SYBASE®
An SAP® Company

Java SDK Guide

Sybase Event Stream Processor 5.0

DOCUMENT ID: DC01618-01-0500-02
LAST REVISED: December 2011
Copyright © 2011 by 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 the Sybase trademarks page 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.
SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.
Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.
Unicode and the Unicode Logo are registered trademarks of Unicode, Inc.
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.

Contents

Migration from Aleri Streaming Platform

The SDK interface provided by Sybase® Event Stream Processor (ESP) differs from the SDK interface provided in Aleri Streaming Platform (ASP). In Event Stream Processor, the SDK has been modified for improved flexibility and performance, and to accommodate projects running in a clustered environment.

Clusters and Projects

Because projects now run in a cluster, they are no longer accessed using the command and control host and port. A project has a unique identity denoted by its URI which typically consists of the cluster information, workspace name, and project name. The SDK takes care of resolving the URI to the physical address internally. The project object in ESP loosely corresponds to the platform object in ASP. There is no analogue of an ESP Server in the Pub/Sub API.

Note: There are methods to connect to a standalone project but these should not be used as they will be removed in a future release.

The ESP SDK includes new functionality to configure and monitor the cluster. There is no counterpart for these in the ASP Pub/Sub API.

Access Modes

In the ASP Pub/Sub, the Platform and Publisher objects were accessed using synchronous method calls. The Subscriber object required callback handlers. In ESP, this has changed. All entities—that is server, project, publisher, and subscriber—can be accessed using either DIRECT method calls or CALLBACK handlers. In addition, ESP introduces a third method called SELECTION access.

DIRECT access is similar to the way old Platform and old Publisher objects were called in ASP. Each call blocks until the task completes or results in an error. In ESP, you can use this mode for Subscribers too.

In CALLBACK, users register handler functions and the SDK calls the functions when anything of interest happens. This was the only way to work with subscribers in ASP. In ESP, you can optionally use this method for other entities too.

The SELECT access mode lets you register multiple entities with a selector and have a single thread wait for an event on any of those entities. Functionally, this is similar to the select/poll mechanism of monitoring multiple file descriptors in a single thread.

Automatic Reconnection and Monitoring

In ASP, the Pub/Sub API supported automatic reconnection to a peer when working in hot-active mode. ESP supports automatic reconnection but adds some additional functionality when working in CALLBACK or SELECT access modes. Additional functionality includes

checking if a cluster or project has gone down and optionally monitoring the backend for restarts.

Publishing

In DIRECT access mode, you can now optionally have the SDK spin a background thread when publishing to lead to better throughput. When using ASP, tasks such as these had to be done by the Pub/Sub user.

In ASP, a message was formatted using temporary storage (vectors) which needed to be filled in before calling the Pub/Sub API to create the buffer. In ESP, this is avoided by writing directly to a buffer. To create a message in the ESP SDK, users will indicate the start of a block or row, then populate it in sequence. The fields must be filled in the same order as they appear in the schema.

Subscribing

In ASP, the data from a message was available as a collection of objects. In the ESP SDK, that step is skipped. Methods are provided to read the buffer directly as native data types or helper objects (Money, BigDatetime, Binary). The data fields can be accessed in random order.

Entity Lifecycles and Access Modes

In the Sybase® Event Stream Processor Java SDK, all entities share a common lifecycle and set of access modes.

User interaction in the Event Stream Processor (ESP) SDK is handled through entities the SDK exposes. The main entities are Server, Project, Publisher, and Subscriber. These entities correspond to the functional areas of the SDK. For example, the Server object represents a running instance of a cluster, the Project corresponds to a single project deployed to the cluster, the Publisher object deals with publishing data to a running project, and so on.

On initial retrieval an entity is considered to be open. When an entity is open, you can retrieve certain static information about it. To accomplish its assigned tasks, an entity has connect to the corresponding component in the cluster. A server connects to a running instance of a cluster, and Project, Publisher and Subscriber all connect to running instances of a project in a cluster.

In the connected state, an entity can interact with the cluster components. Once an entity is disconnected, it can no longer interact with the cluster but is still an active object in the SDK, and can be reconnected to the cluster. Once an entity is closed, it is no longer available for interaction and is reclaimed by the SDK. To reuse an entity that has closed, retrieve a fresh copy of the entity.

For example, you can retrieve a Project object and connect it to a project in the cluster. If the back-end project dies, the SDK Project receives a disconnected event. You can attempt to reconnect manually, or, if you are using callback mode and your configuration supports it, the SDK tries to reconnect automatically. Upon successful reconnection, the SDK generates a connected event. If you actively close the entity, it disconnects from the back-end project and the SDK reclaims the Project object. To reconnect, you first need to retrieve a new Project object.

