SYBASE®
An SAP® Company

.NET SDK Guide

Sybase Event Stream Processor 5.0

DOCUMENT ID: DC01619-01-0500-01
LAST REVISED: September 2011

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

Contents

Entity Lifecycles and Access Modes

The Sybase® Event Stream Processor .NET SDK offers the same functionality and uses the same concepts as the C SDK. All entities exposed by the SDK have a common lifecycle.

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 to connect to the corresponding component in the cluster. A server connects to a running instance of a cluster, and NetEspProject, NetEspPublisher, and NetEspSubscriber 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 three access modes 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 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. To register an entity, call the **select_with(...)** method on the entity you want to monitor (NetEspServer, NetEspPublisher, NetEspSubscriber, or NetEspProject), passing in the NetEspSelector instance together with the events to monitor for. Then, call the **select(...)** method on the NetEspSelector instance, which blocks until a monitored update occurs in the background. The function returns a list of NetEspEvent 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 NetEspSelector. The event is then handled in your own thread.

Starting the SDK

Before performing operations, start the SDK.

1. Create an error message store for the following:
   ```
   NetEspError error = new NetEspError();
   ```
2. Get an instance of the .NET SDK and invoke the start method:
   ```
   NetEspSdk s_sdk = NetEspSdk.get_sdk();
   s_sdk.start(espError);
   ```

Connecting to a Server

When you have started the SDK, connect to a server.

Prerequisites

Start the SDK.

Task

1. Create a URI object:
   ```
   NetEspUri uri = new NetEspUri();
   uri.set_uri("esp://myserver:19011", error);
   ```
2. Create your credentials. The type of credentials depends on which security method is configured with the cluster:
   ```
   NetEspCredentials creds = new
   NetEspCredentials(NetEspCredentials.NET_ESP_CREDENTIALS_T.NET_ESP
   _CREDENTIALS_SERVER_RSA);
   creds.set_user("auser");
   creds.set_password("1234");
   creds.set_keyfile("..\\test_data\\keys\\client.pem");
   ```

3. Set options:

```
NetEspServerOptions options = new NetEspServerOptions();
options.set_mode(NetEspServerOptions.NET_ESP_ACCESS_MODE_T.NET_CA
LLBACK_ACCESS);
```

4. Connect to the server:

```
server = new NetEspServer(uri, creds, options);
int rc = server.connect(error);
```

Getting and Connecting to a Project

To publish or subscribe to data, get and connect to a project instance.

1. Get the project:

```
NetEspProject project = server.get_project(“workspacename”,
“projectname”,
error);
```

2. Connect to the project:

```
project.connect(error);
```

Publishing

The SDK provides various options for publishing data to a project.

The steps involved in publishing data are:

1. Create a NetEspPublisher from a previously connected NetEspProject instance.
2. Create a NetEspMessageWriter for the stream to publish to. You can create multiple NetEspMessageWriters from a single NetEspPublisher.
3. Create a NetEspRelativeRowWriter.
4. Format the data buffer to publish using NetEspRelativeRowWriter methods.
5. Publish the data.

While NetEspPublisher is thread-safe, NetEspMessageWriter and NetEspRelativeRowWriter are not. Therefore, ensure that you synchronize access to the latter two.

The SDK provides a number of options to tune the behavior of a NetEspPublisher. Specify these options using NetEspPublisherOptions when creating the NetEspPublisher. Once created, options cannot be changed. Like all other entities in the SDK, publishing also 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. Buffering is possible only in direct access mode. Direct and 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 either 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 envelope batches, 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.

Sync mode settings control the publishing handshake between the SDK and the platform. By default, the SDK sends data to the platform without waiting for acknowledgement. But if sync mode is set to true, the SDK waits for acknowledgement from the platform before sending the next batch of data. This provides an application level delivery guarantee, but it reduces throughput.

Publishing in async mode improves throughput, but does not provide an application level delivery guarantee. Since TCP does not provide an application level delivery guarantee either, data in the TCP buffer could be lost when a client exits. Therefore, a commit must be executed before a client exit when publishing in async mode.

