SpFactory ...
// Create SpStatus and SpPlatformParms objects ...
// Create platform object ...

// Create playback. It needs the following parameters to run
// playback name (string) : a name to identify the instance of the playback object
// playback file (string) : name of the file containing previously recorded data
// scale (double)         : allows to scale the playback rate
// max records (int)      : maximum number of data records to playback
// status                 : returns error messages if any

// init playback parameters - recorder name, filename, streams, etc

SpPlayback playback = spPlatform.createPlayback(playName, playFile, scale, maxRecords, status);

if (null == playback) {
    Console.WriteLine("Error starting recorder - " + status.getErrorMessage());
    // cleanup ... and exit
} else {
    playback.setSendUpsert(true); // optionally enable converting opcodes to UPSERT
    playback.start();
}

// Wait, monitor, etc ...

// To stop recording
playback.stop();
```

4.5. Special Topics for SP Publication/Subscription Using .NET 2.0

4.5.1. Publication/Subscription in a High Availability (Hot Spare) Configuration

The Sybase Aleri Streaming Platform can be started with a dual server configuration, in which one server is the primary server and the other is considered the Hot Spare and/or secondary server. The Pub/Sub API can be made aware of the High Availability configuration, and will perform an automatic switchover to the secondary server if the primary server goes down. See the *Administrators Guide* for more details on configuring a hot spare server.

The switchover to a Hot Spare server has an impact on any active `SpSubscription` and `SpPublication` objects. In that case, subscription and publication objects have to be re-established on the secondary server.

The Pub/Sub API is made aware of the High Availability configuration through the configured contents of the `SpPlatformParms` object that was passed into `SpFactory.createPlatform()`. Refer to the set of overloaded `SpFactory.createPlatformParms(...)` methods, and the `SpPlatformParms` object for High Availability configuration detail.

4.5.1.1. Subscription Mechanisms in a High Availability Configuration

When the primary server goes down, the underlying subscription thread receives an exception on the Gateway I/O socket connection that receives the stream updates being delivered from the primary server. When this event occurs, the Pub/Sub API recognizes that there is a High Availability configuration and attempts to connect to the secondary server and re-establish the subscription. Before it does this, the Pub/Sub API has to wait for the secondary server to internally change its state to that of a primary server. Once a successful connection is made to the secondary, Hot Spare, server and it has become the primary server, the subscriptions are re-established. During the switchover to the secondary server, the subscription object delivers several events to the `SpObserver` objects listening for `SpSubscriptionEvents`.

- `SpEventId.EVID_COMMUNICATOR_HALTED`

It is delivered to the `SpObserver` when the exception is received on the socket receiving the subscription messages from the primary server.

- `SpEventId.EVID_HOT_SPARE_SWITCH_OVER_INITIATED`

It is delivered to the `SpObserver` when the Pub/Sub API recognizes that a connection attempt should be made to the Hot Spare server. Note that the High Availability connection parameters were specified in the `SpPlatformParms` object passed to the `SpFactory.createPlatform()` method when the underlying `SpPlatform` was first created.

- `SpEventId.EVID_HOT_SPARE_SWITCH_OVER_SUCCEEDED`

It is delivered to the `SpObserver` when the connection to the Hot Spare server is made successfully.

- `SpEventId.EVID_HOT_SPARE_SWITCH_OVER_FAILED`

It is delivered to the `SpObserver` when the connection to the hot spare server fails.

If the switchover to the Hot Spare server in a High Availability Sybase Aleri Streaming Platform configuration is a success, the subscription(s) will be re-established using the same delivery flag values that were originally used when the subscription(s) against the primary server were established. This means that if the subscription originally requested the BASE snapshot of the stream, the new subscription (now going against the Hot Spare server) will also request the BASE snapshot of the stream as well. It's up to you to determine what needs to be done with the contents of the snapshot received from the Hot Spare server.

When there is a successful switchover to the Hot Spare server, the `SpPlatform` object takes note of this, and performs some internal bookkeeping, so that the `SpPlatform.getHost()` and `SpPlatform.getPort()` methods will return the host name and port number of the new primary server.

4.5.1.2. Publication Mechanisms in a High Availability Configuration

When the primary server goes down during an attempt to send a publication request to the server in a High Availability configuration, the `SpPublication` object will detect this and attempt to perform a switchover to the Hot Spare machine. If the switchover is successful, the publication object will then attempt to re-send the data to the new primary server. If the publication can not take place, a non-zero error code is returned to the caller indicating the problem.

You should treat the secondary server within a High Availability configuration as a passive server. The program should never log on to the secondary server, or attempt to send it data while the primary server is alive and well. It is the responsibility of the running High Availability configuration to manage both the primary and secondary servers appropriately. If the primary server in a High Availability configuration goes down, the secondary server will take over and become the new primary server. Once the sec-

ondary server becomes the primary server, data can then be published to the new primary server. Remember, the Pub/Sub API waits for the secondary server state to switch over to primary, before publishing data.

Chapter 5. The On-Demand SQL Interface

The client application can query streams, and modify source streams, in the Sybase Aleri Streaming Platform through the JDBC interface, ODBC interface, or in C/C++ programs through a native interface.

5.1. Aleri SQL Queries and Statements

The Sybase Aleri Streaming Platform accepts a subset of SQL92 for select, insert, update, and delete statements. A select statement has the form

```
select [distinct] [top <num>]
  <expr> [[as] <name>] [, <expr> [[as] <name>] ]*
from <stream>
[where <expr>]
[group by <expr> [, <expr>]*]
[order by <expr> [asc|desc] [, <expr> [asc|desc]]*]
```

where each expression *expr* is an expression. See *Authoring Reference Guide* for a list of functions and operations that can be used in such expressions. The form of select statements thus allows queries over one stream only (implying no joins). It does not allow subqueries either, nor advanced features like “between”. It is sufficient, though, for many queries.

The other SQL statements can be used to insert, update, or delete records from source streams. An insert statement has the form

```
insert into <stream> (<name> [, <name>]*)
values ( <expr> [, <expr>]*)
```

The column names appear in the second part of the statement, and the corresponding values for those columns appear in the third part (omitted columns are set to null automatically). An update statement has the form

```
update <stream> set <name>=<expr> [, <name>=<expr>]*
[where <expr>]
```

The optional where clause can be used to limit the updates to those records that pass the expression. A delete statement has the similar form

```
delete from <stream> [where <expr>]
```

The insert, update, and delete statements can be grouped into a single statement by separating them with semicolons. Thus,

```
delete from Dept where dn='SWP'; update Emp set dn='' where dn='SWP'
```

is a legal SQL statement too.

5.2. ODBC Connectivity

The Sybase Aleri Streaming Platform distribution includes a modified version of the Postgres ODBC driver. Using this driver, C/C++ programs send SQL queries or statements to the Sybase Aleri Streaming Platform. The drivers in third-party applications may also be used. For example, Microsoft Excel® can be used to pull data from the Sybase Aleri Streaming Platform into a spreadsheet.

The Sybase Aleri Streaming Platform distribution's `drivers/ODBC` directory contains the source code and pre-built binaries for:

- Microsoft Windows® 2000 and XP (in the win32 subdirectory)

- Linux 64-bit (in the `x86_64` subdirectory)
- Solaris 32-bit (in the `sun4` subdirectory)
- Solaris 64-bit (in the `sun4/sparcv9` subdirectory)

Linux and Solaris require the UNIX ODBC package, version 2.2.9 or above. See <http://www.unixodbc.org> for more information on installation and configuration.

The `win32` subdirectory contains the `.msi` file for first-time installation of the drivers, and an `upgrade.bat` file to replace previously installed drivers. Double-click the `.msi` file, or the `upgrade.bat` file, to perform the installation.

To configure Data Source Names (DSNs) on a Windows computer:

1. Open the Windows Control Panel.
2. Select **Administrative Tools**.
3. Select **Data Sources (ODBC)** and configure the DSNs.

This procedure makes these DSNs available for use by third-party tools.

5.3. JDBC Connectivity

The distribution includes a modified version of the Postgres JDBC driver. You can write Java programs that send SQL queries or statements to the Sybase Aleri Streaming Platform using this driver.

The modified version of the Postgres JDBC driver is included with the distribution in the `lib` directory. The JDBC driver JAR file must be in the classpath of the Java program. Load the driver using the following Java statement:

```
Class.forName("org.postgresql.Driver");
```

the form of the JDBC connection string is:

```
jdbc:postgresql://<sp server machine>:<sp sql port num>/<database>
```

In this version, the `<database>` portion of the connection string is ignored. The `<database>` connection will become meaningful when Container objects are implemented. The Java program can then connect using the `DriverManager.getConnection` JDBC method and send SQL statements to the Sybase Aleri Streaming Platform for processing. Refer to the Sun Microsystems Inc. [JDBC online documentation](http://java.sun.com/products/jdbc/) [http://java.sun.com/products/jdbc/] for more details on JDBC.

Chapter 6. The Command and Control Interface

Each Sybase Aleri Streaming Platform instance has a single Command and Control server thread. This module service handles requests that probe the running Sybase Aleri Streaming Platform for information (metadata), as well as those that direct it to perform various tasks (quiesce, shutdown, and so forth). The module is implemented as an XMLRPC server, along with a compact Abyss HTTP server to handle any Command and Control requests. Any external library that supports XMLRPC can be used to interface to Command and Control. The XMLRPC-C has been tested from C/C++, and libXMLRPC has been tested from Java.

6.1. Security for the On-Demand SQL Interface

6.1.1. Authentication Using the SQL On-Demand Interface

Authentication is performed through an initial call to the login function. The login function returns an authorization token (of type string) which must be passed in all subsequent commands.

6.1.2. Encryption Using the SQL On-Demand Interface

Currently, the lightweight Abyss server HTTP embedded in the Sybase Aleri Streaming Platform does not implement HTTPS. To get around this issue, and allow secure communications with the Command and Control interface, a small HTTPS-to-HTTP proxy program is included with the product. It is called `sslwrap` and may be invoked as follows:

```
sslwrap -port CnCPort -accept SecurePort -cert CertFile -key KeyFile -nbio
```

All Command and Control messages may be posted in a secure way using an HTTPS: address to the *SecurePort*. The certificate and key files must be the same files that are used to start the Sybase Aleri Streaming Platform.

The *sp_server* script supplied with the distribution wraps the Sybase Aleri Streaming Platform executable and the *sslwrap* utility together, so it can be managed by a single script.

Chapter 7. Embeddable Aleri Streaming Platform

Sybase Aleri Streaming Platform functionality can be embedded into a user process. You can only have one instance of the Sybase Aleri Streaming Platform in a process, and you can write your own main() function to instantiate it. It may use the option parsing as a normal sp-opt binary or provide options to the model in an already parsed form.

You can store the model in an encrypted form in this executable or receive it from an outside source, decrypt it and pass as an argument to the Sybase Aleri Streaming Platform. The model hiding the option can also prevent the users from seeing the model.

The Sybase Aleri Streaming Platform is instantiated and controlled using three classes:

- SpOptions - parsed option structure
- SpServer - instantiation of the Sybase Aleri Streaming Platform
- SpControl - access to control SpServer after it's started. Currently the only supported functionality is the stopping of the server.

An example of the main() function used by sp-opt as shipped:

```
#include <stdio.h>
#include "SpOptions.hpp"
#include "SpServer.hpp"
#include "SpControl.hpp"

using namespace std;

int main(int ac, char* av[])
{
    SpOptions spo;

    if (!spo.parseOptions(ac, av)) {
        SpOptions::usage(av[0]);
        exit(1);
    }
    if (spo._stopAfterFirstPass) {
        exit(0); //we just exit as the revision is the first thing
        printed.
    }
    if (!spo.isValid())
        exit(1);

    SpServer::setSignalHandlers();

    SpServer server(&spo);
    SpControl *control = server.initialize();
    if (control == 0)
        exit(1);

    int r = server.run();
    delete control;

    return r;
}
```


An example of a custom main() function with an embedded hidden model (also provided in examples/programming/embedded):

```
>
#include <stdio.h>
#include "SpOptions.hpp"
#include "SpServer.hpp"
#include "SpControl.hpp"

using namespace std;

int main(int ac, char* av[])
{
    char model[] =
">?xml version='1.0' encoding='UTF-8'?"
" >!-- _MODEL_MODEL_MODEL_MODEL_MODEL_MODEL_MODEL_MODEL_ -- > "
" >Platform xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' >"
"
" >Store file='store' id='store'/">"
"
" >SourceStream id='input' store='store' ofile='output/input.out'
" >"
" >Column datatype='int32' key='true' name='a'/">"
" >Column datatype='string' key='true' name='b'/">"
" >Column datatype='double' key='true' name='c'/">"
" >Column datatype='date' key='true' name='d'/">"
" >Column datatype='int32' key='false' name='intData'/">"
" >Column datatype='string' key='false' name='charData'/">"
" >Column datatype='double' key='false' name='floatData'/">"
" >Column datatype='date' key='false' name='dateData'/">"
" >/SourceStream>"
"
" >ComputeStream id='compute' istream='input' ofile=
"output/compute.out" store='store' >"
" >ColumnExpression key='true' name='a
\"> input.a >/ColumnExpression >"
" >ColumnExpression key='true' name='b
\">input.b>/ColumnExpression >"
" >ColumnExpression key='true' name='c
\">input.c >/ColumnExpression >"
" >ColumnExpression key='true' name='d
\">input.d >/ColumnExpression >"
" >ColumnExpression key='false' name='intData'>input.intData +
1 >/ColumnExpression >"
" >ColumnExpression key='false' name='charData
\">input.charData>/ColumnExpression >"
" >ColumnExpression key='false' name='floatData
\">input.floatData>/ColumnExpression >"
" >ColumnExpression key='false' name='dateData
\">input.dateData >/ColumnExpression >"
" >/ComputeStream >"
"
" >/Platform >";

    SpOptions spo;

    spo._debug = 7;
    spo._commandPort = 0;
    spo._sqlPort = 0;
    spo._optimize = true;
    spo._hideModel = true;

    if (!spo.initOptions()) {
        fprintf(stderr, "Failed to validate the options???\n");
        SpOptions::usage(av[0]);
        exit(1);
    }
    if (spo._stopAfterFirstPass) {
        exit(0); // we just exit as the revision is the first thing
        printed.
    }
    if (!spo.isValid()) {
        exit(1);
    }

    SpServer::setSignalHandlers();

    SpServer server(&spo);
    SpControl *control = server.initialize(model);
```

```
if (control == 0)
    exit(1);

// erase the model from memory
for (size_t i = strlen(model); i != 0; model[--i] = ' ') {}

int r = server.run();
delete control;

return r;
}
```

The meaning of the options in SpOptions matches those used on the sp-opt command line. The added options are:

- **bool _hideModel;** hides the model from the user.
- **string _platformHome;** allows you to override the Sybase Aleri Streaming Platform home string.
- **_classPath;** allows you to override the Java class path string.
- **_jvmPath;** lets you override the JVM library path.

The values of the last three options are normally read from the environment variables.

Note:

You must use a compiler that is certified for the Sybase Aleri Streaming Platform. Those include GNU g++ compiler version 4.2.1 for Linux and Solaris and Microsoft Visual C++® compiler 2005 for Windows®.

Chapter 8. Plug-in Connector Framework

8.1. Introduction

Connectors allow for easy integration of the Sybase server with external data sources and sinks. A Sybase data model can include Input Connections and Output Connections. Each connection is associated with a specific Data Location, where it is configured with a connector and all parameters required by that connector.

Sybase provides a number of external Adapters that are integrated with the plug-in connector framework. Once the Adapter and an associated connector profile are installed, Data Locations can be defined using the external Adapter. For more information on the use of these connectors, see the *Authoring Reference*.

The Plug-in Connector Framework provides a mechanism to create and add connectors that are not included in the set that comes with the Sybase Aleri Streaming Platform. Plug-in connectors can be added in the field and provide all necessary information to allow the Aleri Studio and server to manage and interact with an external data source or sink. After a connector profile (.cnxml) file is installed, a plug-in connector can be used in the Data Location Explorer in the same way as built-in connectors.

The plug-in connector framework can also be used by Sybase customers and third parties to extend the range of connectors with additional custom adapters and data sources or sinks. The plug-in connector's framework is fully documented, allowing you to develop your own connectors that are fully integrated with the Aleri Studio.

For example, the Reuters OMM Inbound Plug-in connector may be added to your model in advance of installing the Reuters OMM Adapter. Once the external Adapter is installed, simply fill in the appropriate connector profile parameters to use it.

Similarly, you may create your own connector profiles to control custom Adapters. These custom Adapters are simply data source/sink applications which themselves are built with Sybase's Pub/Sub API. By using the Plug-in Connector Framework, a connector can start and stop the Adapter and (optionally) provide discovery.

8.2. Plug-in Connector Profile

The profile is an XML file that contains the commands used by the Sybase components to start and stop the external Adapter or application, to optionally run data discovery, and other information that allows the external Adapter/application to be configured from the Aleri Studio. The framework defines the structure of the XML (.cnxml) file that contains the connector profile.

The example below shows a connector profile that uses four of the utilities shipped with the Sybase Aleri Streaming Platform (**sp_convert**, **sp_upload**, **sp_cli**, **sp_discXMLfiles**) to fully define a functional plug-in connector that supports browsing a directory of files, schema discovery, the creation of a source stream and data loading. This example profile, `simplified_xml_input_plugin.cnxml`, can be found in the `$PLATFORM_HOME/lib/connections` directory. The directory is included in the standard Sybase Aleri Streaming Platform distribution package.

Some of the very long lines below have been split for readability and formatting issues. If you are using this to create your own connector XML file, make sure that all command parameters are a single line, regardless of length.

```
<?xml version="1.0" encoding="UTF-8"?>
<Connector type="input" external="true"
  id="simplified_xml_input_plugin"
  label="Simplified external XML file input plugin connector"
```

```

descr="Example of uploading an XML file through a simple external connector"
>
<Library file="simple_ext" type="binary"/>

<!--
  the special section contains the special internal parameters
  which are prefixed with "x_". Although these are parameters,
  the framework requires them to be defined using the <Internal
  .../> element. They are hidden from the user in the Aleri Studio.
-->
<Special>
  <Internal id="x_initialOnly"
    label="Does Initial Loading Only"
    descr="Do initial loading, or the continuous loading"
    type="boolean"
    default="true"
  />
  <Internal id="x_addParamFile"
    label="Add Parameter File"
    type="boolean"
    default="false"
  />
  <Internal id="x_killRetryPeriod"
    label="Period to repeat the stop command until the process exits"
    type="int"
    default="1"
  />

  <!--
    Convert a file of xml record to Sybase Binary format using sp_convert,
    pipe into the sp_upload program, naming the upload connection:
    $platformStream.$platformConnection
  -->
  <Internal id="x_unixCmdExec"
    label="Execute Command"
    type="string"
    default="$PLATFORM_HOME/bin/sp_convert
      -p $platformCommandPort
      &lt;&quot;$directory/$filename&quot;; |
      $PLATFORM_HOME/bin/sp_upload
      -m $platformStream.$platformConnection
      -p $platformCommandPort"
  />
  <Internal id="x_winCmdExec"
    label="Execute Command"
    type="string"
    default="$+/{ $PLATFORM_HOME/bin/sp_convert}
      -p $platformCommandPort
      &lt;&quot;$directory/$filename&quot;; |
      $+/{ $PLATFORM_HOME/bin/sp_upload}
      -m $platformStream.$platformConnection
      -p $platformCommandPort"

  <!--
    use the sp_cli command to stop an existing sp_upload connection named:
    $platformStream.$platformConnection
  -->
  <Internal id="x_unixCmdStop"
    label="Stop Command"
    type="string"
    default="$PLATFORM_HOME/bin/sp_cli
      -p $platformCommandPort
      'kill every {$platformStream.$platformConnection}'
      &lt;/dev/null"
  />
  <Internal id="x_winCmdStop"
    label="Stop Command"
    type="string"
    default="$+/{ $PLATFORM_HOME/bin/sp_cli}
      -p $platformCommandPort &quot;kill every

```

```

        {$platformStream.$platformConnection}&quot; &lt;nul"
    />
    <!--
        use the sp_discXMLfiles command to do data discovery.
        The command below will have '-o "<temp file>"' added to it. It
        will write the discovered data in this file.
    -->
    <Internal id="x_unixCmdDisc"
        label="Discovery Command"
        type="string"
        default="$PLATFORM_HOME/bin/sp_discXMLfiles -d &quot;$directory&quot;"
    />
    <Internal id="x_winCmdDisc"
        label="Discovery Command"
        type="string"
        default="$+/{ $PLATFORM_HOME/bin/sp_discXMLfiles}
            -d &quot;$+/{ $directory}&quot;"
    />
</Special>


<Section>

    <!--
        Any parameter defined here, is visible in the Aleri Studio, and may
        be configured by the user at runtime in the data location explorer.
        These are defined according to the $PLATFORM_HOME/etc/Connector.xsd
        schema.
    -->

    <Parameter id="filename"
        label="File"
        descr="File to upload"
        type="tables"
        use="required"
    />
    <Parameter id="directory"
        label="path to file"
        descr="directory to search"
        type="directory"
        use="required"
    />
</Section>
</Connector>

```

To see how this works, fire up an instance of the Aleri Studio. You can use the following steps to test this plug-in connector:

1. Create a new data model using the visual editor or open an existing one.
2. Go to the Data Location Explorer and select the **plugin** section.
3. Click the **Create Data Location** icon on the top of the panel 
4. Enter a name like "xmlInputPlug-in".
5. From the pop-up lists of connector types, choose **Simplified external file input plugin connector**.
6. In the right panel labeled **Basic**
 - a. Enter a dummy file name, like **foo**.
 - b. Click in the **Path to File** attribute and navigate the directory selection dialog to

`$PLATFORM_HOME/examples/input/xml_tables.`

7. Now the Data Location is complete. After closing the Data Location panel, you can right click and discover and drag and drop any of the discovered tables onto the authoring palette in exactly the same way.

This simple plug-in and Data Location uses only a few of the supported system parameters and features of the Plug-In Connector Framework. The following sections contain the comprehensive list of all available parameters and command with definitions.

8.3. System Parameters and Commands

See examples in `$PLATFORM_HOME/lib/connections/PLUGIN_TEMPLATE.cnxml` for a sample cnxml file that may be copied and customized. It has all possible internal parameters embedded in it, and has comment blocks indicating their usage.

Parameters

x_paramFile Specifies the file name that where the connector framework writes all internal and user-defined parameters. It may use other internal parameters in specifying the file name. For example:

```
/tmp/mymodel.$platformStream.
$platformConnector.$platformCommandPort.cfg
```

Type: string

x_paraFormat Set to "prop", "shell" or "xml" to choose the format for the parameter file.

Type: string

x_addParamFile It determines if the parameter file name is automatically appended to all `x_cmd*` strings. For example, if you specify the command as "cmd -f", and this is set to true, the actual command that is executed will be "cmd -f <value of x_paramFile>".

Type: Boolean

x_initialOnly If true, does initial loading only. Set to false for continuous loading. Initial loading is useful for connectors that start, load some static data then finish, thus allowing another connector group to start up in a staged loading scenario.

Type: Boolean

x_killRetryPeriod If this parameter is >0 the `x_{unix,win}CmdStop` command will be retried every `x_killRetry` seconds, until the framework detects that the `x_{unix,win}CmdExec` command has returned. If it is equal to zero, only run the `x_{unix,win}CmdStop` once and assume that it has stopped the `x_{unix,win}CmdExec` command.

Type: integer

8.4. Read Only System Parameters

These parameters are filled in and available only when a model is started and the Sybase Aleri Streaming Platform is running. You cannot use these parameters for the discovery command.

<i>platformHost</i>	name of the host where the platform runs
<i>platformCommandPort</i>	number of the platform control port
<i>platformSsl</i>	1 if SSL is used, 0 otherwise
<i>platformSqlPort</i>	number of the Sybase Aleri Streaming Platform, SQL port
<i>platformAuth</i>	authentication of the Sybase Aleri Streaming Platform, with one of: "none", "pam", "rsa", "gssapi"
<i>platformStream</i>	stream on which this connector runs
<i>platformConnection</i>	name of this connector

8.5. Commands

Plug-in connector commands fall into two categories: those that run on the same host as the Aleri Studio, and those that run on the same host as the Sybase Aleri Streaming Platform. The discovery commands, **x_unixDiscCmd** and **x_winDiscCmd** always run on the Aleri Studio host. All other commands run on the Sybase Aleri Streaming Platform host.

The Aleri Studio and the Sybase Aleri Streaming Platform are frequently on the same host so the development of all command and driving scripts for the plug-in are straightforward. However, in the case of remote execution, when the Aleri Studio and the Sybase Aleri Streaming Platform are running on different hosts, the configuration becomes more complex.

For example, if the Aleri Studio is running on a Windows host, and the Sybase Aleri Streaming Platform is set up through the Aleri Studio to execute on a remote Linux host, it implies that the discovery command and the discovery file name that the framework generates are running/generated in a Windows environment. The path to the discovery file is a Windows-specific path with drive letter and '\' characters used as path separators. In this case, the developer of the connector should write the discovery command to run in a Windows environment while coding all other commands to remotely execute on the Linux box via a user-configured **ssh** or **rsh** command.

x_unixCmdConfig , x_winCmdConfig	The configure command should do any required parsing and/or checking of the parameters. It may also convert the parameters into the real format expected by the execution command by reading, parsing, and re-writing the parameter file. If the configure command fails (non-zero return), it's reported as a reset() error, and the connector fails to start.
x_unixCmdExec , x_winCmdExec	When the Sybase Aleri Streaming Platform starts the connector, it executes this command with its ending indicating that the connector has finished.
x_unixCmdStop , x_winCmdStop	The stop command runs from a separate thread, it should stop all processes created with the x_{unix,win}CmdExec commands, thus causing the x_{unix,win}CmdExec to return.
x_unixCmdClean , x_winCmdClean	The clean command runs after the the Sybase Aleri Streaming Platform has stopped the connection, that is, when x_{unix,win}CmdExec returns.

x_winDiscCmd

This command is for discovery. It should write a discovery file into the file name passed to it. The parameter -o "<temporary disc filename>" argument is appended to this command before it is executed.

```
<discover>
  <table name="table_name_1" />
    <column name="col_name_1" type="col_type_1"/>
    .
    .
    <column name="col_name_k" type="col_type_k"/>
  </table>
  .
  .
  <table name="table_name_n" />
    <column name="col_name_1" type="col_type_1"/>
    .
    .
    <column name="col_name_1" type="col_type_1"/>
  </table>
</discover>
```

8.6. User-Defined Parameters and Parameter Substitution

These internal parameters and any number of user-defined parameters can be created in a connector xml (cnxml) file. All parameters, system and user-defined, can be referenced in the command and/or script arguments. These parameters behave it in a similar way to shell substitution variables. The simplest example is from the previously described `simplified_xml_input_plugin.cnxml` file. Please note that some of the very long lines below have been split for readability and formatting issues.

```
<Internal id="x_unixCmdExec"
  label="Execute Command"
  type="string"
  default="$PLATFORM_HOME/bin/sp_convert
  -p $platformCommandPort
  <"$directory/$filename"
  | $PLATFORM_HOME/bin/sp_upload
  -m $platformStream.$platformConnection -p
  $platformCommandPort" />
```

External environment variables, such as `PLATFORM_HOME`, may be expanded, as well as internal system parameters (`platformCommandPort`) and user-defined parameters (`filename`). The full semantics for parameter expansion is:

```
$name
${name}
$name=value?substitution[:substitution]}
$name<>value?substitution[:substitution]}
$name!=value?substitution[:substitution]}
```



```

${name==value?substitution[:substitution]}
${name<value?substitution[:substitution]}
${name<=value?substitution[:substitution]}
${name>value?substitution[:substitution]}
${name>=value?substitution[:substitution]}

```

All forms with {} may have a "+" added after "\$" (for example, \${name}). The presence of "+" means that the result of the substitution will be parsed again and any values in it substituted.

"\" escapes the next character and prevents any special interpretation.

The conditional expression compares the value of a parameter with a constant value and uses either the first substitution on success or second substitution on failure. The comparisons "==" and "!=" try to compare the values as numbers. The "=" comparisons and "<>" try to compare values as strings. Any characters like "?", ":", and "." in the values must be shielded with "\". The characters "{" and "}" in the substitutions must be balanced, all unbalanced braces must be shielded with "\". The quote characters are NOT treated as special.

This form of substitution, \${...}, may contain references to other variables. This is implemented by passing the result of a substitution through one more round of substitution. The consequence is that extra layers of "\" may be needed for shielding. For example, the string

```

${name=?\\}

```

would produce one "\" if the parameter "name" is empty. On the first pass each pair of backslashes is turned into one backslash, and then on the second pass "\" turns into a single backslash.

Special substitution syntax for Windows convenience:

```

${/value}

```

```

${+/{value}}

```

Replaces all the forward slashes in the value by backslashes, for convenience of specifying the Windows paths that otherwise would have to have all the slashes escaped.

```

${%value}

```

```

${+{%value}}

```

Replaces all the '%' with '%%' as escaping for Windows.

If the resulting string is passed to shell or cmd.exe for execution, shell or cmd.exe would do its own substitution too.

Here is an example using some of the more powerful substitution features to define the execution command as in the simple example. However, you may make use of the conditional features to support optional authentication/encryption and an optional user-defined date format.