The SDK provides a flexible way to structure access to the various entities exposed by the API. The modes available to access entities are direct, callback, and select. The default access mode—in force whenever an entity is retrieved—is direct. In this mode, all operations on an entity are synchronous. Once the call returns, either the operation is considered complete or an error has occurred.

In callback access, register a handler function with the entity. Most calls to the entity return immediately. Completion of the request is indicated by the generation of the corresponding event. The SDK has two internal threads to implement the callback mechanism. The update thread monitors all entities currently registered for callbacks for applicable updates. If an update is found, an appropriate event is created and queued to the dispatch thread. The dispatch thread calls the registered handlers for the user code to process them.

The following sample illustrates accessing a project in callback mode. If you are working in callback mode and want to receive the callback events, register your callback handlers before you call connect on the entity you are interested in.

```
    ProjectOptions opts = new
ProjectOptions.Builder().setAccessMode(AccessMode.CALLBACK).create(
);

   Project project = SDK.getInstance().getProject(projectUri, creds,
opts);



    project.setCallback(EnumSet.allOf(ProjectEvent.Type.class), new
ProjectHandler("Handler"));



    project.connect(60000);



    //

    // Wait or block.
 Rest of the project lifecycle is handled in the project callback
handler

    //




    // Project handler class

    public class ProjectHandler implements Callback

    {

        String m_name;



        ProjectHandler(String name) {

            m_name = name;

        }



        public String getName() {

            return m_name;
```

```
        }


        public void processEvent(ProjectEvent pevent)

        {

            Project p = pevent.getProject();

            try {

                switch ( pevent.getType() ) {

                // Project has connected - can retrieve streams,
deployment etc.

                case CONNECTED:

                    String[] streams =
pevent.getProject().getModelledStreamNames();

                    break;

               // Project disconnected - only call possible connect
again

                case DISCONNECTED:

                    break;

              // Project closed - this object should not be accessed
anymore by user code

                case CLOSED:

                    break;

                case STALE:

                case UPTODATE:

                    break;

                case ERROR:

                    break;

                }

            } catch (IOException e) {

            }

        }
```

```
    }
```

The select access mode lets you multiplex various entities in a single user thread—somewhat similar to the select and poll mechanisms available on many systems—to monitor file descriptors. Register an entity using a Selector together with the events to monitor for. Then, call **Selector.select()**, which blocks until a monitored update occurs in the background. The function returns a list of SdkEvent objects. First determine the category (server, project, publisher, subscriber) of the event, then handle the appropriate event type. In this mode, the SDK uses a single background update thread to monitor for updates. If detected, the appropriate event is created and pushed to the Selector. The event is then handled in your own thread.