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

```
net_esp_publisher_options_set_access_mode(options, CALLBACK_ACCESS,
error);
net_esp_publisher_set_callback(publisher, events, callback, NULL,
error)
net_esp_publisher_connect(publisher, error);
```

Publishing in Direct Access Mode

Publishing in direct access mode is a multistep process that involves creating and connecting to a publisher, then identifying the stream to publish to and the data to publish.

The following code snippets illustrate one way of publishing data. Adapt this sample as necessary to suit your specific publishing scenario.

1. Create a publisher:

   ```
   NetEspPublisher publisher = project.create_publisher(null,
   error);
   ```

2. Connect to the publisher:

   ```
   Publisher.connect(error);
   ```

3. Get a stream:

   ```
   NetEspStream stream = project.get_stream("WIN2", error);
   ```

4. Get the Message Writer:

   ```
   NetEspMessageWriter writer = publisher.get_message_writer(stream,
   error);
   ```

5. Get and start the Row Writer, and set an opcode to insert one row:

   ```
   NetEspRelativeRowWriter rowwriter =
   writer.get_relative_row_writer(error);
   rowwriter.start_row(error);
   rowwriter.set_opcode(1, error);
   ```

6. Set the column values sequentially, starting from the first column. Call the appropriate set method for the data type of the column. For example, if the column type is string:

   ```
   rc = rowwriter.set_string(“some value”, error);
   ```

7. When you have set all column values, end the row:

   ```
   rc = rowwriter.end_row(error);
   ```

8. Publish the data:

   ```
   rc = publisher.publish(writer, error);
   ```

Subscribing

The SDK provides various options for subscribing to a project.

The steps involved in subscribing to data using the SDK are:

1. Create a NetEspSubscriber from a previously connected NetEspProject instance.
2. Subscribe to streams.
3. In direct access mode, retrieve events using NetEspSubscriber.get_next_event(). In callback and select access modes, the event is generated by the SDK and passed back to user code.
4. For data events, retrieve NetEspMessageReader. 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.
5. Retrieve one or more NetEspRowReader. Use the methods in NetEspRowReader to read in individual fields.

Subscribing to a Stream in Callback Mode

Subscribing in callback mode is a multistep process that involves creating a subscriber and callback registry, connecting to the subscriber, and then subscribing to a stream.

The following code snippets illustrate one way of subscribing. Adapt this sample as necessary to suit your particular subscription scenario.

1. Create a subscriber:

```
NetEspSubscriberOptions options = new NetEspSubscriberOptions();
options.set_mode(NetEspSubscriberOptions.NET_ESP_ACCESS_MODE_T.NE
T_CALLBACK_ACCESS);
NetEspSubscriber subscriber =
project.create_subscriber(options,error);
```

2. Create the callback registry:

```
NetEspSubscriber.SUBSCRIBER_EVENT_CALLBACK callbackInstance = new
NetEspSubscriber.SUBSCRIBER_EVENT_CALLBACK(subscriber_callback);
subscriber.set_callback(NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT
.NET_ESP_SUBSCRIBER_EVENT_ALL, callbackInstance, null, error);
```

3. Connect to the subscriber:

```
subscriber.connect(error);
```

4. Subscribe to a stream:

```
subscriber.subscribe_stream(stream, error);
```

   - Callback function implementation:

```
Public static void subscriber_callback(NetEspSubscriberEvent
event, ValueType
```

```
data) {
        switch (evt.getType())
            {
               case (uint)
(NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_
EVENT_CONNECTED):
                    Console.WriteLine("the callback happened:
connected!");
                    break;
(uint)
( NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER
_EVENT_DATA):
        //handleData
        ...
            break;

                default:
                    break;
            }
}//end subscriber_callback
```