```

<Internal id="x_unixCmdExec"
  label="Execute Command"
  type="string"
  default="$PLATFORM_HOME/bin/sp_convert
    ${platformSsl=1?-e}
    ${dateFormat<>?-m '$dateFormat'}
    -c '${user=?user:$user}:$password'
    -p $platformCommandPort

```

```

        <"$directory/$filename" |
        $PLATFORM_HOME/bin/sp_upload
        ${platformSsl=1?-e} -m
        $platformStream.$platformConnection
        -c '$user:$password' -p $platformCommandPort"
    />

```

8.7. Notes on Auto Generated Parameter Files

The basic plug-in framework, when started, writes its set of parameters (system and user-defined) to a parameter file. This file is written in either:

- Java properties
- shell assignments
- simple XML format

Commands then have full access to the parameter file.

If you would like to see how the commands are used, suppose you added the following to the previous example, `simplified_xml_input_plugin.cnxml`,

```

        <Internal id="x_paramFile"
            label="Parameter File"
            type="string"
            default="/tmp/PARAMETER_FILE.txt"
        />
        <Internal id="x_paramFormat"
            label="Parameter Format"
            type="string"
            default="prop"
        />
        <Internal id="x_addParamFile"
            label="Add Parameter File"
            type="boolean"
            default="false"
        />

```

when the connector starts, it writes in `/tmp/PARAMETER_FILE.txt`

```

directory=/home/sjk/work/alari/cimarron
/branches/3.1/examples/input/xml_tables
filename=trades.xml
platformAuth=none
platformCommandPort=31415
platformConnection=Connection1
platformHost=sjk-laptop
platformSqlPort=22200
platformSsl=0
platformStream=Trades

```

Or a full list of all parameters, in the Java properties format. Note the format could have been specified

as "shell" for shell assignments, or as "xml" for a simple XML format.

When *x_addParamFile* is specified as true,

```
<Internal id="x_addParamFile"
  label="Add Parameter File"
  type="boolean"
  default="true"
/>
```

the argument `/tmp/PARAMETER_FILE.txt` is added to all commands prior to being executed.

8.8. A Parameter of Type `configFilename`

If you create a user-defined parameter of type *configFilename*, such as:

```
<Parameter id="ConfigFile"
  label="Connector configuration filename"
  type="configFilename"
  default=" "
/>
```

Then clicking in the value portion of this field in the Data Location Explorer will bring up a file selector dialog, allowing the user to choose a file on the local file system. Right-clicking on the read-only name brings up a user interface gesture, allowing for editing of the file contents. This provides the connector author a way to specify user editable configuration files.

8.9. Other Parameter Types

The `Connector.xsd` schema allows several useful types for user-defined parameters, including:

string	simple text
int	integer
uint	unsigned integer
range	fixed range integer
double	double precision floating point
choice	choose from fixed number of choices
filename	filename (brings up file selector dialog
directory	directory (brings up a directory selector dialog
tables	A string that is automatically filled in when a stream is created via the discovery mechanism.
password	text field that is hidden when entering data into it.
configFilename	filename (brings up a fileselector and file editor).

The other types in the `Connector.xsd` schema used in the internal connector framework but should not be used when creating plug-in connectors. Those include `runtimeFilename`, `runtimeDirectory`, `text`, `query`, and `permutation`.

Note

The Start/Stop commands are run by the Sybase Aleri Streaming Platform while discovery is run by the Aleri Studio. This distinction can effect use of the aforementioned parameters.

Appendix A. Reference Guide to the Java Object Model

A.1. Objects for Subscription

The following objects have been defined to use for creating applications that subscribe to the Sybase Aleri Streaming Platform.

A.1.1. SpFactory Object

The SpFactory object is used by the client code to create the set of objects that are required to use/control the Pub/Sub API. The SpFactory interface includes the following methods:

```
public static SpPlatform createPlatform(SpPlatformParms parms,
    SpPlatformStatus status);
public static SpPlatformParms createPlatformParms(String host,
    int port, String user, String password, boolean isEncrypted);
public static SpPlatformParms createPlatformParms(String host,
    int port, String user, String password, boolean isEncrypted,
    boolean useRsa);
public static SpPlatformParms createPlatformParms(String host,
    int port, String user, String password, boolean isEncrypted,
    String hotSpareHost, int hotSparePort);
public static SpPlatformStatus createPlatformStatus
    (String host, int port, String appName, String user,
    String password, boolean isEncrypted, String hotSpareHost,
    int hotSparePort);
public static SpPlatformStatus createPlatformStatus();
public static SpStreamDataRecord createStreamDataRecord(SpStream stream,
    Collection fieldData, int opCode, int flags, SpPlatformStatus status);
```

Details:

- The `createPlatform (SpPlatformParms parms, SpPlatformStatus status)` method returns a reference to an `SpPlatform` object if the Pub/Sub API was able to connect to the Sybase Aleri Streaming Platform and initialize properly.

Before calling this method, you have to use one of the overloaded `SpFactory.createPlatformParms(...)` methods, and the `SpFactory.createPlatformStatus()` method, to create the two parameters required by the `SpFactory.createPlatform (SpPlatformParms parms, SpPlatformStatus status)` method. The contents of the `SpPlatformParms` parameter control how the connection and authentication from the Pub/Sub API to the Sybase Aleri Streaming Platform takes place. If the connection can not be established, the `createPlatform(SpPlatformParms parms, SpPlatformStatus status)` method returns null, and a non-zero error code is set within the `SpPlatformStatus` object (see [Section A.1.3, “SpPlatformStatus Object”](#) for information on how to retrieve the error code/message).

- The `createPlatformParms(String host, int port, String user, String password, boolean isEncrypted)` method returns an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory.createPlatform(SpPlatformParms parms, SpPlatformStatus status)` method. This `createPlatformParms` method call sets up for basic connectivity (user name/password are used for authentication). If the `isEncrypted` flag is set to true, then https will be used to connect to the Command and Control process, and SSL socket connections will be made to the Gateway I/O process. If the `isEncrypted` flag is set to false, then http will be used to connect to the Command and Control process and regular (non-SSL) socket connections will be made to the Gateway I/O process.

- The `createPlatformParms (String host, int port, String user, String password, boolean isEncrypted, boolean useRsa)` method returns an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory.createPlatform(SpPlatformParms parms, SpPlatformStatus status)` method. This method also adds the `useRsa` flag. If this flag is set to `true`, the Pub/Sub API will attempt to authenticate to the Sybase Aleri Streaming Platform using the RSA mechanism. To use this mechanism, the Sybase Aleri Streaming Platform must be started with the `-k` option, whose argument specifies the path to the directory where the user's public RSA key file is stored —. See the *Administrators Guide* for information about key generation and placement.

When the Sybase Aleri Streaming Platform is using RSA authentication, the password of the `SpPlatformParms` object must specify the user's private RSA key file. For example, for a user called `foo`, there would be two RSA key files: the file `foo` (containing the public RSA key for user `foo`) and the file `foo.private.der` (containing the private RSA key for user `foo` in DER format). The public RSA key file called `foo` must be placed in a directory specified by the `-k` option to the Sybase Aleri Streaming Platform during startup.

The private RSA key file called `foo.private.der` must be placed on the client machine that is using the Pub/Sub API to connect to the server, and is specified using the `password` parameter of the `createPlatformParms(...)` method.

There are five variations of the `createPlatformParams` method; all accomplish the same creation of an `SpPlatformParams` object:

- basic
- basic with *UseRSA* flag
- basic with HotSpare
- HotSpare with *UseRSA*
- Kerberos authentication with or without the Hotspare

Choose the method that suits your needs.

- The `createPlatformParms(String host, int port, String user, String password, boolean isEncrypted, String hotSpareHost, int hotSparePort)` method returns an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory.createPlatform(SpPlatformParms parms, SpPlatformStatus status)` method. In addition to the basic connectivity parameters previously mentioned, this method adds two more parameters: *String hotSpareHost* and *int hotSparePort*. An `SpPlatformParms` object created with this factory method will cause the Pub/Sub API to use a High Availability configuration. In this configuration, the Pub/Sub API automatically attempts to switch over and use the secondary, Hot Spare, server if the primary server goes down. See [Section 2.4.6, “Publication/Subscription in a High Availability \(Hot Spare\) Configuration”](#) for more information on the High Availability configuration.
- The `createPlatformParms(String host, int port, String user, String password, boolean isEncrypted, boolean useRsa, String hotSpareHost, int hotSparePort)` method returns an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory.createPlatform(SpPlatformParms parms, SpPlatformStatus status)` method. This method allows you to set up the Pub/Sub API for RSA authentication and High Availability (Hot Spare), see the previous `createPlatformParms(...)` methods for a description of the RSA authentication and High Availability mechanisms.
- The `createPlatformStatus()` method returns an `SpPlatformStatus` object that is passed as the second parameter to the `SpFactory.createPlatform (SpPlatformParms`

parms, SpPlatformStatus status) method, in order to return status information to the caller. It is also used in several other methods within the Pub/Sub API that are needed to retrieve error code/status information. See the SpPlatformStatus object.

- The createPlatformParms (String host, int port, String appName, String user, String password, boolean isEncrypted, String hotSpareHost, int hotSparePort) method returns an SpPlatformParms object initialized to authenticate using the Kerberos V5 mechanism. The Sybase Aleri Streaming Platform must have been started with the option “-V gssapi”. The parameter host should be the fully qualified domain name of the machine running the Sybase Aleri Streaming Platform. The credentials for the Kerberos account are “user” and “password”. The string that points to an entry in the login configuration file in effect when the java virtual machine is started is “appname”. This configuration file can be specified as a command line define (-Djava.security.auth.login.config=) or in the security folder of the Java installation. For more information on configuring security for Java, refer to: <http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/spec/com/sun/security/auth/login/ConfigFile.html>.
- The createStreamDataRecord(SpStream stream, Collection fieldData, int opCode, int flags, SpPlatformStatus status) method returns an SpStreamDataRecord object that is used in the SpPublication object in order to publish data to the Sybase Aleri Streaming Platform.

The SpPlatformStatus object is passed in as the last parameter. If createStreamDataRecord fails, a null is returned to the caller and the SpPlatformStatus object indicates the error condition.

A.1.2. SpPlatformParms Object

The SpPlatformParms object is used by the SpFactory.createPlatform (Sp PlatformParms parms, SpPlatformStatus status). to create the SpPlatform object. You can create it using one of the overloaded SpFactory.createPlatformParms(...) methods described in [Section A.1.1, “SpFactory Object”](#). The SpPlatformParms object contains all of the connection information required by the SpPlatform object to make the appropriate connection(s) to the Sybase Aleri Streaming Platform. This information includes the host and port of the Command and Control Process, username, password, and flags indicating whether to use encryption, RSA authentication, Kerberos authentication, or the High Availability (Hot Spare) mechanism. The SpPlatformParms interface includes the following methods:

```
public String getHost();
public int getPort();
public String getUser();
public String getPassword();
public boolean isEncrypted();
public String getHotSpareHost();
public int getHotSparePort();
public boolean useRsa();
public String getAuthentication();
public String getAppName();
public void setGatewayHost(String host);
public String getGatewayHost();
```

Details:

- The getHost() method returns a string indicating the host name of the computer on which the Streaming Processor's Command and Control process is running.

- The `getPort()` returns an integer indicating the port number of the Command and Control process.
- The `getUser()` method returns a string containing the username for authenticating to the Sybase Aleri Streaming Platform.
- The `getPassword()` method returns a string containing the password that authenticates to the Sybase Aleri Streaming Platform. For RSA authentication, the password contains the file name of the user's private RSA key file.
- The `isEncrypted()` method returns a Boolean indicating whether or not encrypted connections used for the Command and Control and Gateway I/O processes. If the encryption mechanism is enabled, the Command and Control process connection will be made using https, while the Gateway I/O process will make SSL socket connections.
- The `getHotSpareHost()` method returns a string containing the host name of the secondary High Availability Streaming Processor. See the *Administrators Guide* for details on setting up a High Availability configuration.
- The `getHotSparePort()` method returns an integer indicating the port number of the secondary (Hot Spare) server in the High Availability configuration.
- The `useRsa()` method returns a boolean indicating if RSA authentication will be used when attempting to make connections to the Command and Control and Gateway I/O processes. If the RSA authentication mechanism is enabled, the password instance variable of the `SpPlatformParms` object must be set to the filename of the user's private RSA key file. If the RSA authentication mechanism is disabled, then the normal user name/password authentication mechanism will be used. See the *Administrators Guide* for RSA key file generation.

When using the Pub/Sub RSA authentication mechanism, the Streaming Processor must be started using the `-k public_rsa_key_directory` option See the *Administrators Guide* for more information.

- The `getAuthentication()` method returns a string describing the authentication mechanism currently in use. The `SpPlatformParms` provides predefined string constants for the authentications currently supported. These are:

```
public static final String AUTH_PAM = "PAM";
public static final String AUTH_RSA = "RSA";
public static final String AUTH_KERBV5 = "KERBV5";
```

- The `getAppName()` returns the string for the application name used when creating the `SpPlatformParms` object. If Kerberos authentication was not used, it returns NULL.
- The `getGatewayHost()` method returns the name of the gateway machine if it has been explicitly set by the user.
- The `setGatewayHost()` method sets the gateway machine which connects to the API. If set, the API ignores the value returned from the Sybase Aleri Streaming Platform. This is useful if the Sybase Aleri Streaming Platform is running on a machine without Domain Name System (DNS) entries.

A.1.3. SpPlatformStatus Object

The `SpPlatformStatus` object is used by several of the Pub/Sub API methods to return status in-

formation back to the caller. The `SpPlatformStatus` interface includes the following methods:

```
public int getErrorCode();
public String getErrorMessage();
public boolean isError();
```

Details:

- The `getErrorCode()` method returns an integer. If a problem was detected by the method `SpPlatformStatus` object was passed into, the value is non-zero; otherwise, zero is returned to indicate success.
- The `getErrorMessage()` method returns a string containing the error message text.
- The `isError()` method returns a boolean: true if an error was detected, false if no error was detected.

A.1.4. `SpPlatform` Object

The notion of the Sybase Aleri Streaming Platform has been abstracted into an object of the `SpPlatform` type.

An `SpPlatform` object is created using the `SpFactory.createPlatform(SpPlatformParms parms, SpPlatformStatus status)` method. Once instantiated, the `SpPlatform` object offers the following Sybase Aleri Streaming Platform functionality:

```
public String getUrl();
public String getUser();
public String getPassword();
public String getHost();
public String getGatewayHost();
public String getXMLModelVersion();
public int getPort();
public int getGatewayPort();
public int getDateSize();
public int getAddressSize();
public int getQuiesced();
public int getPrimaryServerFlag();
public Vector getBaseStreams();
public Vector getDerivedStreams();
public Vector getStreams();
public SpStream getStream(String streamName);
public SpStream getStream(int streamId);
public SpStreamDefinition
getStreamDefinition(String streamName);
public SpStreamDefinition
getStreamDefinition(int streamId);
public boolean isBigEndian();
public boolean isConnected();
public boolean isEncrypted();
public boolean useRsa();
public int shutdown();

public String getConfig(SpPlatformStatus status);
public int loadServerConfigFile(String configFile, String flags);
public int loadConfigString(String configString, String flags);
```

```
public int loadConfigStringApplyingConversion(String configString, \
String flags, String conversionConfigString);

public int addStreamToClient(int clientHandle, String streamName);

public int removeStreamFromClient(int clientHandle, String streamName);

public SpSubscription createSubscription(String name,
    int flags, int deliveryType, SpPlatformStatus status);

public SpSubscriptionProjection createSubscriptionProjection(String name,
    int flags, int deliveryType, String sqlQuery,
    SpPlatformStatus status);

public SpPublication createPublication(String name,
    SpPlatformStatus status);
```

Most of the `SpPlatform` methods communicate internally with the Command and Control process through the XMLRPC protocol. The `SpPlatform` methods allow you to retrieve Sybase Aleri Streaming Platform configuration information, source and derived stream objects, and so forth.

The `getUrl()` method returns a string representing the URL to connect to the Command and Control Process through XMLRPC. This string depends on whether the `SpPlatform` object was created with encryption enabled. Refer to [Appendix F, *Using Encryption with Java Client Applications*](#) for more information. The `isEncrypted()` method can be used to check if encryption was enabled when the `SpPlatform` object was instantiated.

The `getUser()` and `getPassword()` methods return strings that contain a username and password. These values are used internally for authentication when connecting to the Command and Control and Gateway I/O processes.

The `getHost()` method returns the name of the host machine on which the Command and Control Process is running. The `getGatewayHost()` method returns the name of the host machine on which the Gateway I/O Process is running. These two Sybase Aleri Streaming Platform processes reside on the same machine.

The `getPort()` and `getGatewayPort()` methods return the Command and Control port number and Gateway I/O port number, respectively. They are different because they refer to two different processes.

The `getXMLModelVersion()` method returns the XML model version used by the Sybase Aleri Streaming Platform.

The `getDateSize()` method returns the size of the datetime field type. If the Pub/Sub API is used to communicate with the Gateway I/O process, the datetime field type size is handled transparently to the user. If the client application uses custom Gateway I/O code, it will have to take care of this, as well as endianness, when sending datetime fields to the Sybase Aleri Streaming Platform.

The `getAddressSize()` method returns a value representing how the instance of the running Sybase Aleri Streaming Platform Server was compiled (either 32 or 64 bit).

The `getQuiesced()` method returns an integer that represents the “quiesced” state of the Sybase Aleri Streaming Platform. If successful, the method will return either a zero (`false`) or a one (`true`). If command execution fails, an error code is returned.

Note:

The error message associated with the error code can be retrieved by calling

`SpUtils.getErrorMessage(rc)`, where “rc” is the return code sent back from the `getQuiesced()` call.

The `getPrimaryServerFlag()` method returns an integer indicating whether or not the targeted server is considered to be the primary server in a High Availability (Hot Spare) configuration. It returns 1 for yes and 0 for no. If the command cannot be executed successfully, an error code (neither 0 nor 1) is returned.

You can use the Pub/Sub API to explicitly establish a connection to the secondary server within a High Availability (Hot Spare) configuration. If this happens, calling `getPrimaryServerFlag()` on the secondary server returns a value of zero, indicating that it's not a primary server.

The next group of methods is used to return stream metadata from the Sybase Aleri Streaming Platform. The metadata/schema for a stream is represented within the Pub/Sub API as an object of type `SpStream`. Refer to [Section 2.3.2.2, “SpSubscription Example”](#) for an example. The `getBaseStreams()` method returns a vector of `SpStream` objects representing all of the source streams residing on the Sybase Aleri Streaming Platform. Similarly, `getDerivedStreams()` returns a vector of `SpStream` objects that represents all of the derived streams residing on the Sybase Aleri Streaming Platform. The `getStreams()` method returns a vector of `SpStream` objects that represents “all” streams (both source and derived streams) residing on the Sybase Aleri Streaming Platform. For a particular stream, you can look up the stream by its “name” or “id” using the `getStream(String streamName)` or `getStream(int streamId)` method, respectively.

The `getStreamDefinition (String streamName)` and `getStreamDefinition (int streamId)` methods return an object of type `SpStreamDefinition` for the specified `streamName` or `streamId`, respectively. Refer to [Section A.1.6, “SpStreamDefinition Object”](#) for more information.

The `isBigEndian()` method returns true if the Streaming Processor is running on a big-endian machine, or false if the Streaming Processor is running on a little-endian machine.

The `isConnected()` method returns true if the `SpPlatform` object is currently connected to the Sybase Aleri Streaming Platform. Otherwise, it returns false. For example, if the client application program issues a *shutdown*, subsequent `isConnected()` calls return false.

Once an `SpPlatform` object is shut down, you should set its reference to null. Later on, another `SpPlatform` object can be instantiated using the `SpFactory.createPlatform(...)` method.

The `shutdown()` method alerts the Command and Control Process to shut down the Sybase Aleri Streaming Platform. This causes all socket connections to the Sybase Aleri Streaming Platform to be closed. If the application has subscriptions running at the time of a shutdown, the `SpObserver` objects of those subscriptions are notified before the shutdown process starts.

The `getConfig(SpPlatformStatus status)` method returns a String containing the AleriML configuration currently being executed by the running Sybase Aleri Streaming Platform instance. If there is an error in retrieving the AleriML configuration information from the server, this method will return an empty string, and the error code will be stored in the `SpPlatformStatus` parameter passed into the method.

The `loadServerConfigFile(String configFile, String flags)` method attempts to load an AleriML configuration file into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information during the AleriML configuration file load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string. Consult the *Administrators Guide* for more information on loading AleriML configurations, and the various options that can be specified in the `flags` parameter. If the AleriML configuration file was loaded successfully, the method returns zero. If the AleriML configuration file could not be loaded successfully, the method returns a non-zero error code. In addition, when loading an AleriML configuration file into the server, inspect the log messages located on the server for more information.

The `loadConfigString(String configString, String flags)` method attempts to

load the AleriML configuration stored in the `configString` parameter, into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information during the AleriML configuration string load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string. If the AleriML configuration was loaded successfully, the method returns zero. If the AleriML configuration could not be loaded successfully, the method will return a non-zero error return code. In addition, when loading an AleriML configuration into the server, inspect the log messages located on the server for more information.

The `loadConfigStringApplyingConversion (String configString, String flags, String conversionConfigString)` method attempts to load the AleriML configuration stored in the `configString` parameter, into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information used during the AleriML configuration string load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string.

The `conversionConfigString` parameter provides an AleriML model that is used to apply specific conversion instructions during a dynamic modification of the Sybase Aleri Streaming Platform's XML configuration. Consult the *Administrators Guide* for more information on loading AleriML configurations, and the various options that can be specified in the `flags` parameter. If the XML configuration was loaded successfully, the method returns zero. If not, the method returns a non-zero error code. You can get more information about the dynamic load process from the server's log messages.

The `SpPlatform` object provides two subscription-related methods if you want to write your own low-level Gateway I/O code for the subscription. These methods are `addStreamToClient(int clientHandle, String streamName)` and `removeStreamFromClient(int clientHandle, String streamName)`. These methods are part of the `SpPlatform` interface because they are XMLRPC calls that are used to manage the subscription characteristics of a Gateway I/O socket on which a subscription is currently running.

The two method calls must be provided when the Pub/Sub subscription mechanism is not being used.

Once a subscription request is issued for an open Gateway I/O socket connection, the connection becomes a read-only connection. Asynchronous stream updates are delivered from the Sybase Aleri Streaming Platform to the client. Because of the “read-only” nature of the socket, additional Gateway I/O commands can not be issued on this socket connection. The XMLRPC mechanism must be used to do this.

The `add` and `remove` methods are passed a `clientHandle` as the argument. The `clientHandle` is an integer value that is returned by the Gateway I/O process when the client application sends a low-level subscription request on the socket. The `addStreamToClient(...)` method allows you to add an additional stream to the subscription list, while the `removeStreamFromClient(...)` method lets you delete a stream from the subscription list.

The `createSubscription (String name, int flags, int deliveryType, SpPlatformStatus)` method is used to create an `SpSubscription` object associated with the `SpPlatform` object. As the name implies, it is also a factory method used to create `SpSubscription` objects. An `SpSubscription` object has its own interface (or little API) that is used to control the subscription. The `SpPlatform` object can be used to create a subscription.

A subscription is used to get asynchronous stream updates from the Sybase Aleri Streaming Platform into the client application. Similarly, there is a factory method called `createPublication(String name, SpPlatformStatus status)` that creates an `SpPublication` object. An `SpPublication` object is used to “publish” stream input data (and/or issue the Gateway I/O **commit()** command) from the client application, to the Sybase Aleri Streaming Platform.

A.1.5. SpStream Object

The `SpStream` object store the metadata associated with a stream on the Sybase Aleri Streaming Platform. The `SpStream` interface includes the following methods:

```
public int getId();
public String getName();
public boolean isBase();
public SpStreamDefinition getDefinition();
```

Details:

- The `getId()` method returns an integer that represents the stream's internal identifier on the Sybase Aleri Streaming Platform.
- The `getName()` method returns the name of the stream.
- The `isBase()` method returns `true` if the stream is a source stream, `false` otherwise.
- The `getDefinition()` method returns a reference to an object of type `SpStreamDefinition`, which contains the stream's schema information.

A.1.6. SpStreamDefinition Object

An `SpStreamDefinition` object stores the schema associated with a stream residing on the Sybase Aleri Streaming Platform. The `SpStreamDefinition` interface includes the following methods and constants:

```
/**
 * Streams ColumnInterface col type definitions.
 */
static public final int COLTYPE_INT32    = 1;
static public final int COLTYPE_INT64    = 2;
static public final int COLTYPE_DOUBLE   = 3;
static public final int COLTYPE_DATE     = 4;
static public final int COLTYPE_STRING   = 5;
static public final int COLTYPE_NULLVALUE = 6;
static public final int COLTYPE_MONEY    = 7;
static public final int COLTYPE_TIMESTAMP = 8;
public int      getNumColumns();
public Vector   getColumnNames();
public Vector   getColumnTypes();
public Vector   getKeyColumns();
public Vector   getKeyColumnVector();
public boolean  isKeyColumn(int columnIndex);
```

Details:

- The `getNumColumns()` method returns the number of columns in the stream.
- The `getColumnNames()` method returns a vector of strings. Each string is the name of the corresponding column. The vector's size equals the value returned from the `getNumColumns()` method.
- The `getColumnTypes()` method returns a vector of integers. Each integer represents the field type of the corresponding column. The `SpStreamDefinition` contains a list of integer "constants" representing the various column types.

- The `getKeyColumns()` method returns a vector of integers. Each integer is the column index (rel 0) of a key column in the streams field list. For example, if the stream has five columns, and the first three are key columns, the `getKeyColumns()` method returns `[0, 1, 2]`.
- The `getKeyColumnVector()` method returns a vector of zeroes and ones whose size equals the number of columns in the stream. For example, if the stream has five columns and the first three are key columns, the `getKeyColumnVector` method returns `[1,1,1,0,0]`.
- The `isKeyColumn(int columnIndex)` returns a Boolean value of `true` if the column index specified is that of a key field; otherwise, it returns `false`.

The `columnIndex` is “rel-0” (in other words, the first column of the field list has an index value of zero).

A.1.7. SpStreamProjection Object

The `SpStreamProjection` object stores the metadata associated with a stream projection based on an SQL query that is supplied to the `createSubscriptionProjection(...)` factory method of the `SpPlatform` object.

The `SpStreamProjection` interface includes the following methods:

```
public SpStream getStream();
public SpStreamDefinition getDefinition();
```

where:

- The `getStream()` method returns a reference to the underlying `SpStream` onto which the SQL query was projected.
- The `getDefinition()` method returns a reference to an object of type `SpStreamDefinition`, containing the schema information of the projection. This information is returned by the Sybase Aleri Streaming Platform when the SQL query associated with an `SpSubscriptionProjection` object is created.

A.1.8. Creating an SpSubscription or SpSubscriptionProjection Object

The client application must create an `SpSubscription` or `SpSubscriptionProjection` object in order to use the Pub/Sub API to make subscription requests. These objects can be created using the appropriate factory method provided by the `SpPlatform` object that has been instantiated previously. These factory methods have the following signatures:

```
SpSubscription createSubscription(String name,
int flags, int deliveryType, SpPlatformStatus status);

SpSubscriptionProjection createSubscriptionProjection(String name,
int flags, int deliveryType, string sqlQuery,
SpPlatformStatus status);
```

Details:

String name: an identifier assigned by the client application program to the `SpSubscription` or `SpSubscriptionProjection` object.

int flags: the flag bitmap that is to be sent to Gateway I/O process when the low-level subscription request is made. The flag settings control delivery from the Sybase Aleri Streaming Platform to the client application, on the Gateway I/O socket connection where the subscription request was made. The “flag bits” are defined as constants in the `SpSubscriptionCommon` interface that abstracts the commonality between the `SpSubscription` and `SpSubscriptionProjection` interfaces. The constants are as follows:

These flag bits can be ORed together using the “|” operator. For example:

```
flags = SpSubscription.NOBASE | SpSubscription.LOSSY.
```

- `BASE = 0x0`;

The `BASE` flag bit tells the Sybase Aleri Streaming Platform to send a complete “snapshot” of each stream of the subscription request before sending each stream subsequent updates. The complete “snapshot” or “state” of the stream is a group of “insert” records sent from the Sybase Aleri Streaming Platform between the “`EVID_GATEWAY_SYNC_START`” and “`EVID_GATEWAY_SYNC_END`” subscription events.

- `LOSSY = 0x1`;

The `LOSSY` flag bit puts the Sybase Aleri Streaming Platform in “data shedding mode”, in which the Sybase Aleri Streaming Platform drops the oldest data if the client cannot keep pace with the data coming out of the Sybase Aleri Streaming Platform.

- `NOBASE = 0x2`

The `NOBASE` flag bit tells the Sybase Aleri Streaming Platform that it should NOT send a complete “snapshot” of each stream in the subscription request. The Sybase Aleri Streaming Platform just proceeds to send subsequent updates for each of the streams.

- `DROPPABLE = 0x8`;

This flag tells the Sybase Aleri Streaming Platform that if the client (the application using this flag) can't keep up with the data it is sending, and its internal buffer is filled, the Sybase Aleri Streaming Platform should drop the connection to the client application. This protects the Sybase Aleri Streaming Platform from getting into a situation where it has to stop processing incoming data because its clients can't keep up with the data it is producing.

- `PRESERVE_BLOCKS = 0x20`

This flag tells the Sybase Aleri Streaming Platform that it should preserve blocks while sending data to the client application.

int deliveryType: the integer value that specifies how the client application program's `SpObserver` object receives the stream update events. Currently, there are several delivery type format specifiers defined in the `SpSubscriptionCommon` interface, as listed below:

- `DELIVER_PARSED = 1`;

This delivery type setting tells the subscription object to deliver “parsed” field data objects repres-

enting the stream update to the client programmer's `SpObserver` object.

- `DELIVER_BINARY = 3;`

This delivery type setting tells the subscription object to deliver the “binary” representation of the stream update record to the client programmer's `SpObserver` object.

- `DELIVER_STREAM_OPCODES = 5;`

This delivery type setting tells the subscription object to extract the stream op code from the record (*INSERT*, *UPDATE*, *DELETE*, *UPSERT* and so forth), otherwise leaving the record in binary format.

`sqlQuery` specifies the SQL query projection that the `SpSubscriptionProjection` object will be based on. The `sqlQuery` parameter is only used to create an `SpSubscriptionProjection` object.

See [Section 5.1, “Aleri SQL Queries and Statements”](#) for some limitations related to the Sybase Aleri Streaming Platform's handling of SQL queries, which also apply in this situation.

The `SpPlatformStatus` object retrieves error code information back from the `createSubscription(...)` and `createSubscriptionProjection(...)` factory methods, if the subscription object could not be created.

The following example shows how to use the `SpPlatform` object called `sp` to create both an `SpSubscription` and an `SpSubscriptionProjection` object:

```
SpSubscription sub = sp.createSubscription("MySubscription_1",
    SpSubscriptionCommon.BASE, SpSubscriptionCommon.DELIVER_PARSED,
    status);

SpSubscriptionProjection subProj = sp.createSubscriptionProjection(
    "MySubscriptionProjection_2", SpSubscriptionCommon.BASE,
    SpSubscriptionCommon.DELIVER_PARSED,
    "select intData, charData from inputstream where intData > 100",
    status);
```

In the above example, `status` is an `SpPlatformStatus` object that was created previously with the `SpFactory.createPlatformStatus()` factory method.

A.1.9. `SpSubscriptionCommon` Method Set

If the `createSubscription(...)` or `sp.createSubscriptionProjection(...)` call is successful, the client application gets back either an `SpSubscription` or `SpSubscriptionProjection` object, which is used to instantiate the subscription. The set of methods that the `SpSubscription` and `SpSubscriptionProjection` types have in common have been abstracted into an interface called `SpSubscriptionCommon`. This interface is extended by both the `SpSubscription` and `SpSubscriptionProjection` interfaces. The `SpSubscriptionCommon` interface includes the following method set:

```
public String getName();
public int getFlags();
public int getDeliveryType();
```



```
public int getClientHandle();  
public int removeObserver(int theCookie);  
public int start();  
public int stop();
```

The `getName()` method returns the name assigned in the application to the subscription object when it was created with the `SpPlatform`'s `createSubscription(...)`, or `createSubscriptionProjection(...)` method.

Similarly, the `getFlags()` and `getDeliveryType()` methods return the flag settings and the delivery type, respectively, that were specified in the `createSubscription(...)` or `createSubscriptionProjection(...)` method.

The `getClientHandle()` method returns the “handle” assigned to the underlying subscription connection by the Sybase Aleri Streaming Platform. Valid handles are positive integers. The value of the handle is acquired from the Sybase Aleri Streaming Platform when the subscription is started through the `start()` method.

The `removeObserver(int theCookie)` method is used to remove an `SpObserver` from the subscription's delivery mechanism. There are differences between how you add observers to the two different types of subscriptions. These differences will be discussed within the `SpSubscription` and `SpSubscriptionProjection` interfaces.

The `start()` method is used to start the subscription process.

There must be at least one stream and `SpObserver` registered with the subscription object before the subscription object can be started up through the `start()` method. See the `SpSubscription` and `SpSubscriptionProjection` descriptions to see how to register `SpObserver` objects.

When you start up a subscription object, the following sequence takes place:

1. The `Subscription` object establishes a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process, and authentication is performed.
2. A subscription request is sent to the Sybase Aleri Streaming Platform on this socket connection.
3. If the subscription request is accepted by the Sybase Aleri Streaming Platform, the `Subscription` object reads the “client handle” that the Sybase Aleri Streaming Platform assigned to this subscription request.
4. A new thread is started up. It is dedicated to reading stream update information off the read-only Gateway I/O socket connection.

When the client application's `SpObserver` objects are “notified” for stream updates, through their `notify(...)` methods, the `SpObserver` objects will actually be running within the context of this thread (not the main thread).

5. Stream update messages flowing from the Sybase Aleri Streaming Platform to the client are read, parsed and delivered to the client application's `SpObserver` objects.
6. The `start()` method returns a zero indicating that the subscription was started up successfully. If there is an error, it returns a non-zero error code.

The `SpUtils.getErrorMessage(errorCode)` method can be used to get the specific error

message.

The `stop()` method is used to shut down the Subscription mechanism. The `stop()` method closes the socket connection and stops the thread that was used to read, parse and deliver the Sybase Aleri Streaming Platform updates to the `SpObserver` objects.

Here are additional methods in the interface:

```
public void setPulseInterval(int pulseInterval);
public int getPulseInterval();
public void setQueueSize(int queue, SpPlatformStatus status);
public int getQueueSize();
public void setBaseDrainTimeout(int millis, SpPlatformStatus status);
public int getBaseDrainTimeout();
public void setExitOnClose(SpPlatformStatus status);
public boolean getExitOnClose();
```

- `setPulseInterval` can set the pulse interval in seconds if the subscription was created with the pulsed flag on.
- `getPulseInterval` is used to retrieve the current setting of the pulse interval in seconds.
- `setQueueSize` is used to set the internal buffer size in the Sybase Aleri Streaming Platform for this subscription. The Sybase Aleri Streaming Platform uses this buffer to queue up messages if the subscriber is slow in retrieving them. It can prevent the subscriber from blocking and slowing down the Sybase Aleri Streaming Platform.
- `getQueueSize` retrieves the current value of the queue size.
- `setBaseDrainTimeout` is used to set the time in milliseconds that the Sybase Aleri Streaming Platform should wait before dropping a blocked subscription. If a subscription is started with the *DROPPABLE* flag set, the Sybase Aleri Streaming Platform closes a subscription connection if the messages block due to a slow client. This parameter specifies how long to wait before closing the connection.
- `getBaseDrainTimeout` retrieves the current value in milliseconds of the base drain timeout.
- If `setExitOnClose` is set, the Sybase Aleri Streaming Platform will shut down once this subscription connection is closed by the client.
- `getExitOnClose` retrieves the current setting of the exit on close flag.

A.1.10. SpSubscription Method Set

If the `createSubscription(...)` call is successful, the client application gets back an `SpSubscription` object, which it ultimately uses to subscribe. An `SpSubscription` object can be used to subscribe to one or more streams, whereas an `SpSubscriptionProjection` object can only be used to subscribe to the projection defined by the *sqlQuery* passed into the `createSubscriptionProjection(...)` factory method. For each stream that is being observed, the `SpSubscription` object will deliver to the `SpObserver` stream events that contain all of the stream's fields. The `SpSubscription` extends the method set defined in the `SpSubscriptionCommon` interface as follows:

```
public int /*Cookie*/ addStreamObserver(String streamName,
    SpObserver theObserver);
```

```
public int /*Cookie*/ addStreamsObserver(Collections theStreamNames,
    SpObserver theObserver);

public int subscribe(String streamName);

public int unsubscribe(String streamName);
```

The next few methods are used to set up the streams and their corresponding `SpObserver` objects for the `SpSubscription`. The client applications must create their own `SpObserver` objects, which are notified by the `SpSubscription` with the updates that arrive from the Sybase Aleri Streaming Platform for the registered streams. The client application programmers create `SpObserver` objects by implementing the `SpObserver` interface.

The `addStreamObserver(String streamName, SpObserver theObserver)` method instructs the `SpSubscription` object to send all updates for the `streamName` to the `SpObserver` object specified by “theObserver” parameter. Several observers can be registered on the same stream.

The `addStreamsObserver(Collection theStreamName, SpObserver theObserver)` method can be used to associate several streams with a particular `SpObserver` at once. The same thing can be accomplished by making multiple calls to the `addStreamObserver(...)` method shown previously. Again, multiple `addStreamsObserver(...)` calls can be made to set up the streams and their corresponding observers.

The `addStreamObserver(...)` and `addStreamsObserver(...)` calls return an integer value that represents a “cookie”/“handle” to the registered `SpObserver` object. Later on, the client application program can use the cookie to remove the `SpObserver`.

Another method, `removeObserver(int theCookie)`, is used to remove the `SpObserver` from the `SpSubscription`'s delivery mechanism. Its signature is defined in the `SpSubscriptionCommon` interface.

The `SpSubscription` object provides two methods that can modify the stream set that is currently being managed by a “running” subscription: `subscribe(String streamName)` and `unsubscribe(String streamName)`.

These two methods take a single *streamName* as parameter. In the case of the `subscribe(String streamName)` call, the client application program must first ensure that an `SpObserver` is associated with the subscribed stream. The client application program must first call the `addStreamObserver(streamName, theObserver)` method to register the observer for the stream, and then call the `subscribe(streamName)` method.

If successful, the `subscribe(String streamName)` and `unsubscribe(String streamName)` methods return zero. Otherwise, a non-zero error code is returned. In the latter case, the `SpUtils.getErrorMessage(rc)` method can be used to retrieve the error text associated with the error code.

A.1.11. SpSubscriptionProjection Method Set

If the `createSubscriptionProjection(...)` call is successful, the client application gets back an `SpSubscriptionProjection` object that is ultimately used to perform the subscription process. The contents of the data returned from the Sybase Aleri Streaming Platform back to the `SpSubscriptionProjection` object is determined by the SQL query that was passed into the `createSubscriptionProjection(...)` factory method. An `SpSubscriptionProjection` can only receive updates for the underlying stream specified in the SQL query, while the `SpSubscription` can get updates for more than one stream if so desired. The `SpSubscriptionProjection` interface extends the method set defined in the `SpSubscriptionCommon` interface, as

follows:

