.NET SDK Guide

SAP Sybase Event Stream Processor 5.1 SP04

DOCUMENT ID: DC01619-01-0514-01
LAST REVISED: November 2013

Contents

Migration from Aleri Streaming Platform

The SDK interface provided by SAP® 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 functions 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 works similarly to how Platform and Publisher objects were called in ASP. Each call blocks until the task completes or results in an error. In ESP, you can also use this mode for Subscribers.

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 also use this method for other entities.

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 functionality when working

in callback or select access modes. Additional functionality includes checking if a cluster or project has gone down and monitoring the back-end for restarts.

Publishing

In DIRECT access mode, you can now have the SDK run a background thread when publishing for 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

The SAP 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. 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 the Subscriber object subscribes to data streams.

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 great flexibility in structuring access to the entities exposed by the API. There are three modes that can be used to access entities: direct, callback, and select.

Direct access is the default mode when retrieving an entity. In this mode, all calls return when an error occurs or the operation completes successfully. There are no events generated later, so there is no need to have an associated event handler.

In callback access, an event handler must be associated with the request. Most calls to the entity return immediately, but 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.

You can register multiple callbacks on each publisher or subscriber by calling `int32_t NetEspPublisher::set_callback (uint32_t events, PUBLISHER_EVENT_CALLBACK^ callback, IntPtr^ user_data, NetEspError^ error)` or `int32_t NetEspSubscriber::set_callback (uint32_t events, SUBSCRIBER_EVENT_CALLBACK^ callback, IntPtr^ user_data, NetEspError^ error)` multiple times. Each registered handler gets the same events.

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 ESP project. 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 ESP project and are read together from the socket by the project. This improves network throughput. Transaction batches, like envelope batches, are also written and read in groups. However, with transaction batches, the ESP project only processes the group if all the rows in the batch are processed successfully. If one fails, the whole batch is rolled back.

When publishing is buffered, you can specify how the SDK batches rows in NetEspPublisherOptions. NET_EXPLICIT_BLOCKING lets you control the batches by using start transaction and end block calls. NET_AUTO_BLOCKING ignores these calls and batches rows internally. The default mode is NET_NO_BLOCKING.

Sync mode settings control the publishing handshake between the SDK and the ESP project. By default, the SDK sends data to the ESP project without waiting for acknowledgement. If sync mode is set to true, the SDK waits for acknowledgement from the ESP project 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.

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 Event Stream Processor project (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 Event Stream Processor sent.

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 are provided to illustrate one way of publishing and not as a complete, working example. Adapt this sample as necessary to suit your specific publishing scenario.

1. Create a publisher:

```
NetEspCredentials creds = new NetEspCredentials
(NetEspCredentials.NET_ESP_CREDENTIALS_T.NET_ESP_CREDENTIALS_USER
_PASSWORD);
```

```
creds.set_user("user");
creds.set_password("password");
NetEspPublisher publisher = project.create_publisher(creds,
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);
```

Working Example

The previous sample code on publishing is provided for illustration purposes, but does not comprise a full, working example.

SAP Sybase Event Stream Processor ships with fully functioning examples you can use as a starting point for your own projects. Examples for publishing are located in:

`%ESP_HOME%\examples\net\PublisherExample` (Windows)

`$ESP_HOME/examples/net/PublisherExample` (Linux and Solaris)

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 one or more streams. Call `int32_t NetEspSubscriber::subscribe_stream (NetEspStream^ stream, NetEspError^ error )` for each stream you are connecting to.
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 ESP project. It may consist of a single data row or a 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 are provided to illustrate one way of subscribing and not as a complete, working example. 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);
}
```

Working Example

The previous sample code on subscribing is provided for illustration purposes, but does not comprise a full, working example.

SAP Sybase Event Stream Processor ships with fully functioning examples you can use as a starting point for your own projects. Examples for subscribing are located in:

`%ESP_HOME%\examples\net\SubscriberExample` (Windows)

`$ESP_HOME/examples/net/SubscriberExample` (Linux and Solaris)

Subscribing with Guaranteed Delivery

Use guaranteed delivery (GD) to ensure that events are still delivered to the subscriber if the connection is temporarily lost or the server is restarted.

Prerequisites

Enable guaranteed delivery in a window and attach a log store in the CCL. To receive checkpoint messages from the server on streams using GD with checkpoint, set the Auto Checkpoint parameter in the project configuration file. The client may also receive checkpoint messages if the consistent recovery option is turned on and a publisher commits a message.

Task

Guaranteed delivery is a delivery mechanism that preserves events produced by a window, keeps data in a log store, and tracks events consumed by GD subscribers. For more information on guaranteed delivery, see the *Programmers Guide*.

A CCL project can be set to checkpoint after a number of messages pass through it. Once the configured number of messages pass through the project, the server commits the log store and sends a checkpoint message to the subscriber. This indicates that all messages up to the checkpoint sequence number are safely logged in the system.

A subscriber must indicate to the server when it has processed the messages and can recover them without the server. The subscriber can call `NetEspPublisher.commit_gd` at any time to tell the server the sequence number of the last message that has been processed. The commit call ensures that the server will not resend messages up to and including the last sequence number committed, and allows it to reclaim resources consumed by these messages. The subscriber should not commit sequence numbers higher than the sequence number received via the last checkpoint message. This ensures that no data is lost if the server restarts.

1. Request a GD subscription by calling `NetEspSubscriberOptions.set_gd_session(string session_name)` and creating the NetEspSubscriber object.
2. Create and connect a NetEspPublisher object.
3. Check if streams have GD or GD with checkpoint enabled by calling `NetEspStream.is_gd_enabled(NetEspError^ error)` and `NetEspStream.is_checkpoint_enabled(NetEspError^ error)`.
4. Retrieve active and inactive GD sessions by calling `NetEspProject.get_active_gd_sessions(NetEspError^ error)` and `NetEspProject.get_inactive_gd_sessions NetEspError^ error)`.
5. Retrieve the checkpoint sequence number for the last checkpointed data by calling `NetEspSubscriberEvent.get_checkpoint_sequence_number(NetEspError^ error)`.
6. Tell the server that the subscriber has committed messages up to a given sequence number and no longer needs them by calling `NetEspPublisher.commit_gd(String^ session_name, array<int32_t>^ stream_ids, array<int64_t>^ seq_nos, NetEspError^ error)`.
7. Cancel the GD session by closing the subscriber or by calling `NetEspProject.cancel_gd_subscriber_session(String^ gd_session, NetEspError^ error)`.

Example