- handleData implementation:

```
NetEspRowReader row_reader = null;
while ((row_reader = evt.getMessageReader().next_row(error)) !=
null) {
             for (int i = 0; i < schema.get_numcolumns(); ++i)
{
                    if ( row_reader.is_null(i) == 1) {
                        Console.Write("null, ");
                        continue;
                    }
                    switch
(NetEspStream.getType(schema.get_column_type((uint)i, error)))
                    {
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_INTEGER:
                           ivalue = row_reader.get_integer(i,
error);
                            Console.Write(ivalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_LONG:
                        lvalue = row_reader.get_long(i, error);
                            Console.Write(lvalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_FLOAT:
                            fvalue = row_reader.get_float(i,
error);
                            Console.Write(fvalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_STRING:
                            svalue = row_reader.get_string(i,
error);
                            Console.Write(svalue);
```

```
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_DATE:
                        dvalue = row_reader.get_date(i, error);
                            Console.Write(dvalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_TIMESTAMP:
                          tvalue = row_reader.get_timestamp(i,
error);
                            Console.Write(tvalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BOOLEAN:
                         boolvalue = row_reader.get_boolean(i,
error);
                            Console.Write(boolvalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BINARY:
                            uint buffersize = 256;
                           binvalue = row_reader.get_binary(i,
buffersize, error);

Console.Write(System.Text.Encoding.Default.GetString(binvalue)
+ ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_INTERVAL:
                       intervalue = row_reader.get_interval(i,
error);
                            Console.Write(intervalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY01:
                          mon = row_reader.get_money(i, error);
                        Console.Write(mon.get_long(error) + ",
");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY02:
                            lvalue =
row_reader.get_money_as_long(i, error);
                            Console.Write(lvalue + ", ");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY03:
                          mon = row_reader.get_money(i, error);
                        Console.Write(mon.get_long(error) + ",
");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY10:
                          mon = row_reader.get_money(i, error);
                        Console.Write(mon.get_long(error) + ",
");
```

```
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_MONEY15:
                          mon = row_reader.get_money(i, error);
                        Console.Write(mon.get_long(error) + ",
");
                            break;
                        case
NetEspStream.NET_DATA_TYPE_T.NET_ESP_DATATYPE_BIGDATETIME:
                          bdt2 = row_reader.get_bigdatetime(i,
error);
                            long usecs =
bdt2.get_microseconds(error);
                            Console.Write(usecs + ", ");
                            break;
                    }
                }
            }

            rc = subscriber.disconnect(error);
}
```

Stopping the SDK

When your operations are complete, stop the .NET SDK to free up resources.

To stop the .NET SDK, use:

```
s_sdk.stop(espError);
```

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 NetEspServer instance even if all known manager instances are down. Instead, it generates a NET_ESP_SERVER_EVENT_STALE event. If it manages to reconnect after a (configurable) number of attempts, it generates a NET_ESP_SERVER_EVENT_UPTODATE event. Otherwise, it disconnects and generates a NET_ESP_SERVER_EVENT_DISCONNECTED event.
- **Project failovers –** an Event Stream Processor cluster allows a project to be deployed with failover. Based on the configuration settings, a cluster restarts a project if it detects that is has exited (however, projects are not restarted if they are explicitly closed by the user). To support this, you can have NetEspProject instances monitor the cluster for project restarts and then reconnect. This works only in callback or select modes. A NET_ESP_PROJECT_EVENT_STALE is generated when the SDK detects that the project has gone down. If it is able to reconnect, it generates a NET_ESP_PROJECT_EVENT_UPTODATE event. Otherwise, it generates a NET_ESP_PROJECT_EVENT_DISCONNECTED event.
- **Active-active deployments –** You can deploy a project in active-active mode. In this mode, the cluster starts two instances of the project, a primary instance and a secondary instance. Any data published to the primary instance 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, NetEspProject 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 NetEspProject, 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 NET_ESP_SUBSCRIBER_EVENT_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 NET_ESP_SUBSCRIBER_EVENT_SYNC_START event, followed by the data events, and finally a NET_ESP_SUBSCRIBER_EVENT_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 *.NET 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