```
public SpStreamProjection getStreamProjection();  
public int /*Cookie*/ addObserver(SpObserver theObserver);
```

The `getStreamProjection()` method returns the `SpStreamProjection` object produced when the SQL query was sent to the Streaming Processor for parsing, which happens during the execution of the `createSubscriptionProjection(...)` factory method. If the SQL query could not be parsed, `createSubscriptionProjection(...)` returns null, and the corresponding error information is sent to the associated `SpPlatformStatus` object. If the `createSubscriptionProjection(...)` method succeeds, the caller gets back an `SpSubscriptionProjection` object, and can then make a `getStreamProjection()` call for the schema information produced by the SQL query parse. The `SpStreamProjection` that is returned should be treated as "read-only", and should not be modified by the client application in any way. Typically, the `SpStreamProjection` objects are passed into the `SpObserver`'s constructor, giving the `SpObserver` the list of fields and their corresponding data types. This information is used by the `SpObserver` to process the updates that come back from the server.

See [Section 5.1, "Aleri SQL Queries and Statements"](#) for SQL query handling limitations of the Sybase Aleri Streaming Platform.

You must create your own `SpObserver` objects, which are notified by the `SpSubscriptionProjection` with the updates that arrive from the Sybase Aleri Streaming Platform. You can create `SpObserver` objects by implementing the `SpObserver` interface.

The `addObserver(SpObserver theObserver)` method is used to register an `SpObserver` with the `SpSubscriptionProjection` object.

The `addObserver(...)` call returns an integer value that represents a "cookie"/"handle" to the registered `SpObserver` object. Later on, the client application can use the cookie to remove the `SpObserver`.

You can use the `removeObserver(int theCookie)` method to remove the `SpObserver` from the `SpSubscriptionProjection`'s delivery mechanism.

A.1.12. SpSubscriptionEvent

An `SpSubscriptionEvent` provides the following method set:

```
public String getSubName();  
public int getType();  
public String getTypeName();  
public int getId();  
public String getIdName();  
public int getStreamId();  
public int getStreamOpCode();  
public Collection getData();
```

The `getSubName()` method returns the name of the subscription object that generated and delivered this event to the `SpObserver`. This name was assigned to the subscription object when it was created through the `SpPlatform.createSubscription(...)` method.

The `getType()` method returns an integer representing the "type" of this `SpSubscriptionEvent`. There are four types of events defined in the `SpSubscriptionEvent.java` interface:

- `SpSubscriptionEvent.EVTYPE_PARSED_DATA`

It is delivered from a `Subscription` object created with a delivery type of `SpSubscriptionCommon.DELIVER_PARSED`.

- `SpSubscriptionEvent.EVTYPE_BINARY_DATA`

It is delivered from a `Subscription` object created with a delivery type of `SpSubscriptionCommon.DELIVER_BINARY`.

- `SpSubscriptionEvent.EVTYPE_STREAM_OPCODE_DATA`

It is delivered from a `Subscription` object created with a delivery type of `SpSubscriptionCommon.DELIVER_STREAM_OPCODES`.

- `SpSubscriptionEvent.EVTYPE_SYSTEM`

It is delivered from a `Subscription` object, indicating a system event has occurred. The event could be an error, a halt in communication, the shutting down of the Sybase Aleri Streaming Platform, and so forth.

The `getTypeName()` method returns the type of event as a string, as opposed to the internal integer representation. The client application can use this for output messages.

An integer is returned by the `getId()` method that uniquely identifies what actually happened on the Sybase Aleri Streaming Platform. The event IDs are unique across the entire set of event types. For example, an `SpSubscriptionEvent` may have a “type” of `SpSubscriptionEvent.EVTYPE_SYSTEM`, which means it is a system-related notification. The `getId()` method returns what was actually detected by the system (for example, `SpSubscriptionEvent.EVID_PARSING_ERROR`, `SpSubscriptionEvent.EVID_COMMUNICATOR_HALTED`, and so forth). As with the event types, all of the event IDs are enumerated within the `SpSubscriptionEvent` interface.

The following describes the `SpSubscriptionEvent` identifiers:

- `SpSubscriptionEvent.EVID_GATEWAY_SYNC_START`

It is delivered to the `SpObserver` if the subscription is sent a `START_SYNC` Gateway I/O message from the Sybase Aleri Streaming Platform. The `START_SYNC` message contains the ID for the stream with its associated message.

It indicates the start of the stream's “snapshot”. Following this event are `INSERT` messages for each record in the stream until the `END_SYNC` Gateway I/O message is received from the Sybase Aleri Streaming Platform. A call to the `START_SYNC` event's `getData()` method returns null. For this message to be sent from the Sybase Aleri Streaming Platform to the client application, the subscription has to be created with the `SpSubscriptionCommon.BASE` flag specified. If the `SpSubscriptionCommon.NOBASE` flag is specified instead, the `START_SYNC` message would never have been delivered from the Sybase Aleri Streaming Platform to the client application.

This event is also delivered after a dynamic change, if the stream's contents gets regenerated after the `WIPEOUT` event. In this situation, the `START_SYNC` event is delivered even if the subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- `SpSubscriptionEvent.EVID_GATEWAY_SYNC_END`

It is delivered to the `SpObserver` if the subscription is sent an `END_SYNC` Gateway I/O message

from the Sybase Aleri Streaming Platform. The `END_SYNC` message contains the ID for the stream with which the message is associated.

It indicates that the end of the stream's "snapshot" has been reached. A call to the `END_SYNC` event's `getData()` method returns null. For this message to be sent from the Sybase Aleri Streaming Platform to the client application, the subscription has to be created with the `SpSubscriptionCommon.BASE` flag specified. If the `SpSubscriptionCommon.NOBASE` flag is specified instead, the `END_SYNC` message would never be delivered from the Sybase Aleri Streaming Platform to the client application.

This event is also delivered after a dynamic Sybase Aleri Streaming Platform change, if the stream's contents gets regenerated. On the dynamic regeneration, the `WIPEOUT` event is followed by the `START_SYNC` event, insertion of the new data, and the `END_SYNC` event. In this situation all the events are delivered even if the subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- `SpSubscriptionEvent.EVID_GATEWAY_WIPEOUT`

It is delivered to the `SpObserver` after dynamic Sybase Aleri Streaming Platform changes, if the stream's contents gets regenerated. It means that the whole current contents of the stream is being discarded. The `WIPEOUT` event is followed by the `START_SYNC` event, insertion of new data, and the `END_SYNC` event. In this situation, all the events are delivered even if the subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- `SpSubscriptionEvent.EVID_BINARY_DATA`

If the subscription is created with the delivery type of `SpSubscriptionCommon.DELIVER_BINARY`, the `getData()` method returns a `ByteBuffer` object containing the binary message delivered from the Sybase Aleri Streaming Platform.

- `SpSubscriptionEvent.EVID_PARSED_FIELD_DATA`

When a `Subscription` object is created with a delivery type of `SpSubscriptionCommon.DELIVER_PARSED`, it attempts to parse the field data of the stream messages transmitted by the Sybase Aleri Streaming Platform and delivers this parsed field information to the `SpObserver`. The `getStreamOpCode()` method can be used to determine whether the message was an `INSERT`, `UPDATE`, or `DELETE`. The parsed field data is accessed by the `SpObserver` through the event's `getData()` method.

- `SpSubscriptionEvent.EVID_PARSED_PARTIAL_FIELD_DATA`

It is similar to the `SpSubscriptionEvent.EVID_PARSED_FIELD_DATA` event described previously. This event is delivered when the `Subscription`'s message parser detects an error in the middle of parsing out the field data for the message sent from the Sybase Aleri Streaming Platform. If an error is encountered during the parse, only those fields that were successfully parsed, up to the place where the error was detected, will be delivered in this event. The `SpObserver` is not obligated to inspect the partial results; however, the application programmer may want to use the partial results for debugging purposes.

- `SpSubscriptionEvent.EVID_COMMUNICATOR_HALTED`

It is delivered when the client application attempts to issue a "shutdown" through the `SpPlatform` object. A call to the `getData()` method returns null.

- `SpSubscriptionEvent.EVID_PLATFORM_SHUTDOWN`

This event is delivered when the client application attempts to issue a "shutdown" through the `SpPlatform` object. A call to the `getData()` method returns null.

- `SpSubscriptionEvent.EVID_PARSING_ERROR`

This event is delivered when a parsing error is detected by the `Subscription`, and there is at least some context to report on. A call to the `getData()` method returns an object of type `SpParserReturnInfo`.

See [Section 2.3.3.2, “Inspect Parsing Errors”](#), for more information.

- `SpSubscriptionEvent.EVID_UNKNOWN_PARSING_ERROR`

This error indicates that the parser encountered an unexpected error before the completion of the process. In this case, the `getData()` method returns an integer object that contains the record length for the message that is about to be parsed.

- `SpSubscriptionEvent.EVID_READ_STREAM_RECORD_ERROR`

It indicates that the parser could not successfully read the record that was delivered from the Sybase Aleri Streaming Platform. The `getData()` method returns an integer object containing the value of the record length that was read for the bad record.

- `SpSubscriptionEvent.EVID_BAD_RECORD_LENGTH_ERROR`

It shows that the record length read from the socket was erroneous. The `getData()` method returns an integer object that contains the bad record length value read from the socket.

- `SpSubscriptionEvent.EVID_BAD_GATEWAY_OP_CODE_ERROR`

It indicates that the Gateway I/O operation code for the message sent from the Sybase Aleri Streaming Platform is invalid. The `getData()` method returns an integer object that contains the bad Gateway I/O operation code that was read.

- `SpSubscriptionEvent.EVID_HOT_SPARE_SWITCH_OVER_INITIATED`

It will be delivered to the `SpObserver` when the Pub/Sub API recognizes that a connection attempt should be made to the High Availability (Hot Spare) server. The High Availability connection parameters were specified in the `SpPlatformParms` object passed to the `SpFactory::createPlatform()` method when the underlying `SpPlatform` was first created.

- `SpSubscriptionEvent.EVID_HOT_SPARE_SWITCH_OVER_SUCCEEDED`

It is delivered to the `SpObserver` when the attempt to connect to the Hot Spare server is successful.

- `SpSubscriptionEvent.EVID_HOT_SPARE_SWITCH_OVER_FAILED`

It is delivered to the `SpObserver` when the attempt to connect to the Hot Spare server fails.

The `getIdName()` method returns the string representation of the numeric event ID. The client application program can use this for output messages.

The `getStreamId()` method returns the stream ID that is associated with this event. For example, an `SpObserver` may receive an event of type `SpSubscriptionEvent.EVTYPE_PARSED_DATA`, where the event ID is `SpSubscriptionEvent.EVID_PARSED_FIELD_DATA`, indicating that the event contains parsed field data. The `getStreamId()` method returns the stream ID to which this event data corresponds.

The `getStreamOpCode()` method returns the stream operation code that is associated with this event. For example, an `SpObserver` may receive an event of type `SpSubscription-`

`tionEvent.EVTYPE_PARSED_DATA`, where the event ID is `SpSubscriptionEvent.EVID_PARSED_FIELD_DATA`, indicating that the event contains parsed field data. The `getStreamOpCode()` method returns a value that indicates whether the event is an *INSERT*, *UPDATE*, *DELETE*, or *UPSERT*.

The `getData()` method returns a vector of objects containing the event data. The objects stored in the collection depend upon the delivery type (which was specified when the subscription object was created). For example, if the delivery type of the subscription is `SpSubscriptionCommon.DELIVER_PARSED`, the `getData()` method returns a vector in which each element is another vector, each of which contains the field list of parsed objects produced by the subscription message parser.

If the Sybase Aleri Streaming Platform delivers a non-transaction message (for example, an isolated *INSERT* or *DELETE* message) to the subscription, the `getData()` method returns a vector that has a size (number of elements) of 1. If the Sybase Aleri Streaming Platform delivers a transaction message to the Subscription, the `getData()` method returns a vector whose size (number of elements) is equal to the number of stream update records within the transaction message sent from the Sybase Aleri Streaming Platform. This is why there are two “levels”/“dimensions” of collections (vectors) within the `SpSubscriptionEvent.EVID_PARSED_FIELD_DATA` event.

It is also important to note that each transactional message sent from the Sybase Aleri Streaming Platform contains updates for one stream. A transaction cannot contain records within it for two or more separate streams.

When the subscription is created using a delivery type of `SpSubscriptionCommon.DELIVER_BINARY`, the `getData()` method returns a `ByteBuffer` that has the raw binary stream message within it. When the Sybase Aleri Streaming Platform sends a transaction block, the `ByteBuffer` contains the entire transaction block.

When the `SpSubscription` is created using a delivery type of `SpSubscription.DELIVER_STREAM_OPCODES`, the `getData()` method returns null. Use the `getStreamOpCode()` method to determine the stream operation code (*INSERT*, *UPDATE*, *DELETE*, or *UPSERT*).

A.1.13. SpParserReturnInfo

The `SpParserReturnInfo` object referenced above has the following method set that you can use within the `SpObserver`:

```
public int getErrorCode();
public String getErrorMessage();
public int getTransMessageIndex(); // REL 0
public int getColumnIndex(); // REL 0
public String getErrorData();
public boolean isSuccess();
```

Details:

- The `getErrorCode()` and `getErrorMessage()` methods return the parser error code and associated error message, respectively.
- The `getTransMessageIndex()` method is only relevant for transaction messages that are being parsed at the time of error. If the message is not a transaction, -1 is returned. The transaction block's message index is rel 0 (message one has an index value of zero) if the message is a transaction block.
- The `getColumnIndex()` method returns the column index (rel 0) for the column where the

parsing error is detected. If the error occurred before the parser event got to the first column (having a column index value of zero), this method will return `-1`.

- The `getErrorMessage()` method returns a string that the parser may have put together at the point of error to indicate what went wrong. Typically, if there was any extra error information that the parser determined was important, it is stored in a string is returned by `getErrorMessage()`.
- The `isSuccess()` method returns `true` if the parse was successful and `false` if there was an error.

A.1.14. SpNullConstants

The `SpNullConstants` class defines the following objects:

```
public static Integer nullInteger16 = new Integer(0)
public static Integer nullInteger32 = new Integer(0)
public static Long nullLong = new Long(0)
public static Double nullMoney = new Double(0)
public static Double nullDouble = new Double(0)
public static Date nullDate = new Date(0)
public static Date nullTimestamp = new Date(0)
public static String nullString = "NULL"
```

Details:

- The `nullInteger16` object represents a null 16-bit Sybase Aleri Streaming Platform integer value.
- The `nullInteger32` object represents a null 32-bit Sybase Aleri Streaming Platform integer value.
- The `nullLong` object represents a null 64-bit Sybase Aleri Streaming Platform integer value.
- The `nullMoney` object represents a null Sybase Aleri Streaming Platform money value.
- The `nullDouble` object represents a null Sybase Aleri Streaming Platform double value.
- The `nullDate` object represents a null Sybase Aleri Streaming Platform date value.
- The `nullTimestamp` object represents a null Sybase Aleri Streaming Platform timestamp value.
- The `nullString` object represents a null Sybase Aleri Streaming Platform string value.

A.1.15. SpUtils

There are a few miscellaneous classes that were briefly referenced in some of the earlier examples. One class is `SpUtils`, which offers the following utility methods:

```
public static String getErrorMessage(int errorCode)
public static String getEventTypeName(int eventType)
public static String getEventIdName(int eventId)
```

Details:

- The `getErrorMessage(int errorCode)` method retrieves the `String` message associated with the `errorCode` passed in through the parameter list. The error messages are stored in the `pubsub.properties` file, which must be present in the Java classpath for this method to work. Typically, the `errorCode` is returned by a previous call to one of the Pub/Sub API methods.
- The `getEventTypeName(int eventType)` method is typically called by an `SpObserver` object. This method returns a string representation of the `eventType` passed in. The event type names are also stored in the `pubsub.properties` file (refer to the `getErrorMessage(int errorCode)` method description). The `eventType` is usually retrieved from a `SpSubscriptionEvent` object delivered to the `SpObserver` object through the `SpObserver's notify(...)` method.
- The `getEventIdName(int eventId)` method is also typically called by an `SpObserver` object. It returns the string representation of the `eventId` passed in. The `eventId` names are also stored in the `pubsub.properties` file (refer to the `getErrorMessage(int errorCode)` method description). The `eventId` is usually retrieved from an `SpSubscriptionEvent` object delivered to the `SpObserver` through the `SpObserver object's notify(...)` method.

Another class referenced in some of the previous examples is `SpGatewayConstants`. It stores a group of constant values used in Gateway related methods/activities within the Pub/Sub API. For example, the `SpGatewayConstants` class stores the stream op codes, flag values, and so forth.

A.2. Objects for Publication

If the `sp.createPublication(...)` call is successful, the client application program gets an `SpPublication` object back from the Sybase Aleri Streaming Platform. An `SpPublication` object can be used to publish data to one or more streams. It implements an interface that provides the following method set:

```
public String getName();

public int start();

public int publish(SpStreamDataRecord streamRecord);

public int publish(Collection streamRecords,
    int streamOpCodeOverride,
    int streamFlagOverride);

public int publishTransaction(
    Collection streamRecords,
    int streamOpCodeOverride,
    int streamFlagsOverride,
    int maxRecordsPerBlock);

public int publishEnvelope(
    Collection streamRecords,
    int streamOpCodeOverride,
    int streamFlagsOverride,
    int maxRecordsPerBlock);

public int commit();

public int stop();
```

Details:

The `getName()` method returns the name assigned to this `SpPublication` object when it was created through the `SpPlatform`'s `createPublication(...)` factory method.

The `start()` method is used to start the publication process.

The `publish(SpStreamDataRecord streamRecord)` method is used to publish/send a single stream input record to a source stream on the Sybase Aleri Streaming Platform. It is more efficient to send Input Stream records to the Sybase Aleri Streaming Platform in batches known as “transactions”. However, in some cases, you may want to publish one record at a time — for example, when testing a new data model. If this call is successful, a return code of zero is sent back to the caller. Otherwise, an error code is sent back. The `SpUtils.getErrorMessage(errorCode)` method can be called to get the specific error message.

Each of the `publish` methods of the `SpPublication` object takes one or more `SpStreamDataRecord` objects as input. The `SpStreamDataRecord` object represents one row of stream data that is to be sent to the Sybase Aleri Streaming Platform. Each `SpStreamDataRow` object has an “op code” which indicates how the row is to be handled by the Sybase Aleri Streaming Platform when it is received. For example, the “op code” may indicate that the row is to be treated as an *INSERT*, a *DELETE*, an *UPDATE*, and so forth.

The `publish(Collection streamRecords, int streamOpCodeOverride, int streamFlagOverride)` method sends a vector of `SpStreamDataRecord` objects to the Sybase Aleri Streaming Platform with one call. The *streamOpCodeOverride* and *streamFlagOverride* parameters can be used to override the corresponding values found in the individual `SpStreamDataRecord` objects that comprise the collection.

Although the method takes in a collection of stream data records, these records are nevertheless published to the Sybase Aleri Streaming Platform one at a time. However, this method allows you to create a set of stream records in which each record is published to a different stream. It may be useful in debugging or testing scenarios, where the ordered sequence of updates to various source streams is important.

`publishTransaction(Collection streamRecords, int streamOpCodeOverride, int streamFlagsOverride, int maxRecordsPerBlock)` and `publishEnvelope(Collection streamRecords, int streamOpCodeOverride, int streamFlagsOverride, int maxRecordsPerBlock)` are the most efficient publication methods, because they bundle multiple `SpStreamDataRecords` and send them to the Sybase Aleri Streaming Platform as a single batch.

There is a restriction/constraint related to publishing transactions and/or envelopes to the Sybase Aleri Streaming Platform. Each `SpStreamDataRecord` that is to be placed within the transaction/envelope must be for the same source stream. However, each `SpStreamDataRecord` can have a different “op code” (such as *INSERT*, *UPDATE*, *DELETE* or *UPSERT*).

As mentioned previously, the override op code and flag values can be used to override the corresponding values found within the individual `SpStreamDataRecord` objects that make up the collection. Additionally, it takes a parameter called *maxRecordsPerBlock* — an integer value that specifies the maximum number of `SpStreamDataRecords` to be sent as a transactional/envelope unit to the Sybase Aleri Streaming Platform. If the value is set to zero, then the method will try to send all of the `SpStreamDataRecord` objects in the collection within one transaction block. If *maxRecordsPerBlock* is less than the actual number of records in the collection, the record set will be broken up into multiple transactions/envelopes during the transmission to the server.

The difference between a transaction and envelope block transmitted to the server is how the server processes the block of records upon receiving it. As the name implies, a group of records within a transaction block is treated as a single transactional unit on the server side. In the case of an envelope, the group

of records contained within the envelope are processed one record at a time by the server. Basically, the envelope mechanism allows the client to send a batch of records for a specific stream to the server in one shot, as opposed to sending a single record at a time and having to wait for each record's *ack* or *nak* reply from the server.

The `commit()` method issues a special Gateway I/O command to the Sybase Aleri Streaming Platform, requesting that all pending input records previously sent to the Sybase Aleri Streaming Platform be synced to disk. Making a commit call is a tremendously expensive operation relative to latency. It's designed to be used only as part of a two-phase commit process when reading from a persistent source, such as an ActiveMQ series, and writing to a Sybase Aleri Streaming Platform instance that uses Sybase's log store persistence model.

In a real time, low latency streaming scenario, the commit call should not be used after each record.

Typically the commit call should be used as follows: For a standard two-phase commit process that guarantees against data loss, the client reads messages from the source, such as ActiveMQ, and publishes to the Sybase Aleri Streaming Platform until reaching a pre-determined number of processed messages (>1024 is recommended) or a specified amount of time has elapsed. After reaching the value set for the maximum number messages or the elapsed time has passed, the `commit()` call is made and upon return to the client, the client may inform the source, such as ActiveMQ, that the messages can be deleted.

The `stop()` method shuts down the underlying Gateway I/O socket connection.

A.2.1. Stream Operation Codes

The stream operation codes that can be set for each individual `SpStreamDataRecord` (or as one of the *streamOpCodeOverride* parameters) can be found in the `SpGatewayConstants` interface, and are as follows:

- `SpGatewayConstants.SO_NOOP`

When specified as the “streamOpCodeOverride” parameter, it instructs the Sybase Aleri Streaming Platform to use the stream op code stored in each of the individual `SpStreamDataRecord` objects. If the stream op code within the `SpStreamDataRecord` is set to `SpGatewayConstants.SO_NOOP`, the Sybase Aleri Streaming Platform is instructed to use the default stream operation, which is INSERT.

- `SpGatewayConstants.SO_INSERT`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as an INSERT operation.

- `SpGatewayConstants.SO_UPDATE`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as an UPDATE operation.

- `SpGatewayConstants.SO_DELETE`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as a DELETE operation.

- `SpGatewayConstants.SO_UPSERT`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as an UPSERT operation. An UPSERT operation either inserts the stream record into the source stream if it is not already present, or updates the existing source stream record using the contents of

the stream record.

A.2.2. Stream Flag Values

The set of stream flag values can be for each individual `SpStreamDataRecord` or one of the `streamFlagOverride` parameters. It can be found in the `SpGatewayConstants` interface and includes:

- `SpGatewayConstants.SF_NULLFLAG`

When specified as the `streamFlagOverride` parameter, it instructs the Sybase Aleri Streaming Platform to use the stream flag stored in each of the individual `SpStreamDataRecord` objects. If the stream flag within the `SpStreamDataRecord` is set to `SpGatewayConstants.SF_NULLFLAG`, the default synchronous publication sequence will take place.

- `SpGatewayConstants.SF_NOACK`

When specified, it tells the Sybase Aleri Streaming Platform not to send an ack or nak back to the client application that issued the publication request. The `publish` method returns immediately. The caller is not notified of potential transmission or publication failures.

- `SpGatewayConstants.SF_SHINE`

It is only relevant for the stream op code values of `SO_UPDATE` and `SO_UPSERT`. Typically, when an existing record is updated by the Sybase Aleri Streaming Platform, all fields will be assigned new values. However, there are cases when some of the fields within the stream record must be updated while others are not. The client application can set the “other” fields of the stream record being published to the Sybase Aleri Streaming Platform to null; if the `SF_SHINE` flag is set, the Sybase Aleri Streaming Platform will ignore the nulls and leave the existing field values. The key fields must always be present, as they are required to locate the record.

In essence, the Sybase Aleri Streaming Platform lets the existing field values “shine through” for each of the null values sent in from the client application program.

The flag values represent bits that can be ORed together. For example:

```
int flags = SpGatewayConstants.SF_SHINE | SpGatewayConstants.SF_NOACK;
```

A.2.3. SpStreamDataRecord Object

Each of the `SpPublication` object's publishing methods sends stream input data from the client application to the Sybase Aleri Streaming Platform. Each stream record (or row of stream data) is encapsulated within an `SpStreamDataRecord` object, which has the following method set:

```
public SpStream getStream();
public Collection getFieldData();
public int getOpCode();
public int setOpCode(int value);
public int getFlags();
public int setFlags(int value);
```

Details:

- The `getStream()` method returns an `SpStream` object associated with this `SpStreamDataRecord`.
- The `getFieldData()` method returns a vector of objects containing the data for each field in the stream record. Currently, the supported datatypes for these objects are Integer, Long, Date, and String. The object can also be null.
- The `getOpCode()` method returns the stream op code currently set for this record.
- The `setOpCode(int value)` method is used to set the value of the stream op code for this record.
- The `getFlags()` method returns the flag settings currently set for this record.
- The `setFlags(int value)` method is used to set the stream flag settings for this record.

A.2.4. Create `SpStreamDataRecord` Objects

For consistency within the Pub/Sub API, an `SpStreamDataRecord` object is created using yet another “Factory” method that has the following method signature:

```
SpStreamDataRecord SpFactory.createStreamDataRecord(  
    SpStream stream,  
    Collection fieldData,  
    int opCode,  
    int flags,  
    SpPlatformStatus status  
);
```

Details:

- *stream* is a reference to the `SpStream` object with which this new `SpStreamDataRecord` object will be associated. The client application can get this value through the appropriate `SpPlatform` method, such as `getStream(String streamName)` or `getStream(int streamId)`.
- *fieldData* is a vector of objects in which each entry matches the corresponding field data type, as indicated in the streams definition (specified in the *SpStream* parameter).

When creating an `SpStreamDataRecord`, all of the key fields must be assigned with non-null values within the *fieldData* collection/vector.

- *opCode* is the stream operation code associated with this `SpStreamDataRecord`.

The op code will inform the Sybase Aleri Streaming Platform how to apply this record to the source stream (in other words, *INSERT*, *UPDATE*, *DELETE*, and so forth).

You have the option to override the stream op code within several of the publishing methods.

- *flags* is the stream flag settings value associated with this `SpStreamDataRecord`.

You have the option to override the stream flag settings within several of the publishing methods.

- *status* is an object that stores error code information generated by the `createStreamDataRecord(...)` factory method if the `SpStreamDataRecord` object cannot be created.

The following code example shows how the client application program uses the `createStreamDataRecord(...)` factory method to create an `SpStreamDataRecord` object that can be published to the Sybase Aleri Streaming Platform:

```
/*
 * Source Stream is called "input", and has the following
 * record layout:
 *
 * int, string, double, date, int, string, double, date
 *
 */

Collection fieldData = new Vector(8);

fieldData.add(new Integer(104));
fieldData.add("do_mystring");
fieldData.add(new Double(5.7));
fieldData.add(new Date(01));
fieldData.add(new Integer(200));
fieldData.add("do_mystring2");
fieldData.add(new Double(8.9));
fieldData.add(new Date(01));

SpStream stream = sp.getStream("input");

/*
 * Use the createStreamDataRecord(...) factory method to
 * bundle up the stream, fieldData vector, stream op code,
 * and stream flags
 * into an SpStreamDataRecord object.
 *
 * At the moment, the SpStreamDataRecord object is the
 * basic unit of publication. You can publish these one
 * at a time, or you can publish them as a group (with
 * or without transaction blocks).
 *
 * NOTE: If you wish to publish a group of
 * SpStreamDataRecord objects
 * as a transaction, then all of the SpStreamDataRecords
 * within the group must belong to the same stream.
 */
SpStreamDataRecord sdr = SpFactory.createStreamDataRecord(
    stream,
    fieldData,
    SpGatewayConstants.SO_UPSERT,
    SpGatewayConstants.SF_NULLFLAG,
    status);

if (sdr == null)
{
    System.out.println("Could not createStreamDataRecord, status=" +
        status.getErrorCode());
    System.out.println("Error Message:" +
        status.getErrorMessage());
    return status.getErrorCode();
}
```

The client application can create a large number of `SpStreamDataRecord` objects, placing each of them within a common vector. Next, one of the `SpPublication`'s publishing methods can be used to send all rows of the stream data that are stored in the vector to the Sybase Aleri Streaming Platform,

either individually or using transactions.

Below is an example of how to publish a collection/vector of `SpStreamDataRecord` objects as a single transaction, where `sp` is an `SpPlatform` object that was previously instantiated and `streamInputData` is a vector that contains a number of `SpStreamDataRecord` objects.

```
/*
 * Create the publication object associated with the
 * platform.
 */
String name = "testPub_1";
SpPublication pub = sp.createPublication(name, status);
if (pub == null)
{
    System.out.println("Couldn't create a publication object, status=" +
        status);
    System.out.println("Error message = " + status.getErrorMessage());
    return status.getErrorCode();
}

/*
 * Start the publication object (this opens up a GW I/O
 * socket connection). Don't forget to eventually close
 * down the SpSubscription object (via the "stop()" method,
 * later on when you are finished using it,
 */

rc = pub.start();

if (rc != 0)
{
    System.out.println("Couldn't start the publication object.");
    System.out.println("Error message = " +
        SpUtils.getErrorMessage(rc));
    return rc;
}

/*
 * Publish the collection/vector of SpStreamDataRecord
 * objects as one big transaction.
 */

rc = pub.publishTransaction(streamInputData,
    SpGatewayConstants.SO_INSERT,
    SpGatewayConstants.SF_NULLFLAG,
    0);

if (rc != 0)
{
    System.out.println("Couldn't publish the transaction.");
    System.out.println("Error message = " +
        SpUtils.getErrorMessage(rc));
    return rc;
}
```

A.3. Objects for recording and playback

A.3.1. SpRecorder Object

You need to call the factory method, `createRecorder` defined in `SpPlatform`, to create an `SpRecorder` cli-

ent programs.

```
public SpRecorder createRecorder(String name, String filename, java.util.Collection streams,
                                long maxRecords, SpPlatformStatus status)
```

The method takes the following parameters:

- `name` String that will uniquely identify this instance of the recorder object
- `filename` Name of the file where recorded data will be stored
- `streams` a `java.util.Collection` instance that contains the name of the streams to record events for
- `flags` Flags that control the subscription. These flags are passed to the underlying subscription to the platform. Can be a bitwise OR of the following values
 - One of `SpSubscriptionCommon.BASE` or `SpSubscriptionCommon.NOBASE` - whether to record data already in streams at time of connection
 - `SpSubscriptionCommon.LOSSY` - whether platform should discard records if client application cannot keep up
- `maxRecords` maximum number of records to process
- `status` an `SpPlatformStatus` object to return information in case of error

`SpRecorder` has the following public interface

```
public String getName();
public int    start();
public long   getRecordCount();
public int    stop();
```

Details:

`getName()` returns the identifier assigned to this instance of the `SpRecorder` object
`start()` spawns a background thread which starts the recording process. The method returns once the thread is started. Returns 0 on success.
`getRecordCount()` returns the number of data records processed.
`stop()` stops the recording process by terminating the recording thread and closing connections to the platform. Returns 0 on success

A.3.2. SpPlayback Object

An `SpPlayback` object is created by calling the following factory method defined in `SpPlatform`.