This example shows multiplexing of different entities.

```
        Uri cUri = new Uri.Builder(REAL_CLUSTER_URI).create();



        Selector selector = SDK.getInstance().getDefaultSelector();



        ServerOptions srvopts = new
ServerOptions.Builder().setAccessMode(AccessMode.SELECT).create();

        Server server = SDK.getInstance().getServer(cUri, creds,
srvopts);



        ProjectOptions prjopts = new
ProjectOptions.Builder().setAccessMode(AccessMode.SELECT).create();

       Project project = null; //SDK.getInstance().getProject(cUri,
creds, prjopts);



        SubscriberOptions subopts = new
SubscriberOptions.Builder().setAccessMode(AccessMode.SELECT).create
();

        Subscriber subscriber = null; //
SDK.getInstance().getProject(cUri, creds, prjopts);



        PublisherOptions pubopts = new
PublisherOptions.Builder().setAccessMode(AccessMode.SELECT).create(
);

        Publisher publisher = null; //
```

```
SDK.getInstance().getProject(cUri, creds, prjopts);


        server.connect();


        server.selectWith(selector,
EnumSet.allOf(ServerEvent.Type.class));


        // Your logic to exit the loop goes here ...
        while (true) {


            List<sdkevent> events = selector.select();


            for (SdkEvent event : events) {
                switch (event.getCategory()) {
                // Server events
                case SERVER:
                    ServerEvent srvevent = (ServerEvent) event;
                    switch (srvevent.getType()) {


                    // Server has connected - can now perform
operations, such as adding removing
                    // applications.
                    case CONNECTED:
                    case MANAGER_LIST_CHANGE:
                        Manager[] managers =
srvevent.getServer().getManagers();
                        for (Manager m : managers)
                            System.out.println("Manager:" + m);
                        break;
                    case CONTROLLER_LIST_CHANGE:
```

```
                        Controller[] controllers =
srvevent.getServer().getControllers();

                        for (Controller cn : controllers)

                            System.out.println("Controller:" + cn);

                        break;

                    case WORKSPACE_LIST_CHANGE:

                        break;


                   // This indicates that the Server has updated its
state with the latest running application

                    // information.
 Project objects can now be retrieved

                    case APPLICATION_LIST_CHANGE:

                    case DISCONNECTED:

                    case CLOSED:

                    case ERROR:

                        break;

                    }

                    break;


                // Project events

                case ESP_PROJECT:

                    ProjectEvent prjevent = (ProjectEvent) event;

                    switch (prjevent.getType()) {

                    case CONNECTED:

                    case DISCONNECTED:

                    case CLOSED:

                    case ERROR:

                    case WARNING:
```

```
                        break;
                    }
                    break;


                // Publisher events
                case PUBLISHER:
                   PublisherEvent pubevent = (PublisherEvent) event;
                    switch (pubevent.getType()) {
                    case CONNECTED:
                    // The publisher is read.
This event is to be used to publish data in callback mode
                    case READY:
                    case DISCONNECTED:
                    case CLOSED:
                        break;
                    }
                    break;


                // Subscriber events
                case SUBSCRIBER:
                  SubscriberEvent subevent = (SubscriberEvent) event;
                    switch (subevent.getType()) {
                    case CONNECTED:
                    case SUBSCRIBED:
                    case SYNC_START:
                    // There is data.
This event is to be used to retrieve the subscribed data.
                    case DATA:
                    case SYNC_END:
```

```
                    case DISCONNECTED:
                    case CLOSED:
                    case UNSUBSCRIBED:
                    case DATA_INVALID:
                    case ERROR:
                    case STREAM_EXIT:
                    case DATA_LOST:
                        break;
                    }
                    break;
                }
            }
        }
```

Publishing

The SDK provides a number of different options to publish data to a project.

The steps involved in publishing data are.

1. Retrieve a Publisher for the project to which you need to publish. You can create a Publisher directly or from a previously retrieved and connected Project object.
2. Create a MessageWriter for the stream to publish to. You can create multiple MessageWriters from a single Publisher.
3. Create a RelativeRowWriter method.
4. Format the data buffer to publish using RelativeRowWriter methods.
5. Publish the data.

While Publisher is thread safe, MessageWriter and RelativeRowWriter are not. Therefore, ensure that you synchronize access to the latter two.

The SDK supports automatic publisher switchover in high availability (HA) configurations, except when publishing asynchronously. Switching over automatically in this instance risks dropping or duplicating records because the SDK does not know which records have been published.

The SDK provides a number of options to tune the behaviour of a Publisher. Specify these options using the PublisherOptions object when the Publisher is created. Once created, options cannot be changed. Like all other entities in the SDK, publishing supports the direct, callback, and select access modes.

In addition to access modes, the SDK supports internal buffering. When publishing is buffered, the data is first written to an internal queue. This is picked up by a publishing thread and then written to the platform. Note that buffering is possible only in direct access mode. Direct access mode together with buffered publishing potentially provides the best throughput.

Two other settings influence publishing: batching mode and sync mode. Batching controls how data rows are written to the socket. They can be written individually, or grouped together in envelope or transaction batches. Envelopes group individual rows together to send to the platform and are read together from the socket by the platform. This improves network throughput. Transaction batches, like envelopes, are also written and read in groups. However, with transaction batches, the platform only processes the group if all the rows in the batch are processed successfully. If one fails, the whole batch is rolled back.

Note: When using shine-through to preserve previous values for data that are null in an update record, publish rows individually or in envelopes, rather than in transaction batches.

Sync mode settings control the publishing handshake between the SDK and the platform. By default, the SDK keeps sending data to the platform without waiting for an acknowledgement.

But if sync mode is set to true, the SDK waits for an acknowledgement from the platform each time it sends data before sending more. Issue an explicit **Publisher.commit()** call when publishing in asynchronous mode before the client application exits or closes the Publisher. This is to guarantee that all data written is read by the platform.

In general terms, the return code from a Publish call indicates whether or not the row was successfully transmitted. Any error that occurs during processing on the platform (such as a duplicate insert) will not get returned. The precise meaning of the return code from a Publish call depends on the access mode and the choice of synchronous or asynchronous transmission.

When using callback or select access mode, the return only indicates whether or not the SDK was able to queue the data. The indication of whether or not the data was actually written to the socket will be returned by the appropriate event. The callback and select access modes do not currently support synchronous publishing.