```
public SpPlayback createPlayback(String name, String filename, double scale, long
```

The method takes the following parameters:

- `name` uniquely identifies this instance of `SpPlayback`.
- `filename` specifies the name of the file containing the recorded data.
- `scale` controls the rate of playback. Values -1 to 1 have no effect and the data is played back at the rate it was recorded. Values greater than 1 speed up playback by that factor, for example, a value of 2 will play back twice as fast. Values less than -1 slows down playback by the factor specified.
- `maxrecords` specifies the maximum number of records to playback.
- `status` is an `SpPlatformStatus` object used to return information in case of error.

`SpPlayback` has the following public interface:

```
public String      getName();
public void        setSendUpsert(boolean upsert);
public boolean     getSendUpsert();
public void        setTimeScaleRate(double scale);
public double      getTimeScaleRate();
public int         start();
public long        getNumRecordsPlayedBack();
public int         getPercentPlayedBack();
public int         stop();
```

Details:

`getName()` returns the identifier assigned to this instance of `SpPlayback` object
`setSendUpsert(boolean)` chooses whether to convert INSERT ops in the data to UPSERT
`getSendUpsert()` returns the current setting of UPSERT flag
`setTimeScaleRate(double)` is a double to control the rate of playback
`getTimeScaleRate()` returns the current value of the scale factor
`start()` spawns a background thread that starts the playback process. Returns 0 on success
`getNumRecordsPlayedBack()` returns the number of data records played back so far
`getPercentPlayedBack()` returns the percentage of the data played back so far
`stop()` terminates the background playback thread and closes connections to the Sybase Aleri Streaming Platform

Appendix B. Reference Guide to the C++ Object Model

B.1. C++ Objects for Subscription

B.1.1. SpFactory Object

The SpFactory object is used by the client code to create the set of objects required to use/control the Pub/Sub API. The SpFactory interface includes the following methods:

```
static int init();
static int dispose();
static SpPlatform *createPlatform(SpPlatformParms *parms,
    SpPlatformStatus *status);
static SpPlatformParms *createPlatformParms(const char * theHost,
    int thePort, const char * theUser, const char * thePassword,
    bool theEncryptedFlag);
static SpPlatformParms *createPlatformParms(const char * theHost,
    int thePort, const char * theUser, const char * thePassword,
    bool theEncryptedFlag, bool theUseRsaFlag);
static SpPlatformParms *createPlatformParms(const char * theHost,
    int thePort, const char * theUser, const char * thePassword,
    bool theEncryptedFlag, const char * theHotSpareHost, int theHotSparePort);
static SpPlatformParms *createPlatformParms(const char * theHost,
    int thePort, const char * theUser, const char * thePassword,
    bool theEncryptedFlag, bool theUseRsaFlag, const char * theHotSpareHost,
    int theHotSparePort);
static SpPlatformParms *createPlatformParms(const char * theHost, int thePort,
    const char * theUser, const char * thePassword, bool theEncryptedFlag,
    SpPlatformParms::auth_type theAuth, const char * theHotSpareHost,
    int theHotSparePort);
static SpPlatformStatus *createPlatformStatus();
static SpStreamDataRecord *createStreamDataRecord(SpStream *stream,
    std::vector<SpDataValue *> *fieldData, int opCode, int flags,
    SpPlatformStatus *status);
```

Where:

- The `init()` method is used to set up the XMLRPC global environment variables. According to the XMLRPC documentation, this method should be called while the application is single threaded.
- The `dispose()` method is used to tear down the XMLRPC global environment variables that were previously setup by the `init()` call. According to the XMLRPC documentation, this method should be called while the application is single threaded.
- The `createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method returns a pointer to an SpPlatform object if the Pub/Sub API was able to connect to the Sybase Aleri Streaming Platform and initialize properly.

Before calling this method, you have to use one of the overloaded `SpFactory::createPlatformParms(...)` methods and the `SpFactory::createPlatformStatus()` method to create the two parameters required by the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method. The contents of the *SpPlatformParms* parameter control how the connection and authentication from the Pub/Sub API to the Sybase Aleri Streaming Platform takes place. See [Section A.1.2, “SpPlatformParms Object”](#) for more information. If the connection to the Sybase Aleri Streaming Platform can not be established, the `createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method returns null, and a non-zero error code is set within the SpPlatformStatus object See [Section A.1.3, “SpPlatformStatus Object”](#) for information on how to retrieve the error code/message.

- The `createPlatformParms(const char * theHost, int thePort, const char * theUser, const char * thePassword, bool theEncryptedFlag)` method returns a pointer to an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method. This `createPlatformParms` method call sets up for basic connectivity. The user name/password are for authentication. If *theEncryptedFlag* is set to *true*, then https will be used to connect to the Sybase Aleri Streaming Platform's Command and Control process and SSL socket connections will be made to the Sybase Aleri Streaming Platform's Gateway I/O process. If *theEncryptedFlag* is set to *false*, then http will be used to connect to the Sybase Aleri Streaming Platform's Command and Control process regular (non-SSL) socket connections will be made to the Sybase Aleri Streaming Platform's Gateway I/O process.
- The `createtPlatformParms(const char * theHost, int thePort, const char * theUser, const char * thePassword, bool theEncryptedFlag, bool theUseRsaFlag)` method returns a pointer to an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method. In addition to the basic connectivity parameters mentioned above, this method adds an additional bool flag called *theUseRsaFlag*. If this flag is set to *true*, the Pub/Sub API will attempt to authenticate to the Streaming Processor using the RSA mechanism. To use this mechanism, the Sybase Aleri Streaming Platform must be started with the `-k` option, which indicates the directory where the public RSA key file is stored. See the *Utilities Guide* for more details about key generation and placement.

When using the RSA authentication mechanism, the password of the `SpPlatformParms` object must specify your private RSA key file. For example, if a user was named `foo`, there would be two RSA key files having the names `foo` containing the public RSA key for user `foo`) and `foo.private.der`, which contains the private RSA key for user `foo` in DER format. The public RSA key file called `foo` must be placed in a directory that is specified by the `-k` option to the Sybase Aleri Streaming Platform during startup.

The private RSA key file `foo.private.der` must be placed on the client machine using the Pub/Sub API to connect to the server. It is specified using the password parameter of the `createPlatformParms(...)` method.

There are five variations of the `createPlatformParams` method. All accomplish the creation of an `SpPlatformParams` object:

- basic
- basic with *UseRSA* flag
- basic with *HotSpare*
- *HotSpare* with *UseRSA*
- Kerberos authentication with or without the *hotspare*

Choose the method which suits your needs.

- The `createPlatformParms(const char * theHost, int thePort, const char * theUser, const char * thePassword, bool theEncryptedFlag, const char * thehotSpareHost, int thehotSparePort)` method returns a pointer to a `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method. In addition to the basic connectivity parameters previously mentioned, this method adds two more parameters called *const char * theHotSpareHost* and *int theHotSparePort*. Using an `SpPlatformParms` object created with this factory method

will cause the Pub/Sub API to use a High Availability configuration. In a High Availability configuration, if the primary Sybase Aleri Streaming Platform goes down, the Pub/Sub API automatically attempts to switch over and use the secondary Sybase Aleri Streaming Platform.

See the *Administrators Guide* for more information on High Availability configurations.

- The `createPlatformParms(const char * theHost, int thePort, const char * theUser, const char * thePassword, bool theEncryptedFlag, bool theUserRsaFlag, const char * theHotSpareHost, int theHotSparePort)` method returns a pointer to an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method. This can be any one of the following values: `SpPlatformParms::AUTH_NONE`, `SpPlatformParms::AUTH_PAM`, `SpPlatformParms::AUTH_RSA`, `SpPlatformParms::AUTH_KERBV5`. While other versions of the factory method can be used, this is the preferred way of creating an `SpPlatformParms` object. If a Hotspare configuration doesn't exist, clients should pass in a null value for the **theHotSpareHost** parameter.
- The `createPlatformStatus()` method returns a pointer to an `SpPlatformStatus` object that is passed as the second parameter to the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method, in order to return status information back to the caller. It is also used in several other methods within the Pub/Sub API that need to return error code/status information. See [Section A.1.3, “SpPlatformStatus Object”](#) for more information.

The `SpPlatformStatus` object is passed in as the last parameter. If `createStreamDataRecord` fails, a null will be returned to the caller and the `SpPlatformStatus` object will indicate the error condition.

- The `createStreamDataRecord(SpStream *stream, std::vector<SpDataValue *> *fieldData, int opCode, int flags, SpPlatformStatus *status)` method returns a pointer to an `SpStreamDataRecord` object that is used in the `SpPublication` object in order to publish data to the Streaming Processor. See

B.1.2. SpPlatformParms Object

The `SpPlatformParms` object is used by the `SpFactory::createPlatform(SpPlatformParms *parms, SpPlatformStatus *status)` method to create the `SpPlatform` object. The `SpPlatformParms` object is created using one of the overloaded `SpFactory::createPlatformParms(...)` methods previously described. The `SpPlatformParms` object contains all of the connection information required by the `SpPlatform` object, in order to make the appropriate connection(s) to the Sybase Aleri Streaming Platform's Command and Control Process, username, password, and flags indicating whether to use encryption, or RSA authentication, or Kerberos authentication and/or the High Availability (Hot Spare) mechanism. The `SpPlatformParms` interface includes the following methods:

```
std::string getHost();
std::string getGatewayHost();
int getPort();
std::string getUser();
std::string getPassword();
bool isEncrypted();
std::string getHotSpareHost();
int getHotSparePort();
bool useRsa();
SpPlatformParms::auth_type getAuthentication();
```

```
void setGatewayHost(const char * host);
```

Details:

- The `getHost()` method returns a string indicating the host name of the machine that the Streaming Processor's Command and Control process is running on.
- The `getGatewayHost()` method returns the name of the gateway machine if it has been explicitly set by the user.
- The `setGatewayHost()` method sets the gateway machine which the API will connect to. If set, the API will ignore the value returned from the Sybase Aleri Streaming Platform. This can be useful if the Sybase Aleri Streaming Platform is running on a machine without Domain Name System (DNS) entries.
- The `getPort()` returns an integer indicating the port number of the Sybase Aleri Streaming Platform's Command and Control process.
- The `getUser()` method returns a string indicating the username used to authenticate to the Sybase Aleri Streaming Platform.
- The `getPassword()` method returns a string indicating the password used to authenticate to the Sybase Aleri Streaming Platform. For RSA authentication, the *password* parameter contains the file name of your private RSA key file.
- The `isEncrypted()` method returns a boolean indicating whether encrypted connections will be used to the Command and Control process and the Gateway I/O process. If the encryption mechanism is enabled, the Command and Control process connection will be made using https, while the Gateway I/O process will make SSL socket connections.
- The `getHotSpareHost()` method returns a string indicating the host name of the secondary High Availability Sybase Aleri Streaming Platform. See the *Administrators Guide* for setting up a High Availability configuration.
- The `getHotSparePort()` method returns an integer indicating the port number of the secondary High Availability Streaming Processor. See [Section 2.4.6, "Publication/Subscription in a High Availability \(Hot Spare\) Configuration"](#) for more information.
- The `useRsa()` method returns a boolean indicating whether RSA authentication will be used when attempting to make connections to the Sybase Aleri Streaming Platform Command and Control process, and the Gateway I/O process.
- The `getAuthentication()` method returns the authentication mechanism specified when the `SpPlatformParms` was created.

B.1.3. SpPlatformStatus Object

The `SpPlatformStatus` object is used by several of the Pub/Sub API methods to return status information back to the caller. The `SpPlatformStatus` interface includes the following methods:

```
int getErrorCode();  
std::string getErrorMessage();  
bool isError();
```

where:

- The `getErrorCode()` method returns an integer. If a problem was detected by the method this `SpPlatformStatus` object was passed into, a non-zero error return code value is returned, otherwise a zero is returned to indicate success.
- The `getErrorMessage()` returns a string containing the error message text.
- The `isError()` method returns a boolean the value of which is `true` if an error was detected or `false` if no error was detected.

B.1.4. SpPlatform Object

The notion of the “Sybase Aleri Streaming Platform” has been abstracted into an object of type `SpPlatform`. As described in [Section 3.2.1, “Set Up Objects for SP Subscription in C++”](#), an `SpPlatform` object is created using the `SpFactory::createPlatform(...)` method. Once instantiated, an `SpPlatform` object implements and offers you the following Sybase Aleri Streaming Platform functionality:

```
std::string getUrl();
std::string getUser();
std::string getPassword();
std::string getHost();
std::string getGatewayHost();
std::string getXMLModelVersion();
int getPort();
int getGatewayPort();
int getDateSize();
int getAddressSize();
int getQuiesced();
int getPrimaryServerFlag();
std::vector<SpStream*> *getBaseStreams();
std::vector<SpStream*> *getDerivedStreams();
std::vector<SpStream*> *getStreams();
SpStream *getStream(const char *streamName);
SpStream *getStream(int streamId);

SpStreamDefinition *getStreamDefinition(const char * streamName);

SpStreamDefinition *getStreamDefinition(int streamId);
bool isBigEndian();
bool isConnected();
bool isEncrypted();
bool useRsa();
int shutdown();

std::string getConfig(SpPlatformStatus * status);
int loadServerConfigFile(const char * configFile, const char * flags);
int loadConfigString(const char * configString, const char * flags);
int loadConfigStringApplyingConversion(const char * configString, const char * flags);

int addStreamToClient(int clientHandle,
const char *streamName);

int removeStreamFromClient(int clientHandle,
const char * streamName);

SpSubscription *createSubscription(const char * name,
int flags, int deliveryType,
```

```
SpPlatformStatus *status);  
  
SpPublication *createPublication(const char * name,  
SpPlatformStatus *status);
```

Most of the `SpPlatform` object's methods communicate internally with the Sybase Aleri Streaming Platform Command and Control process through the XMLRPC protocol. The `SpPlatform` methods allow the client application program to retrieve Sybase Aleri Streaming Platform configuration information and retrieve all the source and/or derived streams.

Details of the method set:

The `getUrl()` method returns a string representing the URL which is used to connect to the Command and Control Process through XMLRPC. The context of this string depends on whether the `SpPlatform` object was created with encryption enabled. Refer to [Appendix F, Using Encryption with Java Client Applications](#) for more information. If there is an instance of the `SpPlatform`, the `isEncrypted()` method can be used to check whether encryption was enabled when the `SpPlatform` object was instantiated.

The `getUser()` and `getPassword()` methods return the strings that represent a username and password. These values are used internally for authentication when connecting to the Sybase Aleri Streaming Platform Command and Control and Gateway I/O processes.

There is a set of methods consisting of `getHost()`, `getGatewayHost()`, `getPort()`, and `getGatewayPort()`. `getHost()` returns the name of the host machine where the Sybase Aleri Streaming Platform Command and Control Process is running. `getGatewayHost()` returns the host machine where the Sybase Aleri Streaming Platform Gateway I/O Process is running. Currently, these two Sybase Aleri Streaming Platform processes reside on the same machine. However, this may change in the future.

The `getPort()` and `getGatewayPort()` methods return, respectively, the Command and Control port number and Gateway I/O port number but refer to two different processes.

The `getXMLModelVersion()` method returns a string indicating the AleriML data model version which started up the Sybase Aleri Streaming Platform.

The `getDateSize()` method returns the size of the *datetime* field type. If the Pub/Sub API is used to communicate with the Gateway I/O process, the *datetime* field type size is automatically fixed. If a different Gateway I/O code will be written, your application will have to deal with this, as well as endianness, when sending datetime fields to the Sybase Aleri Streaming Platform.

The `getAddressSize()` method returns the size of a C/C++ pointer (in bytes) that the Sybase Aleri Streaming Platform currently recognizes. The value represents how the instance of the running Sybase Aleri Streaming Platform was compiled (either 32-bit or 64-bit).

The `getQuiesced()` method returns an integer that represents the “quiesced” state of the Sybase Aleri Streaming Platform. If successful, the method will return either 0 to indicate *false*, or 1 to indicate *true*. If the command cannot be executed successfully, an error code is returned.

The error message associated with the error code can be retrieved by calling `SpUtils::getErrorMessage(rc)`, where `rc` is the return code sent back from the `getQuiesced()` call.

The `getPrimaryServerFlag()` method returns an integer. If a value of 1 is returned, the Sybase Aleri Streaming Platform is considered to be the primary server in a High Availability (Hot Spare) configuration. If a value of zero is returned, the Sybase Aleri Streaming Platform is not the primary server in this configuration. If the command could not be executed successfully, an error code is returned that

is neither zero nor 1.

The Pub/Sub API attempts to alleviate you from having to worry about the details of a High Availability (Hot Spare) switch over in case the primary server goes down. You can use this method to check that the connected Sybase Aleri Streaming Platform is indeed a primary server within a High Availability (Hot Spare) configuration. Theoretically, you could use the Pub/Sub API to establish a connection to a secondary server within this configuration. Calling `getPrimaryServerFlag()` on the secondary server returns a value of zero, indicating that the server is not a primary.

The next group of methods is used to return stream metadata from the Sybase Aleri Streaming Platform. A stream's metadata/schema is represented within the Pub/Sub API as an object of type `SpStream`. Refer to [Section B.1.5, “SpStream Object”](#) for more information. The `getBaseStreams()` method returns a pointer; these pointers reference `SpStream` objects representing all of the Source Streams residing on the Sybase Aleri Streaming Platform. Similarly, `getDerivedStreams()` returns a pointer to a vector of pointers. These pointers reference `SpStream` objects that represents all of the derived streams residing on the Sybase Aleri Streaming Platform. The `getStreams()` method returns a vector of `SpStream` objects that represents all streams (both source streams and derived streams) residing on the Sybase Aleri Streaming Platform. For a particular stream, you can look up the stream by its *name* or *id* using the `getStream(const char * streamName)` or `getStream(int streamId)` method, respectively.

The `getStream(const char * streamName)` and `getStreamDefinition(int streamId)` methods return an object of type `SpStreamDefinition` for the specified *streamName* or *streamId*, respectively. Refer to [Section B.1.6, “SpStreamDefinition Object”](#) for more information.

The `isBigEndian()` method returns `true` if the Sybase Aleri Streaming Platform Server is running on a big-endian machine, `false` if the Sybase Aleri Streaming Platform Server is running on a little-endian machine.

The `isConnected()` method returns `true` if the `SpPlatform` object is still connected to the Sybase Aleri Streaming Platform. Otherwise, it returns `false`. For example, if the client application program issues a “shutdown”, subsequent `isConnected()` calls return `false`.

Once an `SpPlatform` object is shut down, the application program should set its reference to null. Later on, another `SpPlatform` object can be instantiated again using the `SpFactory::createPlatform(...)` method.

The `shutdown()` method tells the Command and Control Process to shut down the Sybase Aleri Streaming Platform. This causes all socket connections to the Sybase Aleri Streaming Platform to be closed. If your subscriptions are running, the `SpObserver` objects of those subscriptions will be notified before the shutdown.

The `getConfig(SpPlatformStatus *status)` method returns a `std::string` containing the AleriML configuration currently being executed by the running Sybase Aleri Streaming Platform instance. If there is an error in retrieving the XML configuration information from the server, this method will return an empty string, and the error code will be stored in the `SpPlatformStatus` parameter passed into the method.

The `loadServerConfigFile(const char * configFile, const char * flags)` method will attempt to load the AleriML configuration file on the server into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information used during the AleriML configuration file load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string. Consult the *Administrators Guide* for more information on loading AleriML configurations, and the various options that can be specified in the `flags` parameter. If the AleriML configuration file was loaded successfully, the method returns zero. If it was unsuccessful, a non-zero error code will be returned. For more information about the attempt to load the AleriML configuration file into the server, inspect the log messages located on the server.

The `loadConfigString(const char * configString, const char * flags)` method attempts to load the AleriML configuration stored in the `configString` parameter, into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information used during the XML configuration string load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string.

If the AleriML configuration was loaded successfully, the method returns zero. If the loading was unsuccessful, a non-zero error is returned. In addition, when loading an AleriML configuration into the server, inspect the log messages located on the server for more information.

The `loadConfigStringApplyingConversion(const char * configString, const char * flags, const char * conversionConfigString)` method will attempt to load the AleriML configuration stored in the `configString` parameter into the running Sybase Aleri Streaming Platform instance. The `flags` parameter provides control information used during the AleriML configuration string load attempt. If additional control information isn't needed, the value of the `flags` parameter can be an empty string.

The `conversionConfigString` parameter points to an AleriML model used to apply specific conversion instructions during the AleriML configuration load.

If the AleriML configuration was loaded successfully, the method returns zero. If it was unsuccessful, a non-zero error code is returned. Also, you should inspect the log messages located on the server for more information when loading an AleriML configuration into the server.

The `SpPlatform` object provides two subscription-related methods you can use if you write your own low-level Gateway I/O code for the subscription instead of utilizing the Pub/Sub API. The methods are `addStreamToClient(int clientHandle, String streamName)` and `removeStreamFromClient(int clientHandle, String streamName)`. Both are part of the `SpPlatform` interface because the two are XMLRPC calls that manage the subscription characteristics of a Gateway I/O socket on which a subscription is currently running.

Once a subscription request is issued for an open Gateway I/O socket connection, the connection becomes a read-only connection. Asynchronous stream updates are delivered from the Sybase Aleri Streaming Platform to the client. Because of the "read-only" nature of the socket, additional Gateway I/O commands can no longer be issued on this socket connection, leaving the XMLRPC mechanism to fill in the gap.

The two methods are passed a `clientHandle`, which is an integer value returned by the Gateway I/O process when you send a low-level subscription request on the socket. The `addStreamToClient(...)` method lets you add an additional stream to the subscription list, and the `removeStreamFromClient(...)` method allows you to delete a stream from the subscription list.

If the Pub/Sub API subscription mechanism is used, `SpPlatform` is an effective method. The `createSubscription(const char *name, int flags, int deliveryType, *SpPlatformStatus)` method can create an `SpSubscription` object that is associated with the `SpPlatform` object. As the name implies, it is also a factory method used to create `SpSubscription` objects. A `SpSubscription` object has its own interface that is used to control the subscription.

As mentioned earlier, a subscription is used to get asynchronous stream updates from the Sybase Aleri Streaming Platform into your client application.

Similarly, there is a factory method called `createPublication(const char *name, SpPlatformStatus *status)` that creates an `SpPublication` object. An `SpPublication` object can publish stream input data and/or issue the Gateway I/O **commit()** command from the client application to the Sybase Aleri Streaming Platform. Refer to [Section 3.3.1.1, "Create an SpPublication Object"](#) for more information.

B.1.5. SpStream Object

The `SpStream` object is used to store the metadata associated with a stream residing on the Sybase Aleri Streaming Platform. The `SpStream` interface includes the following methods:

```
int getId();
std::string getName();
bool isBase();
SpStreamDefinition *getDefinition();
```

Details:

- The `getId()` method returns an integer that represents the stream's internal identifier on the Sybase Aleri Streaming Platform.
- The `getName()` method returns a string that represents the name of the stream.
- The `isBase()` method returns `true` if the stream is a source stream, `false` otherwise.
- The `getDefinition()` method returns a pointer to an object of type `SpStreamDefinition` which contains the schema information of the stream. Refer to [Section 3.3.1.2, "Create SpStream-DataRecord Objects"](#) for more information.

B.1.6. SpStreamDefinition Object

The `SpStreamDefinition` object stores the schema associated with a stream residing on the Sybase Aleri Streaming Platform. The `SpStreamDefinition` interface has the following methods and constants defined within it:

```
int getNumColumns();
std::vector<const char *> *getColumnNames();
std::vector<int> *getColumnTypes();
std::vector<int> *getKeyColumns();
std::vector<int> *getKeyColumnVector();
int bool isKeyColumn(int columnIndex);
```

Details:

- The `getNumColumns()` method returns the number of columns in the stream.
- The `getColumnNames()` method returns a vector of "*const char **", where each "*const char **" represents the name of the corresponding column.
- The `getColumnTypes()` method returns a vector of integers, each one a constant that represents the field type of the corresponding column. The `SpStreamDefinition` contains a list of integer "constants" representing the various column types.

This vector's size is equal to the value returned from the `getNumColumns()` method. In the distribution's `include/Data` directory, there is a file named `DataTypes.hpp`. This file contains an enumeration of the different `DataTypes` supported by the Pub/Sub API.

- The `getKeyColumns()` method returns a vector of integers, each of which is the column index (rel 0) of a key column in the streams field list. For example, if the stream has 10 columns, and the first three are key columns, the `getKeyColumns()` method returns a vector that includes the follow-

ing entries: [0, 1, 2].

- The `getKeyColumnVector()` method returns a vector of integers. Each field in the entire field list is represented by an integer, the value of which is either 1 (if the field is a key field), or zero (if the field is not a key field).
- The `isKeyColumn(int columnIndex)` returns a boolean value of `true` if the column index specified is that of a key field; otherwise, it returns `false`.

The `columnIndex` is “rel-0” as the first column of the field list has an index value of zero.

B.1.7. SpStreamProjection Object

The `SpStreamProjection` object stores the metadata associated with a stream projection based on an SQL query supplied to the `createSubscriptionProjection(...)` factory method of the `SpPlatform` object. The `SpStreamProjection` interface includes the following methods:

<code>getStream()</code>	returns a reference to the underlying <code>SpStream</code> onto which the SQL query was projected.
<code>getDefinition()</code>	returns a pointer to an object of type <code>SpStreamDefinition</code> , containing the schema information of the projection. This information is returned by the Sybase Aleri Streaming Platform when the SQL query associated with an <code>SpSubscriptionProjection</code> object is first created.

```
SpStream *getStream();
SpStreamDefinition *getDefinition();
```

B.1.8. Creating an SpSubscription or SpSubscriptionProjection Object

To make subscription requests to the Sybase Aleri Streaming Platform, you must create an `SpSubscription` or `SpSubscriptionProjection` object. Use the appropriate factory method provided by the `SpPlatform` object that has been previously instantiated. The `SpPlatform` factory methods that are used to create `SpSubscription` and `SpSubscriptionProjection` objects have the following signatures:

```
>
SpSubscription *createSubscription(const char * name,
int flags, int deliveryType, SpPlatformStatus *status);

SpSubscriptionProjection *createSubscriptionProjection(const char * name,
int flags, int deliveryType, const char * sqlQuery,
SpPlatformStatus *status);
```

Where:

<code>const char * name</code>	is an identifier that the client application program intends to assign to the <code>SpSubscription</code> or <code>SpSubscriptionProjection</code> object being created.
<code>int flags</code>	represents the “flag bits” that are to be sent to the Sybase Aleri Streaming Platform Gateway I/O process when the low-level sub-

scription request is made. The flag settings control delivery from the Sybase Aleri Streaming Platform to the client application, on the Gateway I/O socket connection where the subscription request was made. The “flag bits” are defined as constants in the `SpSubConst.hpp` interface file and are:

- `BASE = 0x0;`

The `BASE` flag bit tells the Sybase Aleri Streaming Platform that it should send a complete “snapshot” of each stream of the subscription request before sending subsequent updates to each stream. The complete snapshot or “state” of the stream is a group of “insert” records sent from the Sybase Aleri Streaming Platform between the `EVID_GATEWAY_SYNC_START` and `EVID_GATEWAY_SYNC_END` subscription events.

- `LOSSY = 0x1;`

The `LOSSY` flag bit puts the Sybase Aleri Streaming Platform in “data shedding mode”, where the oldest data is dropped when a client cannot keep up with the pace of the data coming out of the Sybase Aleri Streaming Platform. This ensures that when the client does read gateway data, it is always the most recent data that the Sybase Aleri Streaming Platform has delivered to the output gateway.

- `NOBASE = 0x2`

This flag tells the Sybase Aleri Streaming Platform that it should NOT send a complete “snapshot” of each stream of the subscription request. The Sybase Aleri Streaming Platform will send only subsequent updates for each stream.

- `DROPPABLE = 0x8`

This flag tells the Sybase Aleri Streaming Platform to drop its connection to the client application if that application (the one using this flag) can't keep up with the data being sent and its internal buffer is filled. This protects the Sybase Aleri Streaming Platform from getting into a situation where it has to stop processing incoming data because the its clients can't keep up with the data it is producing.

Because the connection is simply dropped, the client cannot expect a valid error code.

- `PRESERVE_BLOCKS = 0x20`

This flag tells the Sybase Aleri Streaming Platform that it should preserve blocks while sending data to the client application.

These flag bits can be ORed together using the “|” operator. For example, `flags = NOBASE | LOSSY`.

int deliveryType:

This integer value specifies how the client application program's `SpObserver` object receives the stream update events. Currently, there are several delivery type format specifiers defined in the `SpSubConst.hpp` interface file. The delivery type specifiers are as follows:

- `DeliveryType::DELIVER_PARSED = 1;`

This delivery type setting tells the `SpSubscription` object to deliver parsed field data objects representing the stream update to the `SpObserver` object.

- `DeliveryType::DELIVER_BINARY = 3;`

This delivery type setting tells the `SpSubscription` object to deliver the binary representation of the stream update record to the `SpObserver` object.

- `DeliveryType::DELIVER_STREAM_OPCODES = 5;`

This delivery type setting tells the `SpSubscription` object not to use field level parsing, but to simply deliver the stream update operation code (`INSERT`, `UPDATE`, `DELETE`, `UPSERT`, and so forth).

`const char *sqlQuery` specifies the SQL query projection on which the `SpSubscriptionProjection` will be based. The `sqlQuery` parameter is only used to create an `SpSubscriptionProjection` object. See [Section 5.1, “Aleri SQL Queries and Statements”](#) for SQL query handling limitations of the Sybase Aleri Streaming Platform.

`SpPlatformStatus status` is used to return error code information from the `createSubscription(...)` and `createSubscriptionProjection(...)` factory methods, if the subscription object could not be created.

The following example shows how to use the `SpPlatform` object called `sp` to create both an `SpSubscription` object and an `SpSubscriptionProjection` object:

```
SpSubscription *sub = sp->createSubscription("MySubscription_1",
    SpSubConst::BASE,
    SpSubConst::DELIVER_PARSED,
    status);

SpSubscriptionProjection *subProj = sp->createSubscriptionProjection(
    "MySubscriptionProjection_2",
    SpSubConst::BASE,
    SpSubConst::DELIVER_PARSED,
    "select intData, charData from inputstream where intData > 100",
    status);
```

In the above example, `status` is an `SpPlatformStatus` object that was created previously with the `SpFactory::createPlatformStatus()` factory method.

B.1.9. `SpSubscriptionCommon` Method Set

If the `sp->createSubscription(...)` or the `sp->createSubscriptionProjection(...)` call is successful, the client application program gets back either an `SpSubscription` or `SpSubscriptionProjection` object, which is then used in order to make the subscription. The set of methods that the `SpSubscription` and `SpSubscriptionProjection` types have in common have been abstracted into a super type called `SpSubscrip-`

tionCommon. The SpSubscriptionCommon interface is inherited by both the SpSubscription and SpSubscriptionProjection interfaces. The SpSubscriptionCommon interface defines the following method set:

```
>
std::string getName();
int getFlags();
int getDeliveryType();
int getClientHandle();

int removeObserver(int theCookie);

int start();

int stop();
```

The `getName()` method returns the name assigned to the subscription object when it was created with the SpPlatform's `createSubscription(...)`, or `createSubscriptionProjection(...)` method.

Similarly, the `getFlags()` and `getDeliveryType()` methods return the flag settings and the delivery type, respectively, that were specified in the `createSubscription(...)` or `createSubscriptionProjection(...)` method.

The `getClientHandle()` method returns an integer representing a “handle” that is assigned to the underlying subscription connection, by the Sybase Aleri Streaming Platform. A valid handle is one that is greater than zero. The value of the *clientHandle* is acquired from the Sybase Aleri Streaming Platform when the subscription is started through the `start()` method.

The `removeObserver(int theCookie)` method is used to remove the SpObserver from the subscription's delivery mechanism. There are differences between how you add observers to the two different types of subscriptions. These differences will be discussed within the SpSubscription and SpSubscriptionProjection interfaces.

The `start()` method is used to start the subscription process.

At least one stream and SpObserver must be registered with the subscription object before the subscription object can be started up through the `start()` method. See [Section C.2.2, “SpSubscriptionEvent”](#) for information on how to add respective SpObserver objects.

When you start up a subscription object, the following sequence of events takes place:

1. The subscription object establishes a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process, and authentication is performed.
2. A subscription request is sent to the Sybase Aleri Streaming Platform on this socket connection.
3. If the subscription request is accepted by the Sybase Aleri Streaming Platform, the subscription object reads the *clientHandle* that the Sybase Aleri Streaming Platform assigned to it.
4. A new thread is started up dedicated to reading stream update information off the read-only Gateway I/O socket connection.
5. Stream update messages flowing from the Sybase Aleri Streaming Platform to the client are read, parsed and delivered to the SpObserver objects.

When the SpObserver objects are “notified” for stream updates, through `notify(...)` methods, the SpObserver objects will actually be running within the context of this thread instead of the main one.

6. The `start()` method returns a zero back to the caller indicating that the subscription was started up successfully. If there is an error, the `start()` method returns a non-zero error code.

Note:

The `SpUtils.getErrorMessage(errorCode)` method can be used to get the specific error message.

The `stop()` method is used to shut down the subscription mechanism. The `stop()` method closes the socket connection and stops the thread that was used to read, parse and deliver the Sybase Aleri Streaming Platform updates to the `SpObserver` objects.

Here are additional methods in the interface:

```
void setPulseInterval(const uint32_t interval);
uint32_t getPulseInterval();
void setQueueSize(const uint32_t queue, SpPlatformStatus * status);
uint32_t getQueueSize();
void setBaseDrainTimeout(const uint32_t millis, SpPlatformStatus * status);
uint32_t getBaseDrainTimeout();
void setExitOnClose(SpPlatformStatus * status);
bool getExitOnClose();
```

- `setPulseInterval` can be used to set the pulse interval in seconds if the subscription was created with the pulsed flag on.
- `getPulseInterval` is used to retrieve the current setting of the pulse interval in seconds.
- `setQueueSize` is used to set the internal buffer size in the Sybase Aleri Streaming Platform for this subscription. The Sybase Aleri Streaming Platform uses this buffer to queue up messages if the subscriber is slow in retrieving. The buffer prevents it from blocking and slowing down. The setting is made when the subscription is started. It is necessary to keep the *status* parameter valid until the time the start call is made.
- `getQueueSize` retrieves the current value of the queue size.
- `setBaseDrainTimeout` is used to set the time in milliseconds that the Sybase Aleri Streaming Platform should wait before dropping a blocked subscription. If a subscription is started with the *DROPPABLE* flag set, the Sybase Aleri Streaming Platform closes a subscription connection if the messages block due to a slow client. This parameter specifies how long to wait before closing the connection. The setting is made when the subscription is started. It is necessary to keep the *status* parameter valid until the time the start call is made.
- `getBaseDrainTimeout` retrieves the current value in milliseconds of the base drain timeout.
- If the `setExitOnClose` is set, the Sybase Aleri Streaming Platform will shut down once this subscription connection is closed by the client. The setting is made when the subscription is started. You must keep the *status* parameter valid until the time the start call is made.
- `getExitOnClose` retrieves the current setting of the exit on close flag.

B.1.10. SpSubscription Method Set

If the `sp->createSubscription(...)` call is successful, the client application program gets an `SpSubscription` object back that is ultimately used to subscribe. An `SpSubscription` object can be used to subscribe to one or more streams, while an `SpSubscriptionProjection` object

can only subscribe to the projection defined by the *sqlQuery* passed in to the *createSubscriptionProjection(...)* factory method. For each stream being observed, the *SpSubscription* object delivers to the *SpObserver*, stream events that contain all of the stream's fields. The *SpSubscription* object extends the method set defined in the *SpSubscriptionCommon* interface as follows:

```
int /*Cookie*/ addStreamObserver(const char * streamName,
    SpObserver *theObserver);

int /*Cookie*/ addStreamsObserver(
    std::vector<std::string> * theStreamNames, SpObserver *theObserver);

int subscribe(const char *streamName);

int unsubscribe(const char *streamName);
```

The next few methods are used to set up the streams and their corresponding *SpObserver* objects for the *SpSubscription*. The client application programs must create their own *SpObserver* objects, which are notified by the *SpSubscription* with the updates that arrive from the Sybase Aleri Streaming Platform for the registered streams. The client application programs create *SpObserver* objects by implementing the *SpObserver* interface.

The *addStreamObserver(const char * streamName, SpObserver *theObserver)* method tells the *SpSubscription* object to send all updates for the *streamName* to the *SpObserver* object specified by *theObserver* parameter. You can call this method as many times as required.

The *addStreamsObserver(std::vector<std::string> *theStreamName, SpObserver *theObserver)* method can be used to associate several streams with a particular *SpObserver* at once. The same thing can be accomplished by making multiple calls to the *addStreamObserver(...)* method previously shown. Again, multiple *addStreamsObserver(...)* calls can be made to set up the streams and their corresponding observers.

The *addStreamObserver(...)* and *addStreamsObserver(...)* calls return an integer value that represents a “cookie”/“handle” to the registered *SpObserver* object. Later on, you can use the cookie to remove the *SpObserver*.

The *removeObserver(int theCookie)* method, with a signature defined in the *SpSubscriptionCommon* interface, removes the *SpObserver* from the *SpSubscription*'s delivery mechanism.

Once the *SpSubscription*'s *start()* method has been called, the *SpSubscription* object provides two methods to modify the stream set currently being managed by the “running” subscription, *subscribe(const char *streamName)* and *unsubscribe(const char *streamName)*.

These two methods take a single *streamName* parameter. In the case of the *subscribe(const char * streamName)* call, the client application program must first ensure that there is an *SpObserver* associated with the subscribed stream. You can do this by first calling the *addStreamObserver(streamName, theObserver)* method to register the observer for the stream, and then calling the *subscribe(streamName)* method.

If successful, both the *subscribe(const char * streamName)* and *unsubscribe(const char * streamName)* methods returns a zero. Otherwise, a non-zero error code is sent back to the caller, where the *SpUtils::getErrorMessage(rc)* method can be used to see the error text associated with the error code.

B.1.11. SpSubscriptionProjection Method Set

If the `sp->createSubscriptionProjection(...)` call is successful, you get back an `SpSubscriptionProjection` object ultimately used to instantiate the subscription. The contents of the data returned from the Sybase Aleri Streaming Platform back to the `SpSubscriptionProjection` object are determined by the SQL query passed into the `createSubscriptionProjection(...)` factory method. An `SpSubscriptionProjection` can only receive updates for the underlying stream specified in the SQL query while the `SpSubscription` can get updates for more than one stream. The `SpSubscriptionProjection` interface extends the method set defined in the `SpSubscriptionCommon` interface as follows:

```
SpStreamProjection *getStreamProjection();  
int /*Cookie*/ addObserver(SpObserver *theObserver);
```

The `getStreamProjection()` method returns the `SpStreamProjection` object produced when the SQL query was sent to the Sybase Aleri Streaming Platform for parsing during the execution of the `createSubscriptionProjection(...)` factory method. If the SQL query could not be parsed, the `createSubscriptionProjection(...)` factory method returns null; the corresponding error information is added to the `SpPlatformStatus` object that was passed to the `createSubscriptionProjection(...)` factory method. If the `createSubscriptionProjection(...)` method succeeds, the program gets back an `SpSubscriptionProjection` object, which can then make a call to `getStreamProjection()` for the schema information produced by the SQL query parse. The `SpStreamProjection` that is returned should be treated as “read-only”, and not modified by the client application program. Typically, the `SpStreamProjection` objects are passed into the `SpObserver`'s constructor, giving the `SpObserver` the list of fields and corresponding data types. This information is typically used by the `SpObserver` to process the updates that come back from the server.

B.1.12. SpSubscriptionEvent

An `SpSubscriptionEvent` provides the following method set:

```
std::string getSubName();  
int getType();  
std::string getTypeName();  
int getId();  
std::string getIdName();  
int getStreamId();  
int getStreamOpCode();  
void *getData();
```

The `getSubName()` method returns a string that represents the name of the `SpSubscription/SpSubscriptionProjection` object that generated and delivered this event to the `SpObserver`. This “name” was assigned to the `SpSubscription/SpSubscriptionProjection` object when it was first created through the `SpPlatform createSubscription(...)` method.

The `getType()` method returns an integer representing the “type” of this `SpSubscriptionEvent`. Currently, there are four types (or categories) of events defined in the `SpSubscriptionEvent` class:

- `SpSubscriptionEvent::EVTYPE_PARSED_DATA`

Events delivered from a subscription object that was created with a delivery type of `SpSub-`

Const::DELIVER_PARSED.

- *SpSubscriptionEvent::EVTYPE_BINARY_DATA*

Events delivered from a subscription object that was created with a delivery type of *SpSubConst::DELIVER_BINARY.*

- *SpSubscriptionEvent::EVTYPE_STREAM_OPCODE_DATA*

Events delivered from a subscription object that was created with a delivery type of *SpSubConst::DELIVER_STREAM_OPCODES.*

- *SpSubscriptionEvent::EVTYPE_SYSTEM*

Events delivered by the subscription object to indicate that a system event such as an error, halt in communication, or shutdown of the Sybase Aleri Streaming Platform has taken place.

The `getTypeName()` method returns the string literal value, representing the type of event instead of the internal integer representation. You can use this for output messages.

The `getId()` method returns an integer representing a unique event ID that can be safely used within a switch statement to “case” on. The event IDs are unique across the entire set of event types. For example, an `SpSubscriptionEvent` may have a type of `SpSubscriptionEvent::EVTYPE_SYSTEM`, which means it is a system-related notification. The `getId()` method returns what was actually detected by the system, for example, `SpSubscriptionEvent::EVID_PARSING_ERROR`, `SpSubscriptionEvent::EVID_COMMUNICATOR_HALTED`, and so forth. As is the case with the event types, all of the event IDs are enumerated within the `SpSubscriptionEvent.hpp` interface.

These are the `SpSubscriptionEvent` identifiers:

- *SpSubscriptionEvent::EVID_GATEWAY_SYNC_START*

It is delivered to the `SpObserver` if the `SpSubscription` is sent a `START_SYNC` Gateway I/O message from the Sybase Aleri Streaming Platform. The `START_SYNC` message contains the ID for the stream with which the message is associated.

If this event is delivered to the `SpObserver`, it indicates the start of the stream's “snapshot”. Subsequent events should be `INSERT` messages for each record in the stream until the `END_SYNC` Gateway I/O message is received from the Sybase Aleri Streaming Platform. A call to the `START_SYNC` event's `getData()` method returns null. For this message to be sent from the Sybase Aleri Streaming Platform to the client application, the `SpSubscription` had to be created with the `SpSubConst::BASE` flag specified. If the `SpSubConst::NOBASE` flag is specified instead, the `START_SYNC` message would never have been delivered from the Sybase Aleri Streaming Platform to the client application.

This event is also delivered after the dynamic Sybase Aleri Streaming Platform changes if the stream's contents gets regenerated, after the `WIPEOUT` event. In this situation the `START_SYNC` event is delivered even if the subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- *SpSubscriptionEvent::EVID_GATEWAY_SYNC_END*

This event is delivered to the `SpObserver` if the `SpSubscription` is sent an `END_SYNC` Gateway I/O message from the Sybase Aleri Streaming Platform. The `END_SYNC` message contains the ID for the stream with which the message is associated.

If this event is delivered to the `SpObserver`, it indicates that the end of the stream's "snapshot" has been reached. A call to the `END_SYNC` event's `getData()` method returns null. For this message to be sent from the Sybase Aleri Streaming Platform to the client application, the subscription has to be created with the `SpSubConst::BASE` flag specified. If the `SpSubConst::NOBASE` flag is specified instead, the `END_SYNC` message would never be delivered from the Sybase Aleri Streaming Platform to the client application.

This event is also delivered after the dynamic Sybase Aleri Streaming Platform changes if the stream's contents are regenerated. The `WIPEOUT` event is followed on the dynamic regeneration by the `START_SYNC` event insertion of the new data and the `END_SYNC` event. All the events are delivered even if the subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- `SpSubscriptionEvent::EVID_GATEWAY_WIPEOUT`

It is delivered to the `SpObserver` after the dynamic Sybase Aleri Streaming Platform changes if the stream's contents are regenerated. The event means the current contents of the stream are being discarded. The `WIPEOUT` event is followed by the `START_SYNC` event, insertion of the new data, and the `END_SYNC` event. All the events are delivered even if the subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- `SpSubscriptionEvent::EVID_BINARY_DATA`

If the subscription is created with the delivery type of `SpSubConst::DELIVER_BINARY`, the `getData()` method returns a pointer to an `SpBinaryData` object containing the binary message delivered from the Sybase Aleri Streaming Platform. See `SpBinaryData.hpp` for the definition of the interface.

- `SpSubscriptionEvent::EVID_PARSED_FIELD_DATA`

When a subscription object is created with a delivery type of `SpSubConst::DELIVER_PARSED`, it attempts to parse the field data of the stream messages transmitted by the Sybase Aleri Streaming Platform and delivers this parsed field information to the `SpObserver`. The `getStreamOpCode()` method can be used to determine whether the message was an `INSERT`, `UPDATE`, or `DELETE`. The parsed field data is accessed by the `SpObserver` through the event's `getData()` method.

- `SpSubscriptionEvent::EVID_PARSED_PARTIAL_FIELD_DATA`

It is similar to the `SpSubscriptionEvent::EVID_PARSED_FIELD_DATA` event. This event is delivered when the `SpSubscription`'s message parser detects an error as it parses out the field data for the message sent from the Sybase Aleri Streaming Platform. If an error is encountered during the parse, only those fields that were successfully parsed up to the place where the error was detected will be delivered. The `SpObserver` is not obligated to inspect the partial results, but you may want to use the partial results for debugging purposes.

- `SpSubscriptionEvent::EVID_COMMUNICATOR_HALTED`

It is delivered when the low-level socket on which the subscription runs is closed. In this case, a call to the `getData()` method returns null.

- `SpSubscriptionEvent::EVID_PLATFORM_SHUTDOWN`

It is delivered when you attempt to issue a "shutdown" through the `SpPlatform` object. In this case, a call to the `getData()` method returns null.

- `SpSubscriptionEvent::EVID_PARSING_ERROR`

It is delivered when a parsing error is detected by the `SpSubscription`, and some context to report. A

call to the `getData()` method returns an object of type `SpParserReturnInfo`.

For more information, see [Section 3.2.3, “Receive/Process Subscription Updates Using C++”](#).

- `SpSubscriptionEvent::EVID_UNKNOWN_PARSING_ERROR`

It indicates that the parser encountered an unexpected error before the completion of the process. In this case, the `getData()` method returns an integer object that contains the record length for the message that is about to be parsed.

- `SpSubscriptionEvent::EVID_READ_STREAM_RECORD_ERROR`

It indicates that the parser could not successfully read the record that was delivered from the Sybase Aleri Streaming Platform. The `getData()` method returns an integer object containing the value of the record length that was read for the bad record.

- `SpSubscriptionEvent::EVID_BAD_RECORD_LENGTH_ERROR`

It indicates that the record length read of the socket was bad. The `getData()` method returns an integer object that contains the bad record length value read off of the socket.

- `SpSubscriptionEvent::EVID_BAD_GATEWAY_OP_CODE_ERROR`

It indicates that the Gateway I/O operation code for the message sent from the Sybase Aleri Streaming Platform is invalid. The `getData()` method returns an integer object that contains the bad Gateway I/O operation code that was read.

- `SpSubscriptionEvent::EVID_HOT_SPARE_SWITCH_OVER_INITIATED`

It is delivered to the `SpObserver` when the Pub/Sub API recognizes that a connection attempt should be made to the High Availability (Hot Spare) server. The High Availability (Hot Spare) connection parameters were specified in the `SpPlatformParms` object passed to the `SpFactory::createPlatform()` method when the underlying `SpPlatform` was first created.

- `SpSubscriptionEvent::EVID_HOT_SPARE_SWITCH_OVER_SUCCEEDED`

It is delivered to the `SpObserver` when the connection to the High Availability (Hot Spare) server is made successfully.

- `SpSubscriptionEvent::EVID_HOT_SPARE_SWITCH_OVER_FAILED`

It is delivered to the `SpObserver` when the connection to the High Availability (Hot Spare) server fails.

The `getIdName()` method returns the string literal value that corresponds to the numeric event ID. You can use it for output messages.

The `getStreamId()` method returns the stream id that is associated with this event. For example, an `SpObserver` may receive an event of type `SpSubscriptionEvent::EVTYPE_PARSED_DATA`, where the event id is `SpSubscriptionEvent::EVID_PARSED_FIELD_DATA`, indicating that the event contains parsed field data. The `getStreamId()` method returns the stream id to which this event data corresponds.

The `getStreamOpCode()` method returns the stream operation code that is associated with this event. For example, an `SpObserver` may receive an event of type `SpSubscriptionEvent::EVTYPE_PARSED_DATA`, where the event id is `SpSubscriptionEvent::EVID_PARSED_FIELD_DATA`, indicating that the event contains parsed field data.

The `getStreamOpCode()` method returns a value that indicates whether the event is an *INSERT*, *UPDATE*, *DELETE*, and so forth.

The `getData()` method returns a void pointer to the data associated with the event. Depending on the event id, the pointer must be typecast into a pointer to an object of the correct “type” before the data can be inspected.

For example, if a subscription object is set up to deliver parsed data, and the `SpObserver` receives an `SpSubscriptionEvent` whose event id is `SpSubscriptionEvent::EVID_PARSED_FIELD_DATA`, the `getData()` method returns a void pointer that must be typecast into the following:

```
std::vector<SpDataValue *> *fieldData =  
    (std::vector<SpDataValue *> *) ev->getData();
```

Where `fieldData` is a pointer to a vector of pointers to `SpDataValue` objects. Refer to [Section B.1.14, “SpDataValue Object”](#) for more information. Each element of the vector represents the data associated with the corresponding field (in the field order specified in the corresponding stream's `SpStreamDefinition`).

In the case where the event id is `SpSubscriptionEvent::EVID_BINARY_DATA`, the `getData()` method returns a void pointer that must be typecast into a pointer to an `SpBinaryData` object as follows:

```
SpBinaryData *binData = (SpBinaryData *) ev->getData();
```

If the subscription was created using a delivery type of `SpSubConst::DELIVER_STREAM_OPCODES`, the `getData()` method returns null. Use the `ev->getStreamOpCode()` method to determine the stream operation code (*INSERT*, *UPDATE*, *DELETE*, *UPSERT*, and so forth).

If the `getData()` method call returns a pointer to a vector of `SpDataValue` objects, after processing this subscription, the following application should be done to close potential memory leaks.

1. Iterate through the vector and delete each object
2. Delete the vector itself

B.1.13. SpParserReturnInfo object

The `SpParserReturnInfo` object referenced above has the following method set to use within the `SpObserver`:

```
int getErrorCode();  
std::string getErrorMessage();  
int getTransMessageIndex(); // REL 0  
int getColumnIndex(); // REL 0  
std::string getErrorData();  
bool isSuccess();
```

Details:

- The `getErrorCode()` and `getErrorMessage()` methods, respectively return the parser error code and associated error message.
- The `getTransMessageIndex()` method is only relevant for transaction messages that are being parsed at the time of error. If the message is not a transaction, `-1` is returned. If the message is a transaction block, the transaction block's message index is `rel 0` (the first message has an index value of `0`).
- The `getColumnIndex()` method returns the column index (`rel 0`) for the column where the parsing error is detected. Again, if the error occurred before the parser event got to the first column (having a column index value of zero), this method returns `-1`.
- The `getErrorData()` method returns a string that the parser may have put together at the point of error to indicate what went wrong. Typically, if there is any extra error information that the parser determined was important, it is stored in a string, which is returned by the `getErrorData()` method.
- The `isSuccess()` method returns `true` if the parse is successful and returns `false` if any error occurs.

B.1.14. SpDataValue Object

The `SpDataValue` structure encapsulates the `Data/DataTypes.hpp` file's `dataValue` structure. It simply carries the data type of the field data and provides a destructor that is used to free the `dataValue.val.stringv` component of the structure.

If the string is not to be freed, ensure that the `stringv` value of the structure is set to zero (NULL) before calling the destructor. This is required in the case where a literal character string is assigned, such as “my test data string”, to the `dataValue.val.stringv`.

The `SpDataValue` structure is defined as follows:

```
struct SpDataValue
{
    struct DataTypes::DataValue dataValue;
    DataTypes::DataType type;
    ~SpDataValue();
}
```

Where:

The `DataTypes::DataType` enumerates the Sybase Aleri Streaming Platform data types. It is defined in the `Data/DataTypes.hpp` file as follows:

```
enum DataType
{
    INT32=1,
    INT64=2,
    DOUBLE=3,
    DATE=4,
    STRING=5,
    NULLVALUE=6,
    MONEY=7,
    TIMESTAMP=8,
};
```

The struct `DataTypes::DataValue` is also defined in the `Data/DataTypes.hpp` file as follows:

```
struct DataValue
{
    union
    {
        int32_t int32v;
        int64_t int64v;
        money_t moneyv;
        double doublev;
        time_t datetv;
        timestamp_t timestampv;
        const char * stringv;
    } val;
    bool null;
};
```

For example, using an `SpDataValue` object pointer called `ptrData`, an `int32` field value of 100 can be set up, as follows:

```
ptrData->type = DataType::INT32;
ptrData->dataValue.val.int32v = 100;
ptrData->dataValue.null = false;
```

The same mechanism can be used to set up the other field data elements. For setting up an `INT64`, set `ptrData->dataValue.val.int64v = 100`, set the `ptrData->type = DataType::INT64`; and so forth.

Note:

Use the `dataValue.null = true` statement to set a NULL field data value.

Look at the source code for `pubexample.cpp` and `subexample.cpp` (`SubExampleSpObserver.cpp`) to see the `SpDataValue` structure in use. The example shows how to handle the different data types supported by the Sybase Aleri Streaming Platform.

B.1.15. SpBinaryData Object

The `SpBinaryData` class provides the following methods:

```
int getLength();
int getServerDateSize();
char *getDataBuffer();
bool getDoByteSwapFlag();
```

Details:

- The `getLength()` method returns the length of the character buffer that is used for storing the binary data.
- The `getServerDateSize()` returns an integer indicating the number of bytes that are used to

represent the Sybase Aleri Streaming Platform date field having the enumerated type of `DataType::DATE`.

`DataType::TIMESTAMP` values are always a fixed length, whether the Sybase Aleri Streaming Platform was compiled as a 32-bit or a 64-bit application.

- The `getDataBuffer()` method returns a pointer to the buffer that stores the actual binary data read off the low-level Gateway I/O socket connection. The length of this buffer (in addition to being stored internally within the character buffer itself) is retrieved using the `getLength()` method previously referenced. This buffer may contain null characters, but must not be interpreted as a null-terminated character string.
- The `getDoByteSwapFlag()` returns either `true` or `false`, indicating whether the client application needs to do byte swapping in order to interpret the various field data, stored within the buffer. Byte swapping is required if the client and the Sybase Aleri Streaming Platform use different endianness: that is, if the client is little-endian while the Streaming Processor is big-endian, or vice versa.

B.2. C++ Objects for Publication

B.2.1. SpPublication Method Set

If the `sp->createPublication(...)` call is successful, the client application program will get an `SpPublication` object back and can proceed to use it. An `SpPublication` object can be used to publish data to one or more streams. It implements an interface that provides the following method set:

```
std::string getName();

int start();
int publish(SpStreamDataRecord *streamRecord);

int publish(
    std::vector<SpStreamDataRecords *> *streamRecords,
    int streamOpCodeOverride,
    int streamFlagOverride);

int publishTransaction(
    std::vector<SpStreamDataRecords *> *streamRecords,
    int streamOpCodeOverride,
    int streamFlagsOverride,
    int maxRecordsPerBlock);

int publishEnvelope(
    std::vector<SpStreamDataRecords *> *streamRecords,
    int streamOpCodeOverride,
    int streamFlagsOverride,
    int maxRecordsPerBlock);

int commit();

int stop();

int publish(
    std::vector<SpStreamDataRecords *>
    *streamRecords,
    int streamOpCodeOverride,
    int streamFlagOverride,
    SpPlatformStatus * status);

int publishTransaction(
```

```
std::vector<SpStreamDataRecords *>
*streamRecords,
int streamOpCodeOverride,
int streamFlagsOverride,
int maxRecordsPerBlock,
SpPlatformStatus * status);

int publishEnvelope(
std::vector<SpStreamDataRecords *>
*streamRecords,
int streamOpCodeOverride,
int streamFlagsOverride,
int maxRecordsPerBlock,
SpPlatformStatus * status);
```

The `getName()` method returns the “name” assigned to this `SpPublication` object when it was created through the `SpPlatform`’s `createPublication(...)` factory method.

The `start()` method is used to start the publication process. When an `SpPublication` object is started, the following events takes place:

1. The `SpPublication` object creates a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process.
2. The `SpPublication` authenticates with the Sybase Aleri Streaming Platform.
3. The `start()` method returns a zero back to the caller indicating that the `SpPublication` object was successfully started. If there is an error, the `start()` method will return a non-zero error return code.

The `SpUtils::getErrorMessage(errorCode)` method can be used to get the specific error message.

Unlike the `SpSubscription` mechanism, this mechanism does not create a separate thread to manage the publication. Behind the scenes, a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process is used to transmit stream data to the Sybase Aleri Streaming Platform, and to read the its response associated with each individual request. Unless otherwise specified in the flag values used when publishing data, a publication request is synchronous. The client application program calls one of the `Publish` methods, and waits for the Sybase Aleri Streaming Platform to respond with an “ack” or “nak”. However, there is a special stream flag, enumerated as `StreamInterface::NOACK` in `/include/Stream/StreamInterface.hpp`, that can be used to make an asynchronous publication request. When this flag is specified, the `Publish` method sends the request out to the Gateway I/O process and returns control immediately back to the caller, without waiting for a response from the Sybase Aleri Streaming Platform.

The best way for the client application to ensure data integrity is to wait for the `ack`. If the `ack` is not received, the application can retry the publication method call.

If a `nak` is generated by a Sybase Aleri Streaming Platform configured for High Availability, it triggers the failover process.

The `publish(SpStreamDataRecord streamRecord)` method is used to publish/send a single stream input record to the source stream on the Sybase Aleri Streaming Platform. Although it is more efficient to send input stream records to the Sybase Aleri Streaming Platform in batches known as “transactions”, this method can be used initially when you want to test a new data model, perhaps by sending one stream input record at a time. If successful, a return code of zero is sent back to the caller. Otherwise, an

error code is sent back. The `SpUtils::getErrorMessage(errorCode)` method can be called to get the specific error message.

Each of the `Publish` methods of the `SpPublication` object takes one or more `SpStreamDataRecord` objects as shown in the coding example above. The `SpStreamDataRecord` object represents one row of stream data that is to be sent to the Sybase Aleri Streaming Platform. Each `SpStreamDataRecord` object has its own “op code”, which indicates how the row is to be handled by the Sybase Aleri Streaming Platform when it is received. For example, the op code may indicate that the row is to be treated as an *INSERT*, a *DELETE*, an *UPDATE*, and so forth.

Refer to [Section B.2.4, “SpStreamDataRecord Object”](#) for more information.

The `publish(std::vector<SpStreamDataRecord*> *streamRecords, int streamOpCodeOverride, int streamFlagOverride)` sends a vector of `SpStreamDataRecord` objects to the Sybase Aleri Streaming Platform with one call. The `streamOpCodeOverride` and `streamFlagOverride` parameters can be used to override the corresponding values found in the individual `SpStreamDataRecord` objects that comprise the collection.

Although `publish` does work with a vector of `SpStreamDataRecord` objects, it actually iterates over the vector and calls `publish` for each one. However, this method lets you create a set of stream data records where each record can be applied to a different stream. It may be used for debugging or testing, where the ordered sequence of updates to multiple Source Streams is important.

The `publishTransaction(std::vector<SpStreamDataRecord *> *streamRecords, int streamOpCodeOverride, int streamFlagsOverride, int maxRecordsPerBlock)` and the `publishEnvelope(std::vector<SpStreamDataRecord *> *streamRecords, int streamOpCodeOverride, int streamFlagsOverride, int maxRecordsPerBlock)` methods are the most efficient in terms of bundling multiple `SpStreamDataRecords` for a single stream, stored in the collection and sending at once to the Sybase Aleri Streaming Platform as a single batch.

All `SpStreamDataRecord(s)` that are to be sent as a transaction/envelope must be for the same source stream. Each `SpStreamDataRecord` can have a different op code, such as *INSERT*, *UPDATE*, *DELETE*, but the records must be for the same source stream.

As previously mentioned, the override *op code* and *flag* values can be used to override the corresponding values found within the individual `SpStreamDataRecord` objects that make up the collection. In addition, this method takes one more parameter called *maxRecordsPerBlock* which is an integer value that specifies the maximum number of `SpStreamDataRecords` to send as a transactional/envelope unit to the Sybase Aleri Streaming Platform. If the value is set to zero, the method will try to send all of the `SpStreamDataRecord` objects in the vector to the Sybase Aleri Streaming Platform within one transaction block. If the *maxRecordsPerBlock* is less than the actual number of records in the collection, then the record set will be broken up (using the *maxRecordsPerBlock* setting) into multiple transactions/envelopes during transmission to the server.

The difference between a transaction and envelope block transmitted to the server is how the server processes the block of records upon receiving it. As the name implies, a group of records within a transaction block is treated as a single transactional unit on the server side. In the case of an envelope, the group of records contained within the envelope is processed a single record at a time by the server. So the envelope mechanism allows the client to send a batch of records for a specific stream to the server in one shot as opposed to sending a single record at a time and waiting for each record's ack/nak reply from the server.

Each of the three `publish` functions described above has a version that accepts a pointer to `SpPlatformStatus`. These versions are functionally similar, but in error conditions, return extended information if available in the *SpPlatformStatus* parameter.

The `commit()` method issues a special Gateway I/O command to the Sybase Aleri Streaming Platform, requesting that all pending input records previously sent to the Sybase Aleri Streaming Platform

be synced to disk. Making a commit call is a tremendously expensive operation relative to latency. It's designed to be used only as part of a two-phase commit process when reading from a persistent source, such as an ActiveMQ series, and writing to a Sybase Aleri Streaming Platform instance that uses Sybase's log store persistence model.

In a real time, low latency streaming scenario, the commit call should not be used after each record.

Typically the commit call should be used as follows: For a standard two-phase commit process that guarantees against data loss, the client reads messages from the source, such as ActiveMQ, and publishes to the Sybase Aleri Streaming Platform until reaching a pre-determined number (>1024 is recommended) of processed messages or a specified amount of time has elapsed. After reaching the value set for the maximum number messages or the elapsed time has passed, the `commit()` call is made and upon return to the client, the client may inform the source, such as ActiveMQ, that the messages can be deleted.

The `stop()` method shuts down the underlying Gateway I/O socket connection.

B.2.2. Stream Operation Codes

The following is a set of stream operation codes that can be set for each individual `SpStreamDataRecord` or one of the `streamOpCodeOverride` parameters can be found in the `./include/Stream/StreamInterface.hpp` file:

- `StreamInterface::NOOP`

When specified as the `streamOpCodeOverride` parameter, it indicates that the stream op code stored in each of the individual `SpStreamDataRecord` objects should be used by the Sybase Aleri Streaming Platform. If the stream op code within the `SpStreamDataRecord` is set to `StreamInterface::NOOP`, then the Sybase Aleri Streaming Platform will default the stream operation to an `INSERT` operation.

- `StreamInterface::INSERT`

When specified, it tell the Sybase Aleri Streaming Platform to treat the published stream record as an `INSERT` operation.

- `StreamInterface::UPDATE`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as an `UPDATE` operation.

- `StreamInterface::DELETE`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as a `DELETE` operation.

- `StreamInterface::UPSERT`

When specified, it tells the Sybase Aleri Streaming Platform to treat the published stream record as an `UPSERT` operation. An `UPSERT` operation either inserts the stream record into the source stream if it is not already present or it updates the existing source stream record using the contents of the stream record.

B.2.3. Stream Flag Values

The following set of stream flag values that can be set for each individual `SpStreamDataRecord` or as one of the `streamFlagOverride` parameter can be found in the `StreamInterface.hpp` file:

- *StreamInterface::NULLFLAG*

When specified as the “streamFlagOverride” parameter, it indicates that the stream flag stored in each of the individual *SpStreamDataRecord* objects should be used by the Sybase Aleri Streaming Platform. If the stream flag within the *SpStreamDataRecord* is set to *StreamInterface::NULLFLAG*, then it means that nothing significant needs to take place. The normal synchronous publication sequence where the *SpPublication* waits for a Sybase Aleri Streaming Platform response can go on.

- *StreamInterface::NOACK*

It tells the Sybase Aleri Streaming Platform not to send an ack or nak back to the client application that issued the publication request. The *publish* method that is called runs asynchronously, and it assumes that the record is received and processed by the Sybase Aleri Streaming Platform.

- *StreamInterface::SHINE*

This flag is only relevant for the stream op code values of *UPDATE* and *UPSERT*. Typically, all of the fields for a stream record being published to the Sybase Aleri Streaming Platform must be assigned values for each of the available stream operations (*INSERT*, *UPDATE*, *UPSERT*, and so forth). In the case of *UPDATE* and *UPSERT*, you can use the *SHINE* flag to update just a few values of the “other” fields within the stream record without having to specify the values of the other fields. The client application program can set the other fields of the stream record being published to the Sybase Aleri Streaming Platform to null, and if the *SHINE* flag is set, the Sybase Aleri Streaming Platform will ignore the nulls and leave the existing field values in the record being updated. The key fields must always be present, as they are required to locate the record.

In essence, the Sybase Aleri Streaming Platform lets the existing field values “shine through” for each of the null values you sent.

The flag values represent bits that can be ORed together. For example:

```
int flags = StreamInterface::SHINE | SpGatewayConstants.SF_NOACK;
```

B.2.4. *SpStreamDataRecord* Object

Each “publishing” method of the *SpPublication* object sends stream input data from the client application to the Sybase Aleri Streaming Platform. Every stream record or row of stream data is encapsulated within a *SpStreamDataRecord* object, which has the following method set:

```
SpStream *getStream();  
std::vector<SpDataValue *> *getFieldData();  
int getOpCode();  
int setOpCode(int value);  
int getFlags();  
int setFlags(int value);
```

Details:

- The *getStream()* method returns a pointer to an *SpStream* object with which this *SpStreamDataRecord* is associated. Refer to [Section B.1.5, “SpStream Object”](#) for more information.

- The `getFieldData()` method returns a pointer to a vector of pointers to `SpDataValue` objects, representing the data for each field in the stream record. Refer to the `SpDataValue.cpp` file for more details. In addition, see the `pubexample.cpp` and `SubExampleSpObserver.cpp` files for examples on how to use `SpDataValue` objects.
- The `getOpCode()` method returns the stream op code currently set for this record.
- The `setOpCode(int value)` method sets the value of the stream op code for this record.
- The `getFlags()` method returns the flag settings currently set for this record.
- The `setFlags(int value)` method sets the value of the stream flag settings for this record.

B.3. C++ Objects for Record and Playback

B.3.1. SpRecorder object

Client programs need to call the factory method `createRecorder` defined in `SpPlatform` in order to create an `SPRecorder`.

```
>
public virtual SpRecorder * createRecorder(std::string name, std::string filename, std::vector<std::string> streams,
                                          int flags, uint64_t maxRecords, SpPlatformStatus * status)
```

The method takes the following parameters:

- `name` is a string that will uniquely identify this instance of the recorder object
- `filename` is the name of the file where recorded data will be stored
- `streams` is a vector of strings containing names of the streams for which to record events
- `flags` control the subscription. These flags are passed to the underlying subscription on the Sybase Aleri Streaming Platform. Can be a bitwise OR of the following values:
 - One of `SpSubscriptionCommon.BASE` or `SpSubscriptionCommon.NOBASE` - whether to record data already in streams at the time of connection
 - `SpSubscriptionCommon.LOSSY` - whether the Sybase Aleri Streaming Platform should discard records if the client application cannot keep up
- `maxRecords` - maximum number of records to process
- `status` - an `SpPlatformStatus` object to return information in case of error

`SpRecorder` has the following public interface

```
public std::string getName();
public int32_t start();
public int64_t getRecordCount();
public int32_t stop();
```

Details:

`getName()` returns the identifier assigned to this instance of the `SpRecorder` object
`start()` spawns a background thread which starts the recording process. The method returns once the thread is started. Returns 0 on success.
`getRecordCount()` returns the number of data records processed.
`stop()` stops the recording process by terminating the recording thread and closing connections to the Sybase Aleri Streaming Platform. Returns 0 on success.

B.3.2. SpPlayback object

An `SpPlayback` object is created by calling the following factory method defined in `SpPlatform`.