When using direct access mode, the type of transmission used determines what the return from the Publish call indicates. If publishing in asynchronous mode, the return only indicates that the SDK has written the data to the socket. If publishing in synchronous mode, the return from the Publish call indicates the response code the platform sent.

In no case will errors that occur during processing on the platform (such as a duplicate insert) be returned by a Publish call.

There are certain considerations to keep in mind when using callback or select mode publishing. These modes are driven by the PublisherEvent.READY event, which indicates that the publisher is ready to accept more data. In response, users can publish data or issue a commit, but only one such action is permitted in response to a single PublisherEvent.READY event.

Like all entities, if you intend to work in callback mode with a Publisher and want to get notified, register the callback handler before the event is triggered. For example:

```
PublisherOptions opts = new
PublisherOptions.Builder().setAccessMode(AccessMode.CALLBACK).creat
e();
        Publisher publisher = project.createPublisher(opts);
        PublisherHandler handler = new PublisherHandler();

publisher.setCallback(EnumSet.allOf(PublisherEvent.Type.class),
handler);
        publisher.connect();
```

Below are some code snippets that illustrate different ways of publishing data.

This example demonstrates publishing in direct access mode with transaction blocks.

```
    // The Project must be connected first

    project.connect(60000);
```

```
Publisher publisher = project.createPublisher();
publisher.connect();

Stream stream = project.getStream("Stream");
MessageWriter mw = publisher.getMessageWriter(s);
RelativeRowWriter writer = mw.getRelativeRowWriter();
// It is more efficient to cache this
DataType[] types = stream.getEffectiveSchema().getColumnTypes();

// Your logic to loop over data to publish
while (true) {

    // Logic to determine if to start a transaction block
    if ( ...)
        mw.startTransaction(0);

    // Loop over rows in a single block
    while (true) {

        // Loop over columns in a row
        for (i = ...) {

            writer.startRow();
            writer.setOperation(Operation.INSERT);

            switch (types[i]) {
            case DATE:
                writer.setDate(datevalue);
```

```
                    break;
                case DOUBLE:
                    writer.setDouble(doublevalue);
                    break;
                case INTEGER:
                    writer.setInteger(intvalue);
                    break;
                case LONG:
                    writer.setLong(longvalue);
                    break;
                case MONEY:
                    break;
                case STRING:
                    writer.setString(stringvalue);
                    break;
                case TIMESTAMP:
                    writer.setTimestamp(tsvalue);
                    break;
                //
                // Other data types
                //
                }
            }
            writer.endRow();
        }

        // Logic to determine if to end block
        if ( ...)
```

```
            mw.endBlock();

    }


    publisher.commit();
```

This example demonstrates publishing in callback access mode. Notice that the access mode is set before the publisher is connected; this is a requirement for both callback and select access modes.

```
    p.connect(60000);


    PublisherOptions opts = new PublisherOptions.Builder()

        .setAccessMode(AccessMode.CALLBACK)

        .create();


    Publisher pub = p.createPublisher(opts);


    PublisherHandler handler = new PublisherHandler();

    pub.setCallback(EnumSet.allOf(PublisherEvent.Type.class),
handler);

    pub.connect();


    // Block/wait. Publishing happens in the callback handler

    // ....


    //

    // Publisher callback handler

    //

    public class PublisherHandler implements Callback

    {

        Stream m_stream;
```

```
        MessageWriter m_mwriter;
        RelativeRowWriter m_rowwriter;

        public PublisherHandler() throws IOException {
        }

        public String getName() {
            return "PublishHandler";
        }

        public void processEvent(PublisherEvent event)
        {
            switch (event.getType()) {
            case CONNECTED:
                // It is advisable to create and cache these
                try {
                    m_stream =
event.getPublisher().getProject.getStream("Stream");
                    m_mwriter =
event.getPublisher().getMessageWriter(mstr);
                    m_rowwriter = mwriter.getRelativeRowWriter();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                break;

            case READY:
                // Publishing code goes here.
              // NOTE: Only a single publish or a commit call can be
made in one READY callback
```

```
                break;

            case ERROR:
            case DISCONNECTED:
            case CLOSED:
                break;
            }
        }
    }
```

Subscribing

The SDK provides several options for subscribing to data in a project.