```
>
public virtual SpPlayback * createPlayback(std::string name, std::string filename, double scale, int64_t
                                         SpPlatformStatus * status);
```

The method takes the following parameters:

- `name` is a string that will uniquely identify this instance of `SpPlayback`
- `filename` is the name of the file containing the recorded data
- `scale` is a factor controls the rate of playback. Values -1 to 1 have no effect and the data is played back at the rate it was recorded at. Values greater than 1 speed up playback by that factor, for example, a value of 2 will play back twice as fast. Values less than -1 slow down playback by the factor specified.
- `maxrecords` is the maximum number of records to playback
- `status` is an `SpPlatformStatus` object used to return information in case of error

`SpPlayback` has the following public interface

```
public std::string      getName();
public void             setSendUpsert(bool upsert);
public bool            getSendUpsert();
public void             setTimeScaleRate(double scale);
public double           getTimeScaleRate();
public int32_t          start();
public int64_t          getNumRecordsPlayedBack();
public int              getPercentPlayedBack();
public int32_t          stop();
```

Details:

`getName()` returns the identifier assigned to this instance of `SpPlayback` object
`setSendUpsert(bool)` whether to convert INSERT ops in the data to UPSERT
`getSendUpsert()` returns the current setting of UPSERT flag
`setTimeScaleRate(double)` is a double to control the rate of playback
`getTimeScaleRate()` returns the current value of the scale factor
`start()` spawns a background thread that starts the playback process. Returns 0 on success
`getNumRecordsPlayedBack()` returns the number of data records played back so far
`getPercentPlayedBack()` returns the percentage of the data played back so far
`stop()` terminates the background playback thread and closes connections to the Sybase Aleri Streaming Platform

B.4. Other C++ API Classes/Methods

Here are a few miscellaneous classes that were briefly referenced in earlier examples. One such class is the `SpUtils` class. This class stores the following static utility methods:

```
std::string getErrorMessage(int errorCode)
std::string getEventTypeName(int eventType)
std::string getEventIdName(int eventId)
```

Details:

- The `getErrorMessage(int errorCode)` method is used to retrieve the message `std::string`, associated with the `errorCode` passed in through the parameter list. Typically, the `errorCode` is returned by a previous call to one of the Pub/Sub API methods.
- The `getEventTypeName(int eventType)` method is typically called by an `SpObserver` object. It returns an `std::string` representing the literal name of the `eventType` passed in. The `eventType` is usually retrieved from an `SpSubscriptionEvent` object that was delivered to the `SpObserver` through the `SpObserver` object's `notify(...)` method.
- The `getEventIdName(int eventId)` method is also typically called by an `SpObserver` object. It returns an `std::string` representing the literal name of the `eventId` passed in. The `eventId` is usually retrieved from an `SpSubscriptionEvent` object that was delivered to the `SpObserver` through the `SpObserver` object's `notify(...)` method.

Appendix C. Reference Guide to the .NET Object Model

C.1. Common Service Objects for .NET

C.1.1. SpFactory Object

The SpFactory object is used by the client code to create the set of objects that are required to use/control the Pub/Sub API. The SpFactory interface includes the following methods:

```
static int init();
static int dispose();
static SpPlatform ^createPlatform(SpPlatformParms ^parms,
    SpPlatformStatus ^status);
static SpPlatformParms ^createPlatformParms(System::String ^theHost,
    int thePort, System::String ^theUser, System::String ^thePassword,
    bool theEncryptedFlag);
static SpPlatformParms ^createPlatformParms(System::String ^theHost,
    int thePort, System::String ^theUser, System::String ^thePassword,
    bool theEncryptedFlag, bool theUserRsaFlag);
static SpPlatformParms ^createPlatformParms(System::String ^theHost,
    int thePort, System::String ^theUser, System::String ^thePassword,
    bool theEncryptedFlag, System::String ^theHotSpareHost,
    int theHotSparePort);
static SpPlatformParms ^createPlatformParms
    (String ^theHost, int thePort, String ^theUser, String ^thePassword,
    bool theEncryptedFlag, SpAuthType theAuthType, String ^theHotSpareHost, int the
static SpPlatformParms ^createPlatformParms(System::String ^theHost,
    int thePort, System::String ^theUser, System::String ^thePassword,
    bool theEncryptedFlag, bool theUserRsaFlag,
    System::String ^theHotSpareHost, int theHotSparePort);
static SpPlatformStatus ^createPlatformStatus();
static SpStreamDataRecord ^createStreamDataRecord(SpStream ^stream,
    array<Object ^> ^fieldData, int opCode, int flags,
    SpPlatformStatus ^status);
```

Details:

- The `createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status)` method returns an SpPlatform object if the Pub/Sub API was able to connect to the Sybase Aleri Streaming Platform and properly initialize.

You have to use one of the overloaded SpFactory.createPlatformParms(...) methods, and the SpFactory.createPlatformStatus() method to create the two parameters required by the SpFactory.createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status) method. The contents of the *SpPlatformParms* parameter controls how the connection and authentication from the Pub/Sub API to the Sybase Aleri Streaming Platform takes place. If the connection cannot be established, the `createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status)` method returns null, and a non-zero error code is set within the SpPlatformStatus object See [Section C.1.3, “SpPlatformStatus Object”](#) for information on how to retrieve the error code/message.

- The `createPlatformParms(System::String ^theHost, int thePort, System::String ^theUser, System::String ^thePassword, bool theEncryptedFlag)` method returns a SpPlatformParms object that is ultimately passed as the first parameter to the SpFactory.createPlatform(SpPlatformParms ^parms, Sp-

`PlatformStatus status`) method. This `createPlatformParms` method call set up the basic connectivity, with the username/password for authentication. If *theEncryptedFlag* is set to `true`, then https will be used to connect to the Sybase Aleri Streaming Platform's Command and Control process and SSL socket connections will be made to the Sybase Aleri Streaming Platform's Gateway I/O process. If *theEncryptedFlag* is set to `false`, then http will be used to connect to the Sybase Aleri Streaming Platform's Command and Control process and regular socket connections will be made to the Sybase Aleri Streaming Platform's Gateway I/O process.

- The `createPlatformParms(System::String ^theHost, int thePort, System::String ^theUser, System::String ^thePassword, bool theEncryptedFlag, bool theUseRsaFlag)` method returns an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory.createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status)` method. In addition to the basic connectivity parameters mentioned above, this method adds an additional boolean flag called *theUseRsaFlag*. If this flag is set to `true`, the Pub/Sub API will attempt to authenticate to the Sybase Aleri Streaming Platform using the RSA mechanism. To use this mechanism, the Sybase Aleri Streaming Platform must be started with the `-k` option indicating the directory where your public RSA key file is stored. See the *Administrators Guide* for information about key generation and placement.

When using the RSA authentication mechanism, the password of the `SpPlatformParms` object must specify your private RSA key file. For example, if a user was named `foo`, there would be two RSA key files having the names `foo` and `foo.private.der`, where `foo` is a file containing the public RSA key for user "foo", and `foo.private.der` is a file containing the private RSA key for user `foo`. The public RSA key file called `foo` must be placed in a directory that is specified by the `-k` option to the Sybase Aleri Streaming Platform during startup.

The private RSA key file called `foo.private.der` must be placed on the client machine using the Pub/Sub API to connect to the server and specified using the password parameter of the `createPlatformParms(...)` method.

There are five variations of the `createPlatformParams` method that that accomplish the same creation of an `SpPlatformParams` object, so choose the one that suits your needs.

- basic
- basic with *UseRSA* flag
- basic with *HotSpare*
- *HotSpare* with *UseRSA*
- Kerberos authentication with or without the *Hot Spare*
- The `createPlatformParms(System::String ^theHost, int thePort, System::String ^theUser, System::String ^thePassword, bool theEncryptedFlag, System::String ^theHotSpareHost, int theHotSparePort)` method returns an `SpPlatformParms` object that is ultimately passed as the first parameter to the `SpFactory.createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status)` method. In addition to the basic connectivity parameters previously mentioned, this method adds two more parameters called *String ^theHotSpareHost* and *int theHotSparePort*. Using an `SpPlatformParms` object created with this factory method will cause the Pub/Sub API to use a High Availability configuration. In a High Availability configuration, if the primary Sybase Aleri Streaming Platform goes down, the Pub/Sub API will automatically attempt to switch over and use the secondary one. See the *Administrators Guide* for setting up a High Availability configuration. See [Section 2.4.6, "Publication/Subscription in a High Availability \(Hot Spare\) Configuration"](#) for more information.
- The `createPlatformParms(System::String ^theHost, int thePort, Sys-`

tem::String ^theUser, System::String ^thePassword, bool theEncryptedFlag, bool theUseRsaFlag, System::String ^theHotSpareHost, int ^theHotSparePort) method returns an SpPlatformParms object that is ultimately passed as the first parameter to the SpFactory.createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status) method. This method lets you set up the Pub/Sub API for RSA authentication and High Availability (Hot Spare).

- The createPlatformStatus() method returns an SpPlatformStatus object passed as the second. See the SpPublication ^parms, SpPlatformStatus ^status) method in order to return status information back to the caller. It's used in several other methods within the Pub/Sub API needed to return error code/status information. See [Section A.1.3, “SpPlatformStatus Object”](#) for more information.
- The overloaded function createPlatformParms(String ^theHost, int thePort, String ^theUser, String ^thePassword, bool theEncryptedFlag, SpAuthType theAuthType, String ^theHotSpareHost, int theHotSparePort) accepts a parameter of type SpAuthType. This can be any one of the following values: AUTH_NONE, AUTH_PAM, AUTH_RSA, or AUTH_KERBV5. While other versions of the factory method can be used, this is the preferred way of creating an SpPlatformParms object. If no hotspare configuration exists, clients should pass in a null value for the **theHotSpareHost** parameter.
- The createStreamDataRecord(SpStream ^stream, array<Object ^> ^fieldData, int opCode, int flags, SpPlatformStatus ^status) method returns a SpStreamDataRecord object that is used in the SpPublication object in order to publish data to the Sybase Aleri Streaming Platform server.

The SpPlatformStatus object is passed in as the last parameter, if the createStreamDataRecord fails, a null will be returned to the caller and the SpPlatformStatus object will indicate the error condition.

C.1.2. The SpPlatformParms Object

The SpPlatformParms object is used by the SpFactory.createPlatform(SpPlatformParms ^parms, SpPlatformStatus ^status) method to create the SpPlatform object. The SpPlatformParms object is created using one of the overloaded SpFactory.createPlatformParms(...) methods previously described. The SpPlatformParms object contains all of the connection information required by the SpPlatform object in order to make the appropriate connection(s) to the Sybase Aleri Streaming Platform. This information includes the host and port of the Sybase Aleri Streaming Platform's Command and Control Process, username, password, and flags indicating whether to use encryption, or RSA authentication, Kerberos authentication, or the High Availability (Hot Spare) mechanism. The SpPlatformParms interface includes the following methods:

```
System::String ^getHost();
int getPort();
System::String ^getUser();
System::String ^getPassword();
bool isEncrypted();
System::String ^getHotSpareHost();
int getHotSparePort();
bool useRsa();
SpAuthType getAuthentication();
```

Details:

- The `getHost()` method returns a string indicating the host name of the machine running the Sybase Aleri Streaming Platform server's Command and Control process.
- The `getPort()` method returns an integer indicating the port number of the Sybase Aleri Streaming Platform server's Command and Control process.
- The `getUser()` method returns a string indicating the name used to authenticate to the Sybase Aleri Streaming Platform.
- The `getPassword()` method returns a string containing the password used to authenticate to the Sybase Aleri Streaming Platform. For RSA authentication, the password contains the file name of the user's private RSA key file.
- The `isEncrypted()` method returns a boolean indicating whether or not encrypted connections will be used to the Command and Control process and the Gateway I/O process. If the encryption mechanism is enabled, the Command and Control process connection will be made using https, while the Gateway I/O process will make SSL socket connections.
- The `getHotSpareHost()` method returns a string containing the host name of the secondary High Availability Sybase Aleri Streaming Platform. See the *Administrators Guide* for more information about setting up a High Availability configuration.
- The `getHotSparePort()` method returns an integer containing the port number of the secondary High Availability Sybase Aleri Streaming Platform. See the *Administrators Guide* for more information about setting up a High Availability configuration.
- The `useRsa()` method returns a boolean indicating whether or not RSA authentication is used when attempting to make connections to the Sybase Aleri Streaming Platform Command and Control process, and the Gateway I/O process.
- The `getAuthentication` method returns the authentication mechanism specified when the `SpPlatformParms` was created.

C.1.3. SpPlatformStatus Object

The `SpPlatformStatus` object is used by several of the Pub/Sub API methods to return status information to the caller. The `SpPlatformStatus` interface includes the following methods:

```
int getErrorCode();
System::String ^getErrorMessage();
bool isError();
```

Details:

- The `getErrorCode()` method returns an integer. If a problem was detected by the method this `SpPlatformStatus` object was passed into, a non-zero error return code value is returned, otherwise a zero is returned to indicate success.
- The `getErrorMessage()` method returns a string containing the error message text.
- The `isError()` method returns a boolean which is `true` if an error was detected or `false` for no error.

C.1.4. SpPlatform Object

The notion of the Sybase Aleri Streaming Platform has been abstracted into an object of the `SpPlatform` form type.

An `SpPlatform` object is created using the `SpFactory.createPlatform`. Once instantiated, an `SpPlatform` object implements and offers the the following Sybase Aleri Streaming Platform functionality:

```
System::String ^getUrl();
System::String ^getUser();
System::String ^getPassword();
System::String ^getHost();
System::String ^getGatewayHost();
System::String ^getXMLModelVersion();
int getPort();
int getGatewayPort();
int getDateSize();
int getAddressSize();
int getQuiesced();
int getPrimaryServerFlag();
cli::array<alери_PubSubnet::SpStream ^> ^getBaseStreams();
cli::array<alери_PubSubnet::SpStream ^> ^getDerivedStreams();
cli::array<alери_PubSubnet::SpStream ^> ^getStreams();
alери_PubSubnet::SpStream ^getStream(System::String streamName);
alери_PubSubnet::SpStream ^getStream(int streamId);
alери_PubSubnet::SpStreamDefinition ^getStreamDefinition(
    System::String ^streamName);
alери_PubSubnet::SpStreamDefinition ^getStreamDefinition(int streamId);
bool isBigEndian();
bool isConnected();
bool isEncrypted();
bool useRsa();
int shutdown();

String ^getConfig(SpPlatformStatus ^status);
int loadServerConfigFile(String ^configFile, String ^flags);
int loadConfigString(String ^configString, String ^flags);
int loadConfigStringApplyingConversion(String ^configString, String ^flags, String ^conversion);

int addStreamToClient(int clientHandle, System::String ^streamName);
int removeStreamFromClient(int clientHandle, System::String ^streamName);

alери_PubSubnet::SpSubscription ^createSubscription(
    System::String ^name, int flags, int deliveryType,
    alери_PubSubnet::SpPlatformStatus ^status);

SpSubscriptionProjection ^createSubscriptionProjection(String ^name,
    int flags, int deliveryType, String ^sqlQuery,
    SpPlatformStatus ^status);

alери_PubSubnet::SpPublication ^createPublication(System::String ^name,
    alери_PubSubnet::SpPlatformStatus ^status);
```

Most methods provided by the `SpPlatform` object communicate internally with the Sybase Aleri Streaming Platform Command and Control process through the XMLRPC protocol. The `SpPlatform` method lets you retrieve Sybase Aleri Streaming Platform configuration information, all the streams and so forth.

Details of this method set:

The `getUrl()` method returns the context of a string representing the URL, which is used to connect to the Command and Control Process through XMLRPC. This string depends on whether the `SpPlatform` object was created with encryption enabled.

The `isEncrypted()` method can check if encryption was enabled when the `SpPlatform` object

was instantiated.

The `getUser()` and `getPassword()` methods return the strings that represent the username and password. These values are used internally when authentication takes place while connecting to the Sybase Aleri Streaming Platform Command and Control and Gateway I/O processes.

There is a set of methods consisting of `getHost()`, `getGatewayHost()`, `getPort()` and `getGatewayPort()`. `getHost()` returns the name of the host machine where the Sybase Aleri Streaming Platform Command and Control Process is running. `getGatewayHost()` displays the host machine where the Sybase Aleri Streaming Platform Gateway I/O Process is running. These two Sybase Aleri Streaming Platform processes reside on the same machine.

The `getPort()` and `getGatewayPort()` methods respectively return the Command and Control and Gateway I/O port numbers. Unlike the `getHost()` and `getGatewayHost()` commands, the values returned by these two functions will differ because they refer to two separate processes.

The `getXMLModelVersion()` method returns a string indicating the version of XML model with which the Sybase Aleri Streaming Platform started up.

The `getDateSize()` method returns the size of the datetime field type. If the Pub/Sub API is used to communicate with the Gateway I/O process, the datetime field type size is automatically fixed.

The `getAddressSize()` method returns the size of a C/C++ pointer (in bytes) that the Sybase Aleri Streaming Platform server currently recognizes. The value represents how the instance of the running Sybase Aleri Streaming Platform Server was compiled (either 32-bit or 64-bit).

The `getQuiesced()` method returns an integer that represents the "quiesced" state of the Sybase Aleri Streaming Platform. If successful, the method will return either a zero to indicate *false*, or a 1 to indicate *true*. If the command is not executed successfully, an error code is returned.

The error message associated with the error code can be retrieved by calling `SpUtils.getErrorMessage(rc)`, where `rc` is the return code sent back from the `getQuiesced()` call.

The `getPrimaryServerFlag()` method returns an integer. If a value of 1 is returned, the Sybase Aleri Streaming Platform server is considered to be the primary server in a High Availability (Hot Spare) configuration. If a value of zero is returned, it is not the primary server. If the command could not be executed successfully, an error code is returned that is neither a zero nor a 1.

You can use the Pub/Sub API method to check that the connected Sybase Aleri Streaming Platform is a primary server within a High Availability (Hot Spare) configuration. Theoretically, you could use the Pub/Sub API to establish a connection to the secondary server within this configuration. Calling the `getPrimaryServerFlag()` method on the server will return a value of zero, indicating that the server is not the primary.

The next group of methods returns stream metadata from the Sybase Aleri Streaming Platform. The metadata/schema for a stream is represented within the Pub/Sub API as an object of type `SpStream`. Refer to [Section C.1.5, "SpStream Object"](#) for more information. The `getBaseStreams()` method returns an array of `SpStream` objects representing all of the source streams residing on the Sybase Aleri Streaming Platform. Similarly, `getDerivedStreams()` returns an array of `SpStream` objects that represent all of the derived streams residing on the Sybase Aleri Streaming Platform. The `getStreams()` method returns an array of `SpStream` objects that represent all streams (both source streams and derived streams) residing on the Sybase Aleri Streaming Platform. You can look up a particular stream by its *name* or *id* using the `getStream(System::String ^streamName)` or `getStream(int streamId)` method, respectively.

The `getStreamDefinition(System::String ^streamName)` and `getStreamDefinition(int streamId)` methods respectively return the handle to an object of type `SpStreamDefinition` for the specified *streamName* or *streamId*. Refer to [Section C.1.6, "SpStreamDefinition Object"](#) for more

information.

The `isBigEndian()` method returns `true` if the Sybase Aleri Streaming Platform server is running on a big-endian machine, `false` if the Sybase Aleri Streaming Platform server is running on a little-endian machine.

The `isConnected()` method returns `true` if the `SpPlatform` object is still connected to the Sybase Aleri Streaming Platform, `false` otherwise. For example, after you issue a shutdown, subsequent `isConnected()` calls return `false`.

Once an `SpPlatform` object is shut down, you should set its reference to null. Later on, another `SpPlatform` object can be instantiated again using the `SpFactory.createPlatform(...)` method.

The `shutdown()` method tells the Command and Control Process to shut down the Sybase Aleri Streaming Platform. This causes all socket connections to the Sybase Aleri Streaming Platform to be closed. If the application program has subscriptions running at the time of the shutdown, the `SpObject` objects of those subscriptions will be notified before the shutdown. Refer to [Section C.2, “Subscription Objects for .NET”](#) for more information.

The `getConfig(SpPlatformStatus ^status)` method returns a string containing the XML configuration currently being executed by the running Sybase Aleri Streaming Platform instance. If there is an error in retrieving the XML configuration information from the server, this method will return an empty string, and the error code will be stored in the `SpPlatformStatus` parameter passed into the method.

The `loadServerConfigFile(String ^configFile, String ^flags)` method attempts to load the XML configuration file that is located on the server into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information used during the XML configuration file load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string. Consult the *Administrators Guide* for more information on loading XML configurations, and the various options that can be specified in the `flags` parameter. If the XML configuration file was loaded successfully, the method returns zero. If it was unsuccessful, the method will return a non-zero error return code. You can get additional information from the log messages located on the server when loading an XML configuration file into the server.

The `loadConfigString(String ^configString, String ^flags)` method attempts to load the XML configuration stored in the `configString` parameter into the running Sybase Aleri Streaming Platform instance. The `P` parameter provides control information used during the XML configuration string load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string. Consult the *Administrators Guide* for more information on loading XML configurations and various options that can be specified in the `flags` parameter. If the XML configuration was loaded successfully, the method returns zero. If it was unsuccessful, the method will return a non-zero error code. You can get additional information from the log messages located on the server when loading an XML configuration file into the server.

The `loadConfigStringApplyingConversion(String ^configString, String ^flags, String ^conversionConfigString)` method attempts to load the XML configuration stored in the `configString` parameter into the running Sybase Aleri Streaming Platform instance. The `flags` parameter is used to provide control information used during the XML configuration string load attempt. If additional control information is not needed, the value of the `flags` parameter can be an empty string.

The `conversionConfigString` parameter is used to provide an XML model that is used to apply specific conversion instructions during the XML configuration load. See the *Administrators Guide* for more information on loading XML configurations, and the various options that can be specified in the `conversionConfigString` parameter.

If the XML configuration was loaded successfully, the method returns zero. If it was unsuccessful, the

method returns a non-zero error code. You can get additional information from the log messages located on the server when loading an XML configuration file into the server.

The `SpPlatform` object provides two subscription-related methods that you can use to disregard the Pub/Sub API subscription mechanism and write your own low-level Gateway I/O code for the subscription. These methods are `addStreamToClient(int clientHandle, System::String ^streamName)` and `removeStreamFromClient(int clientHandle, System::String ^streamName)`. The two methods are part of the `SpPlatform` interface since both are XMLRPC calls that are used to manage the subscription characteristics of a Gateway I/O socket on which a subscription is currently running.

Note:

Once a subscription request is issued for an open Gateway I/O socket connection, the connection becomes a read-only connection. Asynchronous stream updates are delivered from the Sybase Aleri Streaming Platform to the client. Because of the "read-only" nature of the socket, additional Gateway I/O commands can no longer be issued on this socket connection. The XMLRPC mechanism must be used to do this.

While using the Pub/Sub API subscription mechanism, the `addStreamToClient` and `removeStreamFromClient` method calls are not required. However, these must be provided when the Pub/Sub API subscription mechanism is not being used.

The two methods are passed a *clientHandle*. The *clientHandle* is an integer value that is returned by the Gateway I/O process when you send a low-level subscription request on the socket. The `addStreamToClient(...)` method lets you add an additional stream to the subscription list, while the `removeStreamFromClient(...)` method lets you delete a stream from the subscription list.

If the Pub/Sub API subscription mechanism is to be used to get asynchronous stream updates from the Sybase Aleri Streaming Platform, then `SpPlatform` can be used to create a subscription. There are two forms of subscription objects that can be created using `SpPlatform`. The first is an `SpSubscription` object, which is created using the `createSubscription(System::String ^name, int flags, int deliveryType, aleri_PubSubnet::SpPlatformStatus ^status)` method. The second is created using the `SpSubscriptionProjection ^createSubscriptionProjection(String ^name, int flags, int deliveryType, String ^sqlQuery, SpPlatformStatus ^status)` factory method. Refer to [Section C.2, "Subscription Objects for .NET"](#) for the meaning of each parameter. Similarly, the `SpPlatform` object can be used to publish data to the Sybase Aleri Streaming Platform. To accomplish this, there is a factory method called `createPublication(System::String ^name, aleri_PubSubnet::SpPlatformStatus ^status)` that creates an `SpPublication` object on your behalf. An `SpPublication` object is used to "publish" stream input data and/or issue the Gateway I/O "commit()" command) from the client application to the Sybase Aleri Streaming Platform. Refer to [Section 4.3.1, "Create Objects for SP Publication Using .NET 2.0"](#) for more information.

C.1.5. SpStream Object

The `SpStream` object is used to store the metadata associated with a stream residing on the Sybase Aleri Streaming Platform. The `SpStream` interface includes the following methods:

```
int getId();
aleri_PubSubnet::String ^getName();
bool isBase();
aleri_PubSubnet::SpStreamDefinition ^getDefinition();
```

Details:

- The `getId()` method returns an integer that represents the stream's internal identifier on the Sybase Aleri Streaming Platform.
- The `getName()` method returns a string that represents the name of the stream.
- The `isBase()` method returns `true` if the stream is a source stream, `false` otherwise.
- The `getDefinition()` method returns a handle/reference to an object of type `aleri_PubSubnet::SpStreamDefinition`. The `SpStreamDefinition` contains the schema information for a given stream. See [Section C.1.6, "SpStreamDefinition Object"](#) for more information.

C.1.6. SpStreamDefinition Object

The `SpStreamDefinition` object stores the schema associated with a stream residing on the Sybase Aleri Streaming Platform. The `SpStreamDefinition` interface has the following methods and constants defined within it:

```
int getNumColumns();
cli::array<System::String ^> ^getColumnNames();
cli::array<System::int ^> ^getColumnTypes();
cli::array<System::int ^> ^getKeyColumns();
cli::array<System::int ^> ^getKeyColumnVector();
bool isKeyColumn(int columnIndex);
```

Details:

- The `getNumColumns()` method returns the number of columns in the stream.
- The `getColumnNames()` method returns an array of `System::Strings` handles, where each string represents a column. The column names appear in the same order as they do in the Sybase Aleri Streaming Platform configuration file. This array's size equals the value returned from the `getNumColumns()` method.
- The `getColumnTypes()` method returns an array of integers, where each integer is a constant representing the field type of the corresponding column. The `aleri_pubsubconst::SpDataTypes` class contains a list of integer constants representing the various column types. This array's size equals the value returned from the `getNumColumns()` method.
- The `getKeyColumns()` method returns an array of integers. Each integer is the column index (rel 0) of a key column in the stream's field list. For example, if the stream has 10 columns, and the first three are key columns, the `getKeyColumns()` method will return an array that includes the following entries: [0, 1, 2].
- The `getKeyColumnVector()` method returns an array of integers. Each field in the field list is represented by an integer, the value of which is either 1 if the field is a key field or 0 if it is not.
- The `isKeyColumn(int columnIndex)` returns a boolean value of `true` if the column index specified is that of a key field, otherwise it returns `false`. The `columnIndex` is "rel-0" as the first column of the field list has an index value of zero.

C.1.7. SpStreamProjection Object

The `SpStreamProjection` object stores the metadata associated with a stream projection based on an SQL query supplied to the `createSubscriptionProjection(...)` factory method of the `SpPlatform` object. See [Section A.1.7, “SpStreamProjection Object”](#). The `SpStreamProjection` interface includes the following methods:

```
aleri_PubSubnet::SpStream ^getStream();  
aleri_PubSubnet::SpStreamDefinition ^getDefinition();
```

where:

- The `getStream()` method returns a reference to the underlying `SpStream` that the SQL query was projected onto.
- The `getDefinition()` method returns a handle/reference to an object of type `aleri_PubSubnet::SpStreamDefinition`, containing the schema information of the projection. This information is returned by the Sybase Aleri Streaming Platform when the SQL query associated with an `SpSubscriptionProjection` object is first created. See [Section C.1.6, “SpStreamDefinition Object”](#) for more information.

C.2. Subscription Objects for .NET

C.2.1. SpSubscriptionCommon Method Set

This interface defines the common set of methods used by the `SpSubscription` and `SpSubscriptionProjection` objects (henceforth referred to simply as Subscription). Typically there is no need for an application using this API to directly use this interface.

The `SpSubscriptionCommon` interface defines the following method set that is used by both subscription mechanisms:

```
System::String ^getName();  
int getFlags();  
int getDeliveryType();  
int getClientHandle();  
  
int removeObserver(int theCookie);  
  
int start();  
  
int stop();
```

The `getName()` method returns the name that you assigned to the subscription object when it was created with either the `SpPlatform`'s `createSubscription(...)` or `createSubscriptionProjection(...)` method.

Similarly, the `getFlags()` and `getDeliveryType()` methods return the flag settings and the delivery type specified in the `createSubscription` or `createSubscriptionProjection` object.

The `getClientHandle()` method returns an integer representing a *handle* that is assigned to the underlying subscription connection by the Sybase Aleri Streaming Platform. A valid *handle* is greater than zero. The value of the *clientHandle* is acquired from the Sybase Aleri Streaming Platform when the `SpSubscription` or `SpSubscriptionProjection` is started through the `start()`

method.

The `removeObserver(int theCookie)` method is used to remove the `SpObserver` from the subscription's delivery mechanism. The two types of subscriptions have different ways to add observers, which are discussed in the `SpSubscription` and `SpSubscriptionProjection` interfaces.

The `start()` method is used to start the subscription mechanism.

There must be at least one stream and `SpObserver` registered with the `Subscription` object before the `Subscription` object can be started up through the `start()` method.

When you start up an `SpSubscription` object, the following sequence of events takes place:

1. The `SpSubscription` object establishes a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process and authentication is performed.
2. A subscription request is sent to the Sybase Aleri Streaming Platform on this socket connection.
3. If the subscription request is accepted by the Sybase Aleri Streaming Platform, the `Subscription` object reads the "clientHandle" that the Sybase Aleri Streaming Platform assigned to this subscription request.
4. A new thread is started up, and it is dedicated to reading stream update information through the read-only Gateway I/O socket connection. When your `SpObserver` objects are "notified" about the stream updates, through `notify(...)` methods, the `SpObserver` objects will be running within the context of this thread instead of the main one.
5. Stream update messages flowing from the Sybase Aleri Streaming Platform to the client are read, parsed and delivered to your `SpObserver` objects.
6. The `start()` method returns a zero to the caller indicating that the subscription was started successfully, and a non-zero value if an error occurs. The `SpUtils.getErrorMessage(errorCode)` method can be used to get the specific error message.

The `stop()` method shuts down the subscription mechanism by closing the socket connection and stopping the thread used to read, parse, and deliver Sybase Aleri Streaming Platform updates to the `SpObserver` objects.

Here are some additional interfaces:

```
void setPulseInterval(unsigned int pulseInterval);
unsigned int getPulseInterval();
void setQueueSize(int queue, SpPlatformStatus ^status);
int getQueueSize();
void setBaseDrainTimeout(int millis, SpPlatformStatus ^status);
int getBaseDrainTimeout();
void setExitOnClose(SpPlatformStatus ^status);
bool getExitOnClose();
```

- `setPulseInterval` can be used to set the pulse interval in seconds if the subscription was created with the pulsed flag on.
- `getPulseInterval` is used to retrieve the current setting of the pulse interval in seconds.
- `setQueueSize` is used to set the internal buffer size in the Sybase Aleri Streaming Platform for this subscription. The Sybase Aleri Streaming Platform uses this buffer to queue up messages if the

subscriber is slow in retrieving them. It can prevent the subscriber from blocking and slowing down the Sybase Aleri Streaming Platform. The setting is made when the subscription is started. It is necessary to keep the *status* parameter valid until the time the start call is made.

- `getQueueSize` retrieves the current value of the queue size.
- `setBaseDrainTimeout` is used to set the time in milliseconds that the Sybase Aleri Streaming Platform should wait before dropping a blocked subscription. If a subscription is started with the *DROPPABLE* flag set, the Sybase Aleri Streaming Platform closes a subscription connection if the messages block is due to a slow client. This parameter specifies how long to wait before closing the connection. The setting is made when the subscription is started, and you must keep the *status* parameter valid until the time the start call is made.
- `getBaseDrainTimeout` retrieves the current value in milliseconds of the base drain timeout.
- If `setExitOnClose` is set, the Sybase Aleri Streaming Platform will shut down once this subscription connection is closed by the client. The setting is made when the subscription is started, and you must keep the *status* parameter valid until the time the start call is made.
- `getExitOnClose` retrieves the current setting of the exit on close flag.

C.2.2. SpSubscriptionEvent

An `SpSubscriptionEvent` object encapsulates an event received from the Sybase Aleri Streaming Platform. It provides the following method set:

```
System::String ^getSubName();
int getType();
System::String ^getTypeName();
int getId();
System::String ^getIdName();
int getStreamId();
int getStreamOpCode();
cli::array<Object ^> ^getData();
```

The `getSubName()` method returns a string that represents the name of the subscription object that generated and delivered this event to the `SpObserver`. This “name” was assigned to the subscription object when it was first created through the `SpPlatform createSubscription(...)` or `createSubscriptionProjection(...)` method.

The `getType()` method returns an integer representing the “type” of this `SpSubscriptionEvent`. Currently there are four “types” (or categories) of events defined in the `alери_PubSubconst` namespace as follows:

- `SpEventType.PARSED_DATA`

It is delivered from a `Subscription` object created with a delivery type of `SpDeliveryType.DELIVER_PARSED`.

- `SpEventType.BINARY_DATA`

It is delivered from a `Subscription` object created with a delivery type of `SpDeliveryType.DELIVER_BINARY`.

- `SpEventType.STREAM_OPCODE_DATA`

It is delivered from a *Subscription* object created with a delivery type of *SpDeliveryType.DELIVER_STREAM_OPCODES*.

- *SpEventType.SYSTEM*

It is delivered by the *Subscription* object. The event indicates a system event has occurred, such as an error, halt in communication, or shut down of the Sybase Aleri Streaming Platform.

The *getTypeName()* method returns the string literal value, representing the type of event instead of the internal integer representation. You can use this value for output messages.

The *getId()* method returns an integer representing a unique event ID that can be safely used within a switch statement to “case” on. The event IDs are unique across the entire set of event types. For example, an *SpSubscriptionEvent* may have a “*getType()*” of *SpEventType.SYSTEM*, which means it is a system-related notification. The *getId()* method returns what was actually detected by the system (for example, *SpEventId.PARSING_ERROR*, *SpEventId.COMMUNICATOR_HALTED*, and so forth). As is the case with the event types, all of the event IDs are enumerated within the *alери_PubSubConst* namespace.

The subscription event identifiers are:

- *SpEventId.GATEWAY_SYNC_START*

This event is delivered to the *SpObserver* if the *Subscription* object is sent a *START_SYNC* Gateway I/O message from the Sybase Aleri Streaming Platform. The *START_SYNC* message contains the ID for the stream with which the message is associated.

If this event is delivered to the *SpObserver*, it indicates the start of the stream's “snapshot”. Subsequent events should be *INSERT* messages for each record in the stream until the *END_SYNC* Gateway I/O message is received from the Sybase Aleri Streaming Platform. A call to the *START_SYNC* event's *getData()* method returns null. For this message to be sent from the Sybase Aleri Streaming Platform to the client application, the *Subscription* has to be created with the *SpSubFlags.BASE* flag specified. If the *SpSubFlags.NO_BASE* flag is instead specified, the *START_SYNC* message will never have been delivered from the Sybase Aleri Streaming Platform to the client application.

This event is also delivered after the *WIPEOUT* event, if any dynamic Sybase Aleri Streaming Platform changes result in content regeneration. In this situation, the *START_SYNC* event is delivered even if the *Subscription* was created with the *SpSubscriptionCommon.NOBASE* flag.

- *SpEventId.GATEWAY_SYNC_END*

This event is delivered to the *SpObserver* if the *Subscription* is sent an *END_SYNC* Gateway I/O message from the Sybase Aleri Streaming Platform. The *END_SYNC* message contains the ID for the stream with which the message is associated.

If this event is delivered to the *SpObserver*, it indicates that the end of the stream's “snapshot” has been reached. A call to the *END_SYNC* event's *getData()* method returns null. For this message to be sent from the Sybase Aleri Streaming Platform to the client application, the *Subscription* either has to be created with the *SpSubFlags.BASE* flag specified, or any dynamic Sybase Aleri Streaming Platform changes results in the data in one or more of the relevant streams to be regenerated. In the latter case, this event is preceded by the *WIPEOUT* event and is followed by the *START_SYNC* event, and insertion of the generated data.

- *SpEventId.GATEWAY_WIPEOUT*

It is delivered to the `SpObserver` after any dynamic Sybase Aleri Streaming Platform changes that results in a relevant stream's content being regenerated. The event means that the whole current content of the stream is being discarded. The `WIPEOUT` event is followed by the `START_SYNC` event, insertion of the new data, and the `END_SYNC` event. All events are delivered even if the Subscription was created with the `SpSubscriptionCommon.NOBASE` flag.

- `SpEventId.BINARY_DATA`

If the Subscription is created with the delivery type of `SpDeliveryType.DELIVER_BINARY`, the `getData()` method returns an `SpBinaryData` object containing the binary message delivered from the Sybase Aleri Streaming Platform.

The binary data is located in the unmanaged heap. You need to access the unmanaged heap to manipulate the data. Your program should not free or delete this data since it's done automatically when the object goes out of scope.

- `SpEventId.PARSED_FIELD_DATA`

When a Subscription is created with a delivery type of `SpDeliveryType.DELIVER_PARSED`, it attempts to parse the field data of the stream messages transmitted by the Sybase Aleri Streaming Platform and delivers this parsed field information to the `SpObserver`. The `getStreamOpCode()` method can be used to determine whether the message was an `INSERT`, `UPDATE`, or `DELETE`. The parsed field data is accessed by the `SpObserver` through the event's `getData()` method.

- `SpEventId.PARSED_PARTIAL_FIELD_DATA`

This event is currently not supported in the .NET version of the API.

- `SpEventId.COMMUNICATOR_HALTED`

It is delivered when you attempt to issue a "shutdown" through the `SpPlatform` object. A call to the `getData()` method returns null.

- `SpEventId.PLATFORM_SHUTDOWN`

It is delivered when you attempt to issue a "shutdown" through the `SpPlatform` object. A call to the `getData()` method returns null.

- `SpEventId.PARSING_ERROR`

It is delivered when a parsing error is detected by the subscription, and there is at least some context to report. A call to the `getData()` method returns an object of type `SpParserReturnInfo`. Currently the .NET API does not support this object.

- `SpEventId.UNKNOWN_PARSING_ERROR`

It indicates that the parser encountered an unexpected error before the completion of the process. In this case, the `getData()` method returns an integer object that contains the record length for the message that is being parsed.

- `SpEventId.READ_STREAM_RECORD_ERROR`

It indicates that the parser could not successfully read the record that was delivered from the Sybase Aleri Streaming Platform. The `getData()` method returns an integer object containing the value of the record length for the bad record.

- `SpEventId.BAD_RECORD_LENGTH_ERROR`

It indicates that the record-length read off the socket was bad. The `getData()` method returns an integer object that contains the bad record length value read off the socket.

- `SpEventId.BAD_GATEWAY_OP_CODE_ERROR`

It indicates that the Gateway I/O operation code for the message sent from the Sybase Aleri Streaming Platform is invalid. The `getData()` method returns an integer object that contains the bad Gateway I/O operation code that was read.

- `SpEventId.EVID_HOT_SPARE_SWITCH_OVER_INITIATED`

It is delivered to the `SpObserver` when the Pub/Sub API recognizes that a connection attempt should be made to the High Availability (Hot Spare) server. The High Availability connection parameters were specified in the `SpPlatformParms` object passed to the `SpFactory.createPlatform()` method when the underlying `SpPlatform` was first created.

See [Section 4.5.1, “Publication/Subscription in a High Availability \(Hot Spare\) Configuration”](#) for more information.

- `SpEventId.EVID_HOT_SPARE_SWITCH_OVER_SUCCEEDED`

It is delivered to the `SpObserver` when the connection to the High Availability (Hot Spare) server is made successfully.

See [Section 4.5.1, “Publication/Subscription in a High Availability \(Hot Spare\) Configuration”](#) for more information.

- `SpEventId.EVID_HOT_SPARE_SWITCH_OVER_FAILED`

It is delivered to the `SpObserver` when the connection to the High Availability (Hot Spare) server fails.

See [Section 4.5.1, “Publication/Subscription in a High Availability \(Hot Spare\) Configuration”](#) for more information.

The `getIdName()` method returns the string literal value that corresponds to the numeric event ID. You can use this in output messages.

The `getStreamId()` method returns the stream id that is associated with this event. For example, an `SpObserver` may receive an event type of `SpEventType.PARSED_DATA`, where the event id is `SpEventId.PARSED_FIELD_DATA`, indicating that the event contains parsed field data. The `getStreamId()` method returns the stream id to which this event data corresponds.

The `getStreamOpCode()` method returns the stream operation code that is associated with this event. For example, an `SpObserver` may receive an event type of `SpEventType.PARSED_DATA`, where the event id is `SpEventId.PARSED_FIELD_DATA`, indicating that the event contains parsed field data. The `getStreamOpCode()` method returns a value that indicates whether the event is an *INSERT*, *UPDATE*, *DELETE*, and so forth.

The `getData()` method returns an array of objects representing the event data that the `SpObserver` should process. The selection of objects stored in the collection depend upon the delivery type that was specified when the `Subscription` object was first created. For example, if the delivery type of the `Subscription` is `SpDeliveryType.DELIVER_PARSED`, the `getData()` method returns a vector; each element in the vector is yet another vector that contains the field list of parsed objects produced by the subscription message parser.

The array size (number of elements) returned by the `getData()` method will be 1 when the Sybase Aleri

Streaming Platform delivers a non-transaction message. For example, it could be an isolated *INSERT*, *UPDATE* or *DELETE*). In the case a transaction message is delivered, the array size is equal to the number of messages in the transaction block.

It is also important to note that each transactional message sent from the Sybase Aleri Streaming Platform contains updates for an individual stream. In other words, a single message will not contain records for more than one stream.

If the Subscription is created using a delivery type of *SpDeliveryType.DELIVER_BINARY*, the *getData()* method returns a *ByteBuffer* that has the raw binary stream message within it. If the Sybase Aleri Streaming Platform sends a transaction block, the *SpBinaryData* object contains the entire transaction block. The *ByteBuffer* is located in the unmanaged memory, but your program does not need to free this memory explicitly; this is done automatically when the *SpBinaryData* object goes out of scope.

If the Subscription is created using a delivery type of *SpDeliveryType.DELIVER_STREAM_OPCODES*, the *getData()* method returns null. Use the *getStreamOpCode()* method in order to determine the stream operation code (*INSERT*, *UPDATE*, *DELETE*, *UPSERT*, and so forth).

C.3. Methods for Publication in .NET 2.0

C.3.1. SpPublication Method Set

An *SpPublication* object can be used to publish data to one or more streams. It implements the following interface:

```
System::String ^getName();

int start();

int publish(aleri_PubSubnet::SpStreamDataRecord ^streamRecord);

int publish(
    cli::array<aleri_PubSubnet::SpStreamDataRecord ^> ^streamRecords,
    int streamOpCodeOverride,
    int streamFlagOverride);

int publishTransaction(
    cli::array<aleri_PubSubnet::SpStreamDataRecord ^> ^streamRecords,
    int streamOpCodeOverride,
    int streamFlagsOverride,
    int maxRecordsPerBlock);

int publishEnvelope(
    cli::array<aleri_PubSubnet::SpStreamDataRecord ^> ^streamRecords,
    int streamOpCodeOverride,
    int streamFlagsOverride,
    int maxRecordsPerBlock);

public int commit();

public int stop();

int publish(
    cli::array<aleri_PubSubnet::SpStreamDataRecord ^>
    ^streamRecords,
    int streamOpCodeOverride,
    int streamFlagOverride,
    SpPlatformStatus ^status);
```



```
int publishTransaction(  
cli::array<aleri_PubSubnet::SpStreamDataRecord ^>  
^streamRecords,  
int streamOpCodeOverride,  
int streamFlagsOverride,  
int maxRecordsPerBlock,  
SpPlatformStatus ^status);  
  
int publishEnvelope(  
cli::array<aleri_PubSubnet::SpStreamDataRecord ^>  
^streamRecords,  
int streamOpCodeOverride,  
int streamFlagsOverride,  
int maxRecordsPerBlock,
```

Details:

The `getName()` method returns the “name” assigned to this `SpPublication` object when it was created through the `SpPlatform`'s `createPublication(...)` factory method.

The `start()` method is used to start the publication process.

When an `SpPublication` object is started, the following sequence of events take place:

1. The `SpPublication` object creates a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process.
2. The `SpPublication` authenticates with the Sybase Aleri Streaming Platform.
3. The `start()` method returns a zero to the caller when the `SpPublication` object was successfully started; otherwise, a non-zero error code is returned.

The `SpUtils.getErrorMessage(errorCode)` method can get the specific error message when an error occurs.

The publication mechanism does not create a separate thread to manage the publication, unlike the subscription method. Behind the scenes, a socket connection to the Sybase Aleri Streaming Platform Gateway I/O process transmits stream data to the Sybase Aleri Streaming Platform and reads the response associated with each individual request. A publication request is synchronous unless otherwise specified in the flag values used when publishing data. You can call one of the `publish` methods and wait for the Sybase Aleri Streaming Platform to respond with an `ack` or `nak`. However, there is a special stream flag, `SpGatewayConstants.SF_NOACK`, that can be used to make an asynchronous publication request. When this flag is specified, the `publish` method sends the request out to the Gateway I/O process and returns control immediately back to the caller without waiting for a response from the Sybase Aleri Streaming Platform.

The `publish(aleri_PubSubnet::SpStreamDataRecord ^streamRecord)` method is used to publish/send a single stream input record to a source stream on the Sybase Aleri Streaming Platform although it is more efficient to send Input Stream records to the Sybase Aleri Streaming Platform in either transaction blocks or envelopes. This method can be used initially when the programmer wants to test a new data model, perhaps by sending one stream input record at a time. If the send is a success, a return code of zero is sent back to the caller, otherwise an error code is sent back. The `SpUtils.getErrorMessage(errorCode)` method can be called to get the specific error message.

Each of the publish methods of the `SpPublication` object takes one or more `SpStreamDataRecord` objects as input. The `SpStreamDataRecord` object represents one row of stream data that will be sent to the Sybase Aleri Streaming Platform. Each `SpStreamDataRow` object has its own op code, which indicates how the row will be handled by the Sybase Aleri Streaming Platform when it's received.

For example, the op code may indicate that the row will be treated as an *INSERT*, a *DELETE*, an *UPDATE* and so forth. See [Section B.2.4, “SpStreamDataRecord Object”](#) for more information.

The `publish(cli::array<aleri_PubSubnet::SpStreamDataRecord ^> ^streamRecords, int streamOpCodeOverride, int streamFlagOverride)` sends an array of `SpStreamDataRecord` objects to the Sybase Aleri Streaming Platform with one call. The `streamOpCodeOverride` and `streamFlagOverride` parameters can be used to override the corresponding values found in the individual `SpStreamDataRecord` objects that comprise the collection.

Although an array of `SpStreamDataRecord` objects is sent/published to the Sybase Aleri Streaming Platform, this method sends each element of the array, one record at a time. However, this method allows you to create a set of stream data records where each record can be applied to a different stream. This may be used for a debugging or testing scenario, where the ordered sequence of updates to various source streams is important.

The `publishTransaction(cli::array<aleri_PubSubnet::SpStreamDataRecord ^> ^, int streamOpCodeOverride, int streamFlagsOverride, int maxRecordsPerBlock)` and the `publishEnvelope(cli::array<aleri_PubSubnet::SpStreamDataRecord ^> ^, int streamOpCodeOverride, int streamFlagsOverride, int maxRecordsPerBlock)` methods are the most efficient ways to bundle multiple `SpStreamDataRecord`s stored in the array to send at once to the Sybase Aleri Streaming Platform as a single batch.

Each `SpStreamDataRecord` to be placed within the transaction/envelope must be for the same source stream. It can have a different “op code”, such as *INSERT*, *UPDATE*, *DELETE*, but the records must be for the same source stream.

As previously mentioned, the override op code and flag values can override the corresponding values found within the individual `SpStreamDataRecord` objects that make up the collection. In addition, this method takes one more parameter called *maxRecordsPerBlock*, an integer value that specifies the maximum number of `SpStreamDataRecord`s to be sent as a transactional/envelope unit to the Sybase Aleri Streaming Platform. If the value is set to zero, then the method will try to send all of the `SpStreamDataRecord` objects in the collection to the Sybase Aleri Streaming Platform within one transaction block. If the *maxRecordsPerBlock* is less than the actual number of records in the collection, then the record set will be broken up using the *maxRecordsPerBlock* setting into multiple transactions/envelopes during transmission to the server.

The difference between a transaction and envelope block transmitted to the server is how the server processes the block of records upon receiving it. As the name implies, a group of records within a transaction block is treated as a single transactional unit on the server side. In the case of an envelope, the group of records contained within the envelope are processed a single record at a time by the server. Basically, the envelope mechanism allows the client to send a batch of records for a specific stream to the server in one shot as opposed to sending a single record at a time, and having to wait for each record's ack/nak reply from the server.

Each of the three publish functions described above has a version that accepts a handle to `SpPlatformStatus`. These versions are functionally similar, but in error conditions, return extended information if available in the `SpPlatformStatus`.

The `commit()` method issues a special Gateway I/O command to the Sybase Aleri Streaming Platform, requesting that all pending input records previously sent to the Sybase Aleri Streaming Platform be synced to disk. Making a commit call is a tremendously expensive operation relative to latency. It's

designed to be used only as part of a two-phase commit process when reading from a persistent source, such as an ActiveMQ series, and writing to a Sybase Aleri Streaming Platform instance that uses Sybase's log store persistence model.

In a real time, low latency streaming scenario, the commit call should not be used after each record.

Typically the commit call should be used as follows: For a standard two-phase commit process that guarantees against data loss, the client reads messages from the source, such as ActiveMQ, and publishes to the Sybase Aleri Streaming Platform until reaching a pre-determined number of processed messages (>1024 is recommended) or a specified amount of time has elapsed. After reaching the value set for the maximum number messages or the elapsed time has passed, the commit() call is made and upon return to the client, the client may inform the source, such as ActiveMQ, that the messages can be deleted.

The `stop()` method shuts down the underlying Gateway I/O socket connection.

C.3.2. Stream Operation Codes

The following lists the stream operation codes that can be set for each individual `SpStreamDataRecord` or as one of the "streamOpCodeOverride" parameters can be found in the `aleri_PubSubconst` namespace:

- `SpOpCodes.NOOP`

When specified as the "streamOpCodeOverride" parameter, this value indicates that the stream op code stored in each of the individual `SpStreamDataRecord` objects should be used by the Sybase Aleri Streaming Platform. If the stream op code within the `SpStreamDataRecord` is set to `SpOpCodes.NOOP`, then the Sybase Aleri Streaming Platform will default the stream operation to an INSERT operation.

- `SpOpCodes.INSERT`

When this value is specified, the Sybase Aleri Streaming Platform treats the published stream record as an INSERT operation.

- `SpOpCodes.UPDATE`

When this value is specified, the Sybase Aleri Streaming Platform treats the published stream record as an UPDATE operation.

- `SpOpCodes.DELETE`

When this value is specified, the Sybase Aleri Streaming Platform treats the published stream record as a DELETE operation.

- `SpOpCodes.UPSERT`

When this value is specified, the Sybase Aleri Streaming Platform treats the published stream record as an UPSERT operation. An UPSERT operation either inserts the stream record into the source stream if it is not already present or it updates the existing source stream record using the contents of the stream record.

C.3.3. Stream Flag Values

The following is a set of stream flag values that can be set for each individual `SpStreamDataRecord` or as one of the `streamFlagOverride` parameters found in the `aleri_PubSubconst` namespace:

- *SpStreamFlags.NULLFLAG*

When specified as the *streamFlagOverride* parameter, it indicates that the stream flag stored in each of the individual *SpStreamDataRecord* objects should be used by the Sybase Aleri Streaming Platform. If the stream flag within the *SpStreamDataRecord* is set to *SpStreamFlags.NULLFLAG*, the default synchronous publication sequence takes place where the *SpPublication* waits for a response from the Sybase Aleri Streaming Platform.

- *SpStreamFlags.NOACK*

When specified, it tells the Sybase Aleri Streaming Platform not to send an ack or nak back to the client application that issued the publication request. In other words, the *Publish* method that was called runs asynchronously, and it assumes that the record was received and processed by the Sybase Aleri Streaming Platform.

- *SpStreamFlags.SHINE*

It is only relevant for the stream op code values of *UPDATE* and *UPSERT*. Typically, all of the fields for a stream record being published to the Sybase Aleri Streaming Platform must be assigned values for each of the available stream operations (*INSERT*, *UPDATE*, *UPSERT*, and so forth). In the case of an *UPDATE* or *UPSERT*, you can use the *SHINE* flag to ignore any NULL columns in a record and update only the columns with actual new values. In essence, the Sybase Aleri Streaming Platform lets the existing field values “shine through” for each of the null values you sent in.

The flag values represent bits that can be ORed together as in the example below:

```
int flags = SpStreamFlags.SHINE | SpStreamFlags.NOACK
```

The key column(s) of the record to be updated must be populated for the update/upsert operation to succeed.

C.3.4. *SpStreamDataRecord* Object

Each of the *SpPublication* object's publishing methods sends stream input data from the client application to the Sybase Aleri Streaming Platform. Each stream record or row of stream data is encapsulated within an *SpStreamDataRecord* object, which has the following method set:

```
alери_PubSubnet::SpStream ^getStream();  
cli::array<System::Object ^> ^getFieldData();  
int getOpCode();  
int setOpCode(int value);  
int getFlags();  
int setFlags(int value);
```

Details:

- The *getStream()* method returns an *SpStream* object with which this *SpStreamDataRecord* is associated. See [Section C.1.5, “SpStream Object”](#) for more information.
- The *getFieldData()* method returns an array of objects representing the data for each field in the stream record. Currently, these objects can have a “type” of String, Integer, Long, Double, Date, Money, Timestamp, and null.

- The `getOpCode()` method returns the stream op code currently set for this record.
- The `setOpCode(int value)` method sets the value of the stream op code for this record.
- The `getFlags()` method returns the current flag settings for this record.
- The `setFlags(int value)` method sets the value of the stream flag settings for this record.

C.3.5. Creating SpStreamDataRecord Objects

An `SpStreamDataRecord` object is created using a “factory” method for consistency within the Pub/Sub API model with the following method signature:

```
alери_PubSubnet::SpStreamDataRecord  
^SpFactory.createStreamDataRecord(  
    alери_PubSubnet::SpStream ^stream,  
    cli::array<System::Object ^> ^fieldData,  
    int opCode,  
    int flags,  
    alери_PubSubnet::SpPlatformStatus ^status);
```

Details:

- *SpStream* stream is a handle to the `SpStream` object with which this new `SpStreamDataRecord` object will be associated. You can get this value through one of the appropriate `SpPlatform` methods, such as `getStream(System::String ^streamName)` or `getStream(int streamId)`.
- *cli::array<System::Object ^> ^fieldData* is an array of objects where each object entry in the array matches the corresponding field data type, as indicated in the streams definition specified in the `SpStream` parameter.

Notes:

When creating an `SpStreamDataRecord`, all of the key fields must be specified with non-null values within the *fieldData* array. In addition, the types of the objects that are located in the *fieldData* array must match those in the `SpStream` definition.

- *int opCode* is the stream operation code associated with this `SpStreamDataRecord`. The op code specifies how to apply this record to the source stream: *INSERT*, *UPDATE*, or *DELETE*.
- *int flags* is the stream flag settings value that is associated with this `SpStreamDataRecord`.

Note

Several of the publish methods include the option of overriding the stream op code and flag settings.

- *alери_PubSubnet::SpPlatformStatus ^status* is an object that returns error code information back from the `createStreamDataRecord (...)` factory method in the case where the `SpStreamDataRecord` object cannot be created.

C.3.6. Other Pub/Sub API Classes

Here are some of the miscellaneous classes briefly referenced in earlier examples. One class is the `SpUtils` class, which offers the following utility methods:

```
static System::String ^getErrorMessage(int errorCode)
static System::String ^getEventTypeName(int eventType)
static System::String ^getEventIdName(int eventId)
```

Details:

- The `getErrorMessage(int errorCode)` method retrieves the message string, associated with the `errorCode` passed in through the parameter list. Typically, the `errorCode` is returned by a previous call to one of the Pub/Sub API methods.
- The `getEventTypeName(int eventType)` method is typically called by an `SpObserver` object. This method returns a `String` representing the literal name of the `eventType` passed in. The `eventType` is usually retrieved from an `aleri_PubSubnet::SpSubscriptionEvent` object that was delivered to the `aleri_PubSubnet::SpObserver` through the `aleri_PubSubnet::SpObserver` object's `notify(...)` method.
- The `getEventIdName(int eventId)` method is also typically called by an `aleri_PubSubnet::SpObserver` object. It returns a string representing the literal name of the `eventId` passed in. The `eventId` is usually retrieved from an `aleri_PubSubnet::SpSubscriptionEvent` object that was delivered to the `aleri_PubSubnet::SpObserver` through the `aleri_PubSubnet::SpObserver` object's `notify(...)` method.

C.3.7. The `aleri_PubSubConst` namespace

The `aleri_PubSubConst` namespace contains all of the literal and constant values that can be used as parameters to various Pub/Sub API method calls.

The contents of the `aleri_PubSubConst` namespace is broken up into the following entities:

The `SpDataTypes` class contains the constants defining the various field data types supported by the Sybase Aleri Streaming Platform. The following field data types are supported:

- `DATE`
- `DOUBLE`
- `INT32`
- `INT64`
- `MONEY`
- `NULLVALUE`
- `STRING`
- `TIMESTAMP`

- The MONEY datatype is a 64-bit integer with an implicit decimal place. When receiving data from the Sybase Aleri Streaming Platform, the returned value should be passed as a parameter to the `platform.moneyToDouble(int64 money)` function to derive the correct double value. When sending data to the Sybase Aleri Streaming Platform, the double value should be converted to 64-bit integer using the `platform.doubleToMoney(double money)` function to ensure the data is processed correctly.
- The TIMESTAMP datatype is basically the same as a DATE datatype except that it is capable of holding milliseconds. When subscribing from the Sybase Aleri Streaming Platform or publishing data to the Sybase Aleri Streaming Platform a `System::DateTime` object can be appropriately received or sent. The API takes care of preserving or stripping out the millisecond component of the `DateTime` object depending on the datatype of the corresponding column in the Sybase Aleri Streaming Platform.
- The NULLVALUE datatype is meant for internal use. When subscribing to the Sybase Aleri Streaming Platform a null Object is retrieved when the value in a column is NULL. Similarly, when publishing data, a null Object must be sent to the API when a column value should be set to NULL within the Sybase Aleri Streaming Platform.

`SpDeliveryType` contains the different types of parsers assigned to the `SpSubscription` object. Currently, an `SpSubscription` object can be configured with a parser that returns just Stream Op Codes, parsed field data, binary data, and so on.

- DELIVER_BINARY
- DELIVER_PARSED
- DELIVER_STREAM_OPCODES

`SpEventId`: This class contains the different event identifiers that are returned from an `SpSubscription` object.

- BAD_GATEWAY_OP_CODE_ERROR
- BAD_RECORD_LENGTH_ERROR
- BINARY_DATA
- COMMUNICATOR_HALTED
- GATEWAY_SYNC_END
- GATEWAY_SYNC_START
- GATEWAY_WIPEOUT
- PARSED_FIELD_DATA
- PARSED_PARTIAL_FIELD_DATA
- PARSING_ERROR
- PLATFORM_SHUTDOWN
- READ_STREAM_RECORD_ERROR

- UNKNOWN_PARSING_ERROR

SpEventType contains various types and/or classifications of events that are returned from an SpSubscription object.

- BINARY_DATA
- PARSED_DATA
- STREAM_OPCODE_DATA
- SYSTEM

SpOpCodes contains various stream operation codes, which are used to set or determine the operation code of the record to be published or received via subscription.

- *DELETES*
- *INSERT*
- *NOOP*
- *UPDATE*
- *UPSERT*

SpStreamFlags contains various stream flags that can be used for publishing data.

- *NOACK*
- *NULLFLAG*
- *SHINE*

SpSubFlags contains various flags that can be specified for subscribing to data from the Sybase Aleri Streaming Platform.

- *BASE*
- *LOSSY*
- *NO_BASE*
- *DROPPABLE*
- *PRESERVE_BLOCKS*

C.4. Record and Playback objects for .NET

C.4.1. SpNetRecorder Object

To create an SpRecorder, client programs need to call the factory method - createRecorder - defined in SpPlatform.

```
SpRecorder^ createRecorder(System::String^ name, System::String^ filename, cli::array<System::String^>  
                           int flags, int maxRecords, aleri_pubsubnet::SpPlatformStatus^ status);
```

The method takes the following parameters:

name	This string uniquely identifies this instance of the recorder object.
filename	The name of the file where recorded data will be stored.
streams	This is a vector of strings containing the names of the streams for which to record events.
flags	<p>These flags control the underlying subscription. They can be a bit-wise OR of the following values:</p> <ul style="list-style-type: none">• one of SpSubFlags.BASE or SpSubFlags.NOBASE - indicates whether or not to record data already in streams at the time of connection• SpSubFlags.LOSSY indicates whether or not the Sybase Aleri Streaming Platform should discard records if the client application cannot keep up
maxRecords	Specifies the maximum number of records to process.
status	Specifies an SpPlatformStatus object to return information in case of error.

SpRecorder has the following public interface:

```
public System::String^ getName();  
public int start();  
public long getRecordCount();  
public int stop();
```

Where:

getName() returns the identifier assigned to this instance of the SpRecorder object.

start() spawns a background thread which starts the recording process. The method returns once the thread is started. Returns 0 on success.

getRecordCount() returns the number of data records processed.

stop() ends the recording process. by terminating the recording thread and closing connections to the Sybase Aleri Streaming Platform. Returns 0 on success.

C.4.2. SpNetPlayback object

An SpPlayback object is created by calling the following factory method defined in SpPlatform.

```
SpPlayback^ createPlayback(System::String^ name, System::String^ filename, double scale, int maxrecords,  
                           aleri_pubsubnet::SpPlatformStatus^ status);
```

The method takes the following parameters:

- `name` uniquely identifies this instance of `SpPlayback`.
- `filename` specifies the name of the file containing the recorded data.
- `scale` controls the rate of playback. Values -1 to 1 have no effect and the data is played back at the rate it was recorded at. Values greater than 1 speed up playback by that factor, for example, a value of 2 will play back twice as fast. Values less than -1 slows down playback by the factor specified.
- `maxrecords` specifies the maximum number of records to playback.
- `status` is an `SpPlatformStatus` object used to return information in case of error.

`SpPlayback` has the following public interface:

```
public System::String^  getName();
public void             setSendUpsert(bool upsert);
public bool             getSendUpsert();
public void             setTimeScaleRate(double scale);
public double           getTimeScaleRate();
public int              start();
public long             getNumRecordsPlayedBack();
public int              getPercentPlayedBack();
public int              stop();
```

Where:

<code>getName()</code>	Returns the identifier assigned to this instance of <code>SpPlayback</code> object.
<code>setSendUpsert(bool)</code>	Indicates whether to convert INSERT operations in the data to UPSERT operations.
<code>getSendUpsert()</code>	Returns the current setting of UPSERT flag.
<code>setTimeScaleRate(double)</code>	Controls the rate of playback.
<code>getTimeScaleRate()</code>	Returns the current value of the scale factor.
<code>start()</code>	Spawns a background thread that starts the playback process. Returns 0 on success.
<code>getNumRecordsPlayedBack()</code>	Returns the number of data records played back so far.
<code>getPercentPlayedBack()</code>	Returns the percentage of the data played back so far.
<code>stop()</code>	Terminates the background playback thread and closes connections to the Sybase Aleri Streaming Platform.

Appendix D. Reference Guide to SQL Query Interface

D.1. Aleri SQL Connectivity C++ Library

C++ programs can connect to the Sybase Aleri Streaming Platform using a C++ SQL connectivity library that was developed as a lightweight alternative to ODBC. This native C++ SQL interface to the Sybase Aleri Streaming Platform encapsulates the low-level messages into a more convenient API. The static library is included in the distribution in the file `lib/libsp_sql.a`.

The interface to the classes in this library is located in the distribution in the `include/NativeSql.hpp` file.

This API resembles a collection of stripped-down versions of the JDBC classes, which have been modified for C++. There are four main classes: `Connection`, `Statement`, `ResultSet`, and `ResultSetMetaData`.

The application program establishes a connection to the Sybase Aleri Streaming Platform using the `Connection` class. The constructor for objects sets up the host, port, database, user, and password. You must use the `open` member function after construction to connect to the Sybase Aleri Streaming Platform and then issue `createStatement` calls to create SQL statement objects.

```
class Connection
{
public:
    /// Constructor
    Connection(const char * host, int port, const char * db,
               const char * user, const char * pwd);

    /// Destructor
    virtual ~Connection();

    /// Open this Connection to the database.
    bool open();

    /// Open this Connection to the database using an SSL
    /// socket.
    bool openSSL();

    /// Release this Connection object's database and
    /// resources immediately instead of waiting for them
    /// to be automatically released.
    void close();

    /// Creates a Statement object for sending SQL statements
    /// to the database.
    Statement * createStatement();

    /// Retrieves whether this Connection object has been
    /// closed.
    bool isClosed();
};
```

The `Statement` class is used for execution of queries in the context of a `Connection`.

```
class Statement
{
public:
```

```
/// Constructor
Statement(Connection * connection);

/// Destructor
virtual ~Statement();

/// Cancels this Statement object if both the DBMS and
/// driver support aborting an SQL statement.
void cancel();

/// Releases this Statement object's database and other
///resources immediately instead of waiting for this to
///happen when it is automatically closed.
void close();

/// Executes the given SQL statement, which returns a
/// single ResultSet object.
ResultSet * executeQuery(const char * sql);

/// Retrieves the Connection object that produced this
/// Statement object.
Connection * getConnection();

/// Retrieves the maximum number of rows that a ResultSet
///object produced by this Statement object can contain.
int getMaxRows();

/// Sets the limit for the maximum number of rows that any
///ResultSet object can contain to the given number.
void setMaxRows(int max);
};
```

The `ResultSet` class is used to get information from the results of executing an SQL query on the Sybase Aleri Streaming Platform.

```
class ResultSet
{
public:
    /// Constructor
    ResultSet(Statement * statement, ResultSetMetaData * meta,
        std::vector<void *> & rows);

    /// Destructor
    ~ResultSet();

    /// Moves the cursor to the given row number in this
    ///ResultSet object.
    bool absolute(int row);

    /// Moves the cursor to the end of this ResultSet object,
    /// just after the last row.
    void afterLast();

    /// Moves the cursor to the front of this ResultSet object,
    ///just before the first row.
    void beforeFirst();

    /// Releases this ResultSet object's database and
    /// resources immediately instead of waiting for this to
    /// happen when it is automatically closed.
    void close();
};
```

```
// Maps the given ResultSet column name to its ResultSet
///column index.
int findColumn(const char * columnName);

// Moves the cursor to the first row in this ResultSet
// object.
bool first();

// Retrieves the value of the designated column in the
// current row of this ResultSet object as a time_t.
time_t getDate(int columnIndex);

/// Retrieves the value of the designated column in the
/// current row of this ResultSet object as a time_t.
time_t getDate(const char * columnName);
/// Retrieves the value of the designated column in the
/// current row of this ResultSet object as a double.
double getDouble(int columnIndex);

/// Retrieves the value of the designated column in the
current row of this ResultSet
/// object as a double.
double getDouble(const char * columnName);

/// Retrieves the value of the designated column in the
current row of this ResultSet object
/// as a 32-bit signed integer.
int32_t getInt32(int columnIndex);

/// Retrieves the value of the designated column in the
current row of this ResultSet
/// object as a 32-bit signed integer.
int32_t getInt32(const char * columnName);

/// Retrieves the value of the designated column in the
current row of this ResultSet object
/// as a 64-bit signed integer.
int64_t getInt64(int columnIndex);

/// Retrieves the value of the designated column in the
/// current row of this ResultSet object as a 64-bit
/// signed integer.
int64_t getInt64(const char * columnName);

/// Retrieves the number, types and properties of this
///ResultSet object's columns.
ResultSetMetaData * getMetaData();

// Retrieves the current row number.
int getRow();

/// Retrieves the Statement object that produced this
///ResultSet object.
Statement * getStatement();

/// Retrieves the value of the designated column in the
/// current row of this ResultSet object as a character
/// string.
const char * getString(int columnIndex);

/// Retrieves the value of the designated column in the
/// current row of this ResultSet object as a character
/// string.
```

```
const char * getString(const char * columnName);

/// Retrieves whether the cursor is after the last row in
/// this
ResultSet object.
bool isAfterLast();

/// Retrieves whether the cursor is before the first row
///in this ResultSet object.
bool isBeforeFirst();
/// Retrieves whether the cursor is on the first row of
///this ResultSet object.
bool isFirst();

/// Retrieves whether the cursor is on the last row of
/// this ResultSet object.
bool isLast();

/// Moves the cursor to the last row in this ResultSet
/// object.
bool last();

/// Moves the cursor down one row from its
/// current position.
bool next();

/// Moves the cursor to the previous row in this ResultSet
/// object.
bool previous();

/// Moves the cursor a relative number of rows, either
/// positive or negative.
bool relative(int rows);

/// Reports whether the last column read had a value of
/// SQL NULL.
bool wasNull();
};
```

The `ResultSetMetaData` class describes the format of a result set. It can be used to retrieve information about the column names and column types.