Subscribing to data using the SDK involves:

1. Create a Subscriber object. Create the object directly or retrieve it from a Project object.
2. Connect the Subscriber object.
3. Subscribe to the desired streams.
4. In direct access mode, retrieve events using the Subscriber.getNextEvent() object. In callback and select access modes, the SDK generates events and passes them back to user code.
5. For data events, retrieve MessageReader. This encapsulates a single message from the platform. It may consist of a single data row, or a transaction or envelope block with multiple data rows.
6. Retrieve one or more RowReaders. Use the methods in RowReader to read in individual fields.

This example demonstrates subscribing to a stream in direct access mode with default options.

```
p.connect(60000);
subscriber = p.createSubscriber();
String strName = "WIN1";

subscriber.subscribeStream("WIN1");
subscriber.connect();

// Various data type we will be reading
BigDatetime bigdatetime = null;
Money m = null;
byte[] binary = null;

// Logic to exit loop goes here
while (true) {
    SubscriberEvent event = subscriber.getNextEvent();
```

```
            switch (event.getType()) {
            case SYNC_START:
                break;
            case SYNC_END:
                break;

            // There is data to read
            case DATA:
                while ( reader.hasNextRow() ) {
                    RowReader rr = reader.nextRowReader();
                 for (int j = 0; j < rr.getSchema().getColumnCount();
++j) {
                       if ( rr.isNull(j))
                           continue;

                     // This is legal but it is better to cache the
data types array

                     switch ( rr.getSchema().getColumnTypes()[j]) {
                       case INTEGER:
                           rr.getInteger(j);
                           break;
                       case LONG:
                           rr.getLong(j);
                           break;
                       case STRING:
                           rr.getString(j);
                           break;
                       case TIMESTAMP:
```

```
                        rr.getTimestamp(j));
                        break;
                    case MONEY01:
                        m = rr.getMoney(j);
                        break;

                    // ...
                    // process other data types
                    // ...

                    }
                }
            }
            break;
        }

    }
    subscriber.disconnect();
```

Failover Handling

The SDK supports either fully transparent or automatic failover in a number of situations.

- **Cluster Failovers –** the URIs used to connect to a back-end component can include a list of cluster manager specifications. The SDK maintains connections to these transparently. If any one manager in the cluster goes down, the SDK tries to reconnect to another instance. If connections to all known instances fail, the SDK returns an error. If working in callback or select access modes, you can configure the SDK with an additional level of tolerance for loss of connectivity. In this case, the SDK does not disconnect a Server instance even if all known manager instances are down. Instead, it generates a ServerEvent.STALE event. If it manages to reconnect after a (configurable) number of attempts, it generates a ServerEvent.UPTODATE event. Otherwise, it disconnects and generates a ServerEvent.DISCONNECTED event.
- **Project Failovers –** an Event Stream Processor cluster lets you deploy projects with failover. Based on the configuration settings, a cluster restarts a project if it detects that it has exited (however, projects are not restarted if they are explicitly closed by the user). To support this, you can have Project instances monitor the cluster for project restarts and then reconnect. This works only in callback or select modes. When the SDK detects that a project has gone down, it generates a ProjectEvent.STALE event. If it is able to reconnect, it generates a ProjectEvent.UPTODATE event, otherwise it generates a ProjectEvent.DISCONNECTED event.
- **Active-Active Deployments –** you can deploy a project in active-active mode. In this mode, a cluster starts two instances of the project, a primary instance and a secondary instance. Any data published to the primary is automatically mirrored to the secondary instance. The SDK supports active-active deployments. When connected to an active-active deployment, if the currently connected instance goes down, the Project tries to reconnect to the alternate instance. Unlike failovers, this happens transparently. Therefore, if the reconnection is successful, there is no indication generated to the user. In addition to the Project, there is support for this mode when publishing and subscribing. If subscribed to a project in an active-active deployment, the SDK does not disconnect the subscription if the instance goes down. Instead, it generates a SubscriberEvent.DATA_LOST event. It then tries to reconnect to the peer instance. If it is able to reconnect, the SDK resubscribes to the same streams. Subscription clients then receive a SubscriberEvent.SYNC_START event, followed by the data events, and finally a SubscriberEvent.SYNC_END event. Clients can use this sequence to maintain consistency with their view of the data if needed. Reconnection during publishing is also supported, but only if publishing in synchronous mode. It is not possible for the SDK to guarantee data consistency otherwise. Reconnection during publishing happens transparently; there are no external user events generated.

API Reference

Detailed information on methods, functions, and other programming building blocks is provided in the API documentation download.

Download and install the *Java API documentation* to your local machine.

When the download is complete:

1. Extract the files to the desired location on your local machine.
2. Browse to the location where you extracted the files.
3. Launch `index.html` to view the API documentation.

Index