```
class ResultSetMetaData
{
public:
    /// Constructor
    ResultSetMetaData(std::vector<std::string> & colNames,
        std::vector<int> & colTypes);

    /// Destructor
    ~ResultSetMetaData();

    /// Returns the number of columns in this ResultSet object.
    int getColumnCount();

    /// Get the designated column's name.
    const char * getColumnName(int column);

    /// Get the designated column's position.
    int getColumnPos(const char * name);
};
```

```
/// Retrieves the designated column's SQL type.
int getColumnType(int column);

/// Retrieves the designated column's database-specific
/// type name.
const char * getColumnTypeName(int column);
};
```

Appendix E. Reference Guide to the Command and Control Interface

E.1. Command and Control Messages

All Command and Control functions have a 32-bit signed integer return code that follows the “C” language standard:

- A return code of zero indicates a successful function call.
- A non-zero return code indicates that an error has occurred.

Some of the calls return a structure in which the 32-bit signed integer return code is embedded as a structure member (usually named *Status*).

The following notation is used to specify a structure return value from a command and control function call:

```
s(Status) int
s(StreamNum) int
s(StreamNames) array strings
s(StreamIds) array int
```

This example indicates that the return value is an XMLRPC structure with data members *Status*, *StreamNum*, *StreamNames*, and *StreamIds*, whose respective data types are int, int, array of strings, and array of ints.

All the calls that return a structure include the field:

```
return: s(errMsgs): array string
```

If the return code is not zero, this array contains the error messages describing it. If the return code is zero, this array might still contain warning messages.

The supported Command and Control functions:

- `cimarron.getClockStopOnPause(token)` returns an integer that shows whether the logical platform clock will be stopped (value 1) or not (value 0) when the Sybase Aleri Streaming Platform pauses in the trace mode.

```
input: auth_token: string
return: s(status): int
return: s(value): int
return: s(errMsgs): array string
```

- `cimarron.setClockStopOnPause(token, value)` sets the flag that determines whether the logical Sybase Aleri Streaming Platform clock will be stopped (value 1) or not (value 0) when it

pauses in the trace mode.

```
input: auth_token: string
input: value: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.getClock(token)` returns the current status of the logical platform clock. The state consists of:

clocktime	clockTime is the current logical time in the Sybase Aleri Streaming Platform, in seconds since UNIX epoch.
clockRate	clockRate is the current rate of clock in the Sybase Aleri Streaming Platform relative to real time; 10 means "10 times faster", 0.1 means "10 times slower".
clockReal	A flag showing whether the clock is "real" (that is, matching the system time of the machine where the Sybase Aleri Streaming Platform runs (if 1), or it has been set artificially (if 0). If the status returned is not 0, this value may be -1, and in this case the rest of the returned values are invalid.
stopDepth	How many times the clock has been stopped recursively, meaning how many times <code>resumeClock()</code> would have to be called to actually resume the flow of time; when the clock is running, this value is 0.
maxSleep	maxSleep is a period of time, in real milliseconds, that guarantees all the sleepers discover the changes in the clock rate or time. The calls <code>setClockRate()</code> , <code>setClockRateTime()</code> , <code>setClockReal()</code> with argument <code>wait=1</code> use this value as wait length. If these calls are used with argument <code>wait=0</code> , the caller may use this value to sleep by itself. When running in real time, it uses a larger value (currently 1000, meaning 1 second) for more efficiency since every sleeping thread is waking up every so often; when running in variable time it uses a smaller value (currently 100) for faster reaction to the changes.

```
input: auth_token: string
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.setClockRate(token, double rate, int wait)` changes the rate at which the logical clock of the Sybase Aleri Streaming Platform ticks. The rate is relative to real time. For example, 10 means 10 times faster or 0.1 means "10 times slower".

It differs from the rate expressed in Pub/Sub, which uses a slider position so that 10 means accelerate 10 times while -10 means slow down 10 times.

The wait parameter determines how the rate change is performed. If 0, the call will perform the change and return immediately. But parts of the Sybase Aleri Streaming Platform that are waiting for an event (or "sleeping") might not discover that the clock rate has changed for up to maxSleep milliseconds.

If not 0, it will atomically stop the logical clock, change the rate, wait long enough for all the ongoing sleeps to discover the rate change and restart the clock at the new rate.

This call returns the previous state of the Sybase Aleri Streaming Platform clock. See the description above for `getClock()`.

```
input: auth_token: string
input: rate: double
input: wait: int
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.setClockTime(token, time)` changes the current logical time of the Sybase Aleri Streaming Platform. Time is expressed in seconds since the UNIX epoch. This call returns the previous state of the platform clock. See the above description for `getClock()`.

```
input: auth_token: string
input: time: double
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.setClockRateTime(token, rate, time, wait)` is a combination of setting time and rate as in the calls above. This call returns the previous state of the platform clock. See the above description for `getClock()`.

```
input: auth_token: string
input: rate: double
input: time: double
input: wait: int
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.setClockReal(token, wait)` restores the clock to use the real time. It would return an error if the clock is currently stopped. The wait argument is the same as for `setClockRate()`. This call returns the previous state of the Sybase Aleri Streaming Platform clock. See the description above in `getClock()`.

```
input: auth_token: string
input: wait: int
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.stopClock(token)` stops the logical clock in the Sybase Aleri Streaming Platform. The records will still be processed but the notion of time won't change and the timer events won't happen. While the clock is stopped, the time and rate may be changed but the clock may not be switched to real time. Stopping may be called multiple times, then resume must be called the same number of times to have the time flow resumed. Since the Sybase Aleri Streaming Platform may also stop and resume the clock internally, don't resume the clock more times than you've stopped it. This call returns the previous state of the clock. See the description above for `getClock()`.

```
input: auth_token: string
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.resumeClock(token)` resumes the flow of time in the Sybase Aleri Streaming Platform. This call returns the previous state of the clock. See the description above in `getClock()`.

```
input: auth_token: string
return: s(status): int
return: s(clockTime): double
return: s(clockRate): double
return: s(clockReal): int
return: s(stopDepth): int
return: s(maxSleep): int
return: s(errMsgs): array string
```

- `cimarron.login(username, password)` authenticates with the server during login, and on success returns an authentication token to be used in all other Command and Control calls.

```
input: username: string
input: password: string
return: s(status): int
return: s(hash): string (authentication token)
return: s(errMsgs): array string
```

- `cimarron.sendStreamsExit(auth_token)` posts an EXIT message to each source

stream. This causes an exit message to propagate through the entire dependency graph of source and streams. When all streams have processed the exit message, the Sybase Aleri Streaming Platform shuts down.

```
input: auth_token: string
return: void - cimarron should exit after all streams
        fully process queued data
```

- `cimarron.getSourceStreams(auth_token)` requests a list of the complete set of source streams. This includes the stream names and stream IDs.

```
input: auth_token: string
return: s(status): int
return: s(streamNum): int (number of source streams)
return: s(streamNames): array string (the stream names)
return: s(streamIds): array int (stream ids)
return: s(errMsgs): array string
```

- `cimarron.getDerivedStreams(auth_token)` requests a list of the complete set of derived streams. This includes the stream names and stream IDs.

```
input: auth_token: string
return: s(status): int
return: s(streamNum): int (number of Derived Streams)
return: s(streamNames): array string (the stream names)
return: s(streamIds): array int (stream ids)
return: s(errMsgs): array string
```

- `cimarron.getStreamDefinition(auth_token, stream-name)` requests detailed metadata for a particular stream. This includes the number of columns (fields), their names and types, and a Boolean array that indicates the key columns for the stream.

```
input: auth_token: string
input: stream_name: string
return: s(status): int
return: s(ColNum): int (number of fields)
return: s(ColNames): array string (the field names)
return: s(ColTypes): array int (field types)
return: s(Keys): array int (0 not key, 1 key)
return: s(errMsgs): array string
```

- `cimarron.getStreamHandleDefinition(auth_token, stream-handle)` requests detailed metadata for a particular stream. This includes the number of columns (fields), their names and types, and a Boolean array that indicates the key columns for the stream. Same as `getStreamDefinition()`, only uses the stream handle to find the stream.

```
input: auth_token: string
input: stream_handle: int
return: s(status): int
```

```
return: s(ColNum): int (number of fields)
return: s(ColNames): array string (the field names)
return: s(ColTypes): array int (field types)
return: s(Keys): array int (0 not key, 1 key)
return: s(errMsgs): array string
```

- `cimarron.getGateway(auth_token)` requests the name of the machine and the port that the Gateway Server interface is bound to and listens on.

```
input: auth_token: string
return: s(status): int
return: s(host): string (hostname)
return: s(port): int (port number)
return: s(errMsgs): array string
```

- `cimarron.isBigEndian(auth_token)` requests the server to identify its endian type.

```
input: auth_token: string
return: s(status): int
return: s(flag): int (1: big endian, 0: little endian)
return: s(errMsgs): array string
```

- `cimarron.isQuiesced(auth_token)` checks whether the server has finished processing all pending data and if there are any active input connections. The check for "no active input connections" is done first, and this call returns false (zero or busy) if there are any active input connections.

```
input: auth_token: string
return: s(status): int
return: s(flag): int (1: quiesced, 0: busy)
return: s(errMsgs): array string
```

- `cimarron.addStreamToClient(auth_token, client_handle, stream_name)`, given the handle of a Gateway client is in subscription mode, augments the list of streams that are subscribed by a client.

```
input: auth_token: string
input: client_handle: int
input: stream_name: string
return: int (status)
```

- `cimarron.addStreamHandleToClient(auth_token, client_handle, stream_handle)`, given the handle of a Gateway client is in subscription mode, augments the list of streams that are subscribed by a client. It's the same as `addStreamToClient()`, only it uses the stream handle to find the stream.

```
input: auth_token: string
input: client_handle: int
input: stream_handle: int
return: int (status)
```

- `cimarron.removeStreamFromClient(auth_token, client_handle, stream_name)`, given the handle of a Gateway client is in subscription mode, trims the list of streams that are subscribed by the client.

```
input: auth_token: string
input: client_handle: int
input: stream_name: string
return: int (status)
```

- `cimarron.removeStreamHandleFromClient(auth_token, client_handle, stream_handle)`, given the handle of a Gateway client is in subscription mode, trims the list of streams that are subscribed by the client. It's the same as `removeStreamFromClient()`, only it uses the stream handle to find the stream.

```
input: auth_token: string
input: client_handle: int
input: stream_handle: int
return: int (status)
```

- `cimarron.getDateSize(auth_token)` returns an integer that represents the native size in bytes of the server's datetime fields.

```
input: auth_token: string
return: s(status): int
return: s(flag): int (8: 64 bit server, 4: 32 bit server)
return: s(errMsgs): array string
```

- `cimarron.getAddressSize(auth_token)` returns an integer that represents the address/pointer size of the connected Sybase Aleri Streaming Platform. In C/C++ terminology, the value returned is: `sizeof(void *)`.

```
input: auth_token: string
return: s(status): int
return: s(flag): int (8: 64 bit server, 4: 32 bit server)
return: s(errMsgs): array string
```

- `cimarron.setParameter(auth_token, parameter_name, value)` sets the value of a parameter within the Sybase Aleri Streaming Platform. The flag value returned is 1 if the command is successful, 0 if the named parameter exists but was not set. An example would be if the value could not be converted to the type of the parameter or -1 the named parameter does not exist.

```
input: auth_token: string
input: parameter_name: string
input: value: string
return: s(status): int
return: s(flag): int (8: 64 bit server, 4: 32 bit server)
return: s(errMsgs): array string
```

- `cimarron.getStreamHandle(auth_token, stream_name)` gets the integer stream handle by stream name.

```
input: auth_token: string
input: stream_name: string
return: s(status): int
return: stream_handle: int
return: s(errMsgs): array string
```

- `cimarron.backup(auth_token)` creates a backup of all the Log Stores. The backup files are created with suffix `.bak`. The the file `dynamic.log` is backed up into the file `dynamic.bak`). The backup files are created as sparse files, with compacted contents.

```
input: auth_token: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.saveConfig(auth_token, file_name)` saves the current running AleriML configuration to this file on the server (the file must not exist before the function call).

```
input: auth_token: string
input: file_name: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.getConfig(auth_token)` returns the current running XML configuration.

```
input: auth_token: string
return: s(status): int
return: s(config): string
return: s(errMsgs): array string
```

- `cimarron.loadConfig(auth_token, file_name, options)` loads the new configuration from this file on the server. If the file contains errors, the error messages are printed in the Sybase Aleri Streaming Platform log, the error code returned, and the Sybase Aleri Streaming Platform will be left unchanged. The options affect the way the changes are applied. See the **sp_cli** man page for a description of these options.

```
input: auth_token: string
```

```
input: file_name: string
input: options: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.loadConfigInline(auth_token, XML_config, options)` loads the new configuration contained in the *XML_config* string. If the configuration contains errors, the error messages will be printed in the Sybase Aleri Streaming Platform log, the error code returned, and the Sybase Aleri Streaming Platform will be left unchanged. The options affect the way the changes are applied. See the description of these options in the **sp_cli** man page.

```
input: auth_token: string
input: XML_config: string
input: options: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.loadConfigInlineConv(auth_token, XML_config, options, XML_conv)` loads the new configuration contained in the *XML_config* string, and uses the model contained in *XML_conv* to convert the date in the base streams. If the configuration contains errors, the error messages are printed in the Sybase Aleri Streaming Platform log, the error code returned, and the Sybase Aleri Streaming Platform will be left unchanged. The options affect the way the changes are applied. See the description of options' format and meaning in the **sp_cli** man page. The `conv` option may not be used with this call.

```
input: auth_token: string
input: XML_config: string
input: options: string
input: XML_conv: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.logLevel(auth_token, level)` sets the logging level on the Sybase Aleri Streaming Platform.

```
input: auth_token: string
input: level: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.getMoneyPrecision(auth_token)` returns the decimal precision of the money data type used by the Sybase Aleri Streaming Platform.

```
input: auth_token: string
return: s(status): int
return: s(moneyPrecision) int
return: s(errMsgs): array string
```


- `cimarron.killClient(auth_token, handle)` closes the connection of the client identified by the connection handle. The handle can be found in the `conn_handle` field of the `Aleri_Clients` metadata stream.

```
input: auth_token: string
input: level: int
return: int status
```

- `cimarron.killClientByName(auth_token, tag_name)` closes the connections of all the clients that have the tag name set to equal to the specified name. The tag name can be found in the `conn_tag` field of the `Aleri_Clients` metadata stream.

```
input: auth_token: string
input: level: int
return: int status
```

- `cimarron.setTraceMode(auth_token on)` changes the trace mode, and disables it if on equals 0 or enables it if on !=0.

```
input: auth_token: string
input: level: int
return: int status
```

- `cimarron.getTraceMode(auth_token)` returns whether the Sybase Aleri Streaming Platform is currently in the trace mode as "tracemode".

```
input: auth_token: string
return: s(status): int
return: s(traceMode) int
return: s(errMsgs): array string
```

- `cimarron.runControl(auth_token, action, string_arg, int_arg)` controls the execution of the Sybase Aleri Streaming Platform while in trace mode. Action specifies a control action to perform. Each action might take a string argument, an integer argument, both or neither, depending on the particular action.

The supported actions are:

- `pause` - pause the platform execution
- `run` - resume the platform execution
- `step [s:stream_name]` - do a single step; if the stream argument is not empty, then step stream with this name, otherwise step any random stream.
- `wait` - wait until the Sybase Aleri Streaming Platform pauses (through `runControl` or on a breakpoint); this command cannot be canceled. Otherwise, to get the asynchronous notifications of the pause, subscribe to the `Aleri_RunControl` stream.

- `runwait` - a combination of `run` and `wait` is paused. Checks whether the Sybase Aleri Streaming Platform is paused; returns the status code 0 if it is paused, or `NOPAUSE` (15) if it is not.
- `setStepTimeout i:milliseconds` - set the timeout for automatic stepping from the integer argument; using the negative or zero value returns the timeout to the default (0.3s).
- `stepTrans s:stream_name i:n_steps` - step the stream either to the end of the transaction, or no more than `n_steps` steps, or until the stream get blocked on input or output for longer than the timeout (0.3s by default). `n_steps` less than 1 is processed as 1.)
- `stepQuiesceStream s:stream_name i:n_steps` - step the stream and its descendant streams until either the whole stream's input queue is processed, no more than `n_steps` steps, or until all the involved streams get blocked on input or output for longer than the timeout (0.3s by default). `n_steps` less than 1 is processed as 1.
- `stepQuiesceFromBase s:stream_name i:n_steps` - step all the non-source streams until the whole stream's input queue is processed, or no more than `n_steps` steps, or until all the involved streams get blocked on input or output for longer than the timeout (0.3s by default). `n_steps` less than 1 is processed as 1.

```
input: auth_token: string
input: action: string
input: string_arg: string
input: int_arg: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.dumpStream(auth_token, prefix, stream)` in trace mode writes the current contents of the stream(s) into a file. If the stream name is empty, writes all the streams. The file name for each stream is `<prefix> dump_<streamname>.xml`.

```
input: auth_token: string
input: prefix: string
input: stream: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.addBreakpoint (auth_token, stream, origin, expr, period)` in trace mode adds a breakpoint on the specified stream. Origin is the name of an input stream, or "*" to break on any input, or "" to break on output. If `expr` is not empty, it gives a conditional expression that must be satisfied to trigger the breakpoint. Period specifies how many times the condition needs to be detected before actually triggering a breakpoint. Returns the unique id of the newly created breakpoint.

```
input: auth_token: string
input: stream: string
input: origin: string
input: expr: string
input: period: int
return: s(status): int
return: s(id): int
return: s(errMsgs): array string
```

- `cimarron.delBreakpoint(auth_token, id)` in trace mode deletes the breakpoint with this id.

```
input: auth_token: string
input: id: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.enableBreakpoint(auth_token, id, period)` in trace mode enables the breakpoint and sets its period, if `period != 0`; otherwise it disables the breakpoint.

```
input: auth_token: string
input: id: int
input: on: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.examineDataStart (auth_token, kind, stream, object, startpos, expr)` in trace mode starts the examination of the data, as identified by kind, stream, and object name. Depending on the kind, stream and/or object name may be empty. The details are described in the `sp_cli(1)` man page. Startpos allows to skip a number of records. If `expr` is not empty, it specifies the filter expression, and only the rows for which the filter evaluates to true are returned. This call returns the format that will be returned but not the data itself.

The data may be a mix of rows from different streams or imitations of streams. The number of possible stream definitions is returned in `streamCount`. And the array `streamDefs` contains the following information for each stream definition:

<code>name</code>	name of the stream
<code>colNum</code>	number of columns
<code>colNames</code>	array of names for each column
<code>colTypes</code>	array of types for each column, represented as an integer code
<code>Keys</code>	array containing 1 for each key column and 0 for each non-key column

The returned information also contains the `bigEndian` flag for the architecture where the Sybase Aleri Streaming Platform is running. If this flag matches the one on the machine where the client program is running, data can be interpreted with the class `Row`, otherwise with `RowRBO`.

The cookie value is to be passed to the follow-up calls `cimarron.examineDataNext` that return the actual data. Currently, only one cookie may be active. If you call `cimarron.examineDataStart` twice, `cimarron.examineDataNext` will return the data only for the last cookie.

```
input: auth_token: string
input: kind: string
input: stream: string
input: object: string
input: startpos: int
input: object: string
return: s(status): int
```

```
return: s(bigEndian): int
return: s(cookie): int
return: s(streamCount): int
return: s(streamDefs): array struct(
    s(name): string
    s(colNum): int
    s(colNames): array string
    s(colTypes): array int
    s(Keys): array int
)
return: s(errMsgs): array string
```

- `cimarron.examineDataNext(auth_token, cookie, maxcnt)` in trace mode returns the data as initiated by `cimarron.examineDataStart`, identified by its cookie. Maxcnt is the maximum number of rows to return. If there are more rows available than returned, this call may be repeated to receive more data. Avoid using maxcnt that is too high, it may overflow the capabilities of XMLRPC. Maxcnt of 10000 is generally safe.

The returned data consists of:

- count - count of rows returned by this call
- eof - flag, non-0 if this is the last set of rows.
- srcIdx - an array containing for each row the index of its stream definition in the array returned by `cimarron.examineDataStart`.
- flags - an array containing for each row an integer with a bitmask or this row's flags. The currently supported flags are:
 - UPDATE_PAIR = 0x0001 - this is a first row of an update pair
 - CONTINUE_TRANS = 0x0002 - this is NOT the last row of a transaction
 - RETENTION_DEL = 0x0004 - this is a part of block of deletes generated by retention rows - an array of base64-encoded binary data for each row. The binary data can be interpreted using the class Row or RowRBO.

```
input: auth_token: string
input: cookie: int
input: maxcnt: int
return: s(status): int
return: s(count): int
return: s(eof): int
return: s(srcIdx): array int
return: s(flags): array int
return: s(rows): array base64
return: s(errMsgs): array string
```

- `cimarron.examineCount(auth_token, cookie)` in trace mode returns the information about the number of rows already returned by `examineDataNext` (pos) and the total number of rows available in this examination (count). This information can be used to get quickly last few rows from a big volume of data: do `examineDataStart(startpos=0)`, `examineCount()`, then `examineDataStart(startpos=count-N)`. The values are returned as double for extended precision, and large values may overflow the integer argument startpos.

```
input: auth_token: string
input: cookie: int
return: s(status): int
return: s(pos): double
return: s(count): double
return: s(errMsgs): array string
```

- `cimarron.getMaxThrottle(auth_token, stream)` returns the "throttle value" that limits the input queue size of a stream, as max. The queue size may grow up to twice the size of throttle value.

```
input: auth_token: string
input: stream: string
return: s(status): int
return: s(max): int
return: s(errMsgs): array string
```

- `cimarron.setMaxThrottle(auth_token, stream, throttle)` sets the "throttle value" that limits the input queue size of a stream. The queue size may grow up to twice the size of throttle value. If the stream name is empty, it sets the throttle value for all the streams.

```
input: auth_token: string
input: stream: string
input: throttle: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.isPaused(auth_token)`, in trace mode, returns whether the Sybase Aleri Streaming Platform is currently paused. It's equivalent to the same action of `cimarron.runControl`, only the pause flag is returned as a separate value instead of a status code.

```
input: auth_token: string
return: s(status): int
return: s(isPaused): int
return: s(errMsgs): array string
```

- `cimarron.evaluateExpr(auth_token, stream, object, expression)`, in trace mode, evaluates a debugging SPLASH expression `expr` in the context of a stream. The argument `object` is currently reserved and must always be an empty string. The result string is currently also a placeholder and is always returned empty.

```
input: auth_token: string
input: stream: string
input: object: string
input: expr: string
return: s(status): int
return: s(result): string
return: s(errMsgs): array string
```

- `cimarron.setHistorySize(auth_token, stream, size)`, in trace mode, sets the size of history kept for the stream. If the stream name is empty, sets the history size for all the streams.

```
input: auth_token: string
input: stream: string
input: size: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.getHistorySize(auth_token, stream)`, in trace mode, returns the size of history kept for the stream.

```
input: auth_token: string
input: stream: string
return: s(status): int
return: s(size): int
return: s(errMsgs): array string
```

- `cimarron.immediateExit(auth_token)` requests the Sybase Aleri Streaming Platform to exit immediately, bypassing the normal shutdown procedure. It's a very abrupt way to stop the Sybase Aleri Streaming Platform, almost equivalent to crashing it. Use it only as last resort, in situations such as when a client stops receiving data and the Sybase Aleri Streaming Platform can't flush its output queues during a normal stop. But killing the client's connection is a better idea even in this case.

```
input: auth_token: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.wipeoutBaseStream(auth_token, stream)` deletes all the contents of a source stream.

```
input: auth_token: string
input: stream: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.startConnector(auth_token, name)` starts the connector identified by name, or all the connectors in a group identified by name.

```
input: auth_token: string
input: name: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.stopConnector(auth_token, name, immediate)` stops the connector identified by name, or all the connectors in a group identified by name. Does not wait for connectors to be actually stopped. The `immediate` flag affects how the output connectors are stopped. If 0, new data stops being queued for the connector, then the Sybase Aleri Streaming Platform waits for the connector to drain its queue normally, and only then stops it. If not 0, the connector's queue gets discarded immediately.

```
input: auth_token: string
input: name: string
input: immediate: int
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.waitConnector(auth_token, name)` waits to exit for the connector identified by name, or all the connectors in a group identified by name.

```
input: auth_token: string
input: name: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.waitConnectorInitial(auth_token, name)` waits for the connector identified by name, or all the connectors in a group identified by name, to complete the initial loading.

```
input: auth_token: string
input: name: string
return: s(status): int
return: s(errMsgs): array string
```

- `cimarron.startUpConnectors(auth_token)` starts the connectors as during the normal start-up sequence. Returns after all the connectors in the sequence have completed the initial loading.

```
input: auth_token: string
return: s(status): int
return: s(errMsgs): array string
```

Appendix F. Using Encryption with Java Client Applications

In order to develop a Java client application that runs against an instance of the Sybase Aleri Streaming Platform in encrypted mode (see the **sp_server** manpage for more information on the `-e` option), you must ensure that the certificate/key information is generated and installed correctly in the environment.

Follow the steps in this section very carefully before attempting to set up the certificate and key information.

1. Create a directory that will be used to store the certificate/key files.

Eventually, this directory will be populated with the certificate/key files that are generated in the following steps. The directory must be specified as an argument to the `-e` option when starting up the **sp_server** in encrypted mode.

2. Generate the certificate and key files using the **genkeys** script.

The Sybase Aleri Streaming Platform ships with a shell script, **bin/genkeys**, that can be used to create the required key and certificate files. Production users must use this script to generate files whose names have the form `server.*` and copy them into the desired location (the directory referenced in step 1).

One of the files that is generated by the **genkeys** script is a file called `server.crt.der`. This file must be imported into the Java cacerts keystore file to get the Java clients to connect to the Sybase Aleri Streaming Platform using HTTPS (for the XMLRPC Command and Control Process connections) and SSL (for secure socket connections to the Gateway I/O Process).

The **genkeys** application takes two command line parameters:

- The number of days before the certificate will expire. This is an integer that is set to a value that is appropriate for a particular environment.
- The “Common Name” value that will be assigned to the *CN* fields within the certificate file `server.crt` that is generated by the **genkeys** script. If this parameter is not specified, the script uses the value returned by the **hostname** operating system command.

Note:

For Java based clients to work with encryption enabled, the *CN*(Common Name) field value of the `server.crt` file must be set to the hostname of the machine running the Sybase Aleri Streaming Platform. In addition, when attempting to connect to the Sybase Aleri Streaming Platform, the Java client (**sp_viewer**, Adapter, and so forth) must use the exact text representation of the hostname as specified in the *CN* field of the `server.crt` file. If the two do not match exactly, the Java client fails to connect to the Sybase Aleri Streaming Platform during the certificate validation process.

For example, if the **genkey** script generates a `server.crt` file in which the *CN* fields are set to the value `ganges.aleri.com`, the Java client must use the identical string, the hostname `ganges` by itself might not work.

In the above scenario, if the Java **sp_viewer** client attempts to connect to the server using a host-name value of “ganges”, while the *CN* field of the certificate is set to the value `ganges.aleri.com`, the Java client generates the following exception:


```
Exception: java.io.IOException: HTTPS hostname wrong: should be <ganges>
java.io.IOException: HTTPS hostname wrong: should be <ganges>
at sun.net.www.protocol.https.HttpsClient.b(DashoA12275)
at sun.net.www.protocol.https.HttpsClient.afterConnect(DashoA12275)
at sun.net.www.protocol.https.AbstractDelegateHttpsURLConnection.connect(DashoA12275)
at sun.net.www.protocol.http.HttpURLConnection.getOutputStream(HttpURLConnection.java:569)
at sun.net.www.protocol.https.HttpsURLConnectionImpl.getOutputStream(DashoA12275)
at org.apache.xmlrpc.DefaultXmlRpcTransport.sendXmlRpc(DefaultXmlRpcTransport.java:83)
at org.apache.xmlrpc.XmlRpcClientWorker.execute(XmlRpcClientWorker.java:71)
at org.apache.xmlrpc.XmlRpcClient.execute(XmlRpcClient.java:193)
at com.aleri.asap.tools.SPXmlRpc.login(Unknown Source)
at com.aleri.asap.tools.SPViewer.initialize(Unknown Source)
at com.aleri.asap.tools.SPViewer$2.construct(Unknown Source)
at com.aleri.asap.tools.SwingWorker$2.run(Unknown Source)
at java.lang.Thread.run(Thread.java:534)
Could not login to the Command and Control process
on <host:port> = <ganges:22000>, for user=cimarron
```

Although it is somewhat unclear, this exception indicates that a certificate was found in the Java trusted keystore, but the *CN* (Common Name) field of the certificate did not exactly match the host-name value specified by the Java client.

3. After running the **genkeys** script, make sure to copy the generated certificate/key files into a directory dedicated to storing the certificate/key information.

The **genkeys** script can be executed from within the directory created earlier to avoid having to copy the certificate files manually.

Here are the important certificate/key files generated by the **genkeys** script:

- `server.crt`
- `server.key`
- `server.crt.der`

These must be copied into the directory that will be specified using the `-e` option when starting the **sp** server in encrypted mode.

Technically, the `server.crt.der` file does not need to exist within the directory where the keys are stored. However, its contents must be imported into the Java `cacerts` file. It is easier to manage if all the generated certificate/key files are kept together in one place.

4. Import the certificate/key information located in the `server.crt.der` file, which was generated by the **genkeys** script, into the Java JRE environment that will be used to run the Java client applications, such as **sp_viewer**, the Java Adapter, and so forth.

If you want to run the Java client application (**sp_viewer** or **Adapter**, for example) on Windows, first copy the `server.crt.der` file that was generated on UNIX through the **genkeys** application onto your Windows machine. Next, use the **keytool** application of the JRE on your Windows machine to import the `server.crt.der` file.

To import `server.crt.der`, use the Java **keytool** program, and keep the target of the import as

the Java cacerts file. The syntax of the **keytool** import command depends on the operating system.

On UNIX, the **keytool**'s import command line is:

```
YourJavaHomeBinDirectory/keytool -keystore
YourJavaHome/lib/security/cacerts -alias
AnyNameForTheCertificate -import -file server.crt.der
```

On Windows, the **keytool**'s import command line is:

```
YourJavaHomeBinDirectory\keytool -import -alias
AnyNameForTheCertificate -file server.crt.der -keystore
YourJavaHome\lib\security\cacerts
```

Enter the password for the cacerts keystore (the default is changeit). Accept the trust certificate.

Note

You must be sure that the server.crt.der information is imported into the cacerts file of the JRE(s) used to run the Java client application(s). If there are several JDKs on your machine, and you plan to run the Java clients on each of them, make sure that the server.crt.der file is imported into each one. Additionally, the path to the Java **keytool** application should reflect the JRE version where the server.crt.der file is being imported. You must also make sure that you must have permission to change the cacerts file so you can run **chmod u+w** on the cacerts file if necessary.

If the import of the server.crt.der file is not performed when the Java client is run, the following exception appears:

```
Exception: javax.net.ssl.SSLHandshakeException:
    sun.security.validator.ValidatorException: No trusted
    certificate found
javax.net.ssl.SSLHandshakeException:
    sun.security.validator.ValidatorException: No trusted
    certificate found
at com.sun.net.ssl.internal.ssl.BaseSSLSocketImpl.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SunJSSE_az.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SunJSSE_az.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SunJSSE_ax.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.a(DashoA12275)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.j(DashoA12275)
at com.sun.net.ssl.internal.
    ssl.SSLSocketImpl.startHandshake(DashoA12275)
at sun.net.www.protocol.https.HttpsClient.
    afterConnect(DashoA12275)
at sun.net.www.protocol.https.
    AbstractDelegateHttpsURLConnection.connect(DashoA12275)
at sun.net.www.protocol.http.
    HttpURLConnection.getOutputStream(HttpURLConnection.java:569)
at sun.net.www.protocol.https.
    HttpsURLConnectionImpl.getOutputStream(DashoA12275)
at org.apache.xmlrpc.
```

```
DefaultXmlRpcTransport.sendXmlRpc(DefaultXmlRpcTransport.java:83)
at org.apache.xmlrpc.
  XmlRpcClientWorker.execute(XmlRpcClientWorker.java:71)
at org.apache.xmlrpc.
  XmlRpcClient.execute(XmlRpcClient.java:193)
at com.aleri.asap.tools.SPXmlRpc.login(Unknown Source)
at com.aleri.asap.tools.SPViewer.initialize(Unknown Source)
at com.aleri.asap.tools.SPViewer$2.construct(Unknown Source)
at com.aleri.asap.tools.SwingWorker$2.run(Unknown Source)
at java.lang.Thread.run(Thread.java:534)
Caused by: sun.security.validator.
  ValidatorException: No trusted certificate found
    at sun.security.validator.
      SimpleValidator.buildTrustedChain(SimpleValidator.java:304)
at sun.security.validator.
  SimpleValidator.engineValidate(SimpleValidator.java:107)
at sun.security.validator.
  Validator.validate(Validator.java:202)
at com.sun.net.ssl.internal.
  ssl.X509TrustManagerImpl.checkServerTrusted(DashoA12275)
at com.sun.net.ssl.internal.
  ssl.JsseX509TrustManager.checkServerTrusted(DashoA12275)
... 18 more
Could not login to the Command and Control process
on <host:port> = <ganges:22000>, for user=cimarron
Error = 0
```

F.1. Ready To Run in Encrypted Mode

If you have successfully performed all the steps, the following are now true:

1. The key directory is set up.
2. The `server.crt`, `server.key`, and `server.crt.der` files are present in this directory.
3. The contents of the `server.crt.der` file have been successfully imported into the appropriate Java cacerts file.

Now you can start up the Sybase Aleri Streaming Platform using the `-e` option to specify the name of the directory where the certificate/key files were stored.

Once the server has been started up in encrypted mode, the Sybase Aleri Streaming Platform Command and Control process will be accessed from a Java client application using XMLRPC over HTTPS. Refer to the *Utilities Guide* for more information. The client connections to the Sybase Aleri Streaming Platform Gateway I/O process will be made using SSL socket connections.

If the **sp** server is to be created using Java, make sure that the Java application specifies the exact same hostname value stored in the *CN* fields of the `server.crt` file. If this is not the case, the Java client will not connect to the **sp** server, and the message `java.io.IOException: HTTPS hostname wrong: should be <hostname>` exception will appear.

To use SSL in JDBC, add `?ssl` to the connection URL, for example:

```
jdbc:postgresql://hostname:22200/database?ssl
```