```
// To connect to ESP server
NetEspError espError = new NetEspError();
NetEspSdk s_sdk = SYBASE.Esp.SDK.NetEspSdk.get_sdk();
s_sdk.start(espError);
```

```
        NetEspCredentials creds = new
NetEspCredentials(NetEspCredentials.NET_ESP_CREDENTIALS_T.NET_ESP_C
REDENTIALS_USER_PASSWORD);
        creds.set_user("sybase");
        creds.set_password("sybase");
        NetEspUri uri = new NetEspUri();
        uri.set_uri("esp://localhost:19011", espError);
        NetEspServerOptions soptions = new NetEspServerOptions();
       NetEspServer server = s_sdk.get_server(uri, creds, soptions,
espError);
        server.connect(espError);

        // To connect to an ESP project
        NetEspProject project = new NetEspProject();
        NetEspProjectOptions projoptions = new
NetEspProjectOptions();
       project = server.get_project("workspace", "gd", projoptions,
espError);
        project.connect(espError);

        // To create a GD subscriber
        NetEspSubscriberOptions suboptions = new
NetEspSubscriberOptions();
        suboptions.set_gd_session("GD999");
        NetEspSubscriber subscriber =
project.create_subscriber(suboptions, espError);

        // To create a publisher to commitGD message
        NetEspPublisher publisher = project.create_publisher(null,
espError);
        publisher.connect(espError);

        //check GD status/mode
        NetEspStream stream1 = project.get_stream("In1", espError);
        subscriber.subscribe_stream(stream1, espError);
        subscriber.connect(espError);
        stream1.is_gd_enabled(espError);
        stream1.is_checkpoint_enabled(espError);
        subscriber.is_gd_enabled();

        // retrieve GD sessions
        project.get_active_gd_sessions(espError);
        project.get_inactive_gd_sessions(espError);

        NetEspSubscriberEvent event1;
        Boolean done = false;

        while (!done)
        {
            event1 = subscriber.get_next_event(espError);

            switch (event1.getType())
            {
                case
(uint)NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_
EVENT_DATA:
```

```
                    NetEspMessageReader reader =
event1.getMessageReader();
                  NetEspRowReader rows = reader.next_row(espError);
                    while (rows != null)
                    {
                        int intcol1 = rows.get_integer(1, espError);
                        Console.Out.Write(intcol1);

                    string stringco2 = rows.get_string(2, espError);
                        Console.Out.WriteLine(" " + stringco2);

                        rows = reader.next_row(espError);
                    }
                    break;
                case
(uint)NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_
EVENT_CLOSED:
                    done = true;
                    break;
                case
(uint)NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_
EVENT_DISCONNECTED:
                    done = true;
                    break;
                case
(uint)NetEspSubscriber.NET_ESP_SUBSCRIBER_EVENT.NET_ESP_SUBSCRIBER_
CHECKPOINT:

                    // retrieve the sequence number returned by
NET_ESP_SUBSCRIBER_CHECKPOINT message
                    long seq_val =
event1.get_checkpoint_sequence_number(espError);
                    if (seq_val > 0)
                    {
                        Console.Out.WriteLine("SeqNo #" + seq_val);

                        int[] idArray = new int[1];
                        long[] seqArray = new long[1];
                        idArray[0] = stream1.get_id();
                        seqArray[0] = seq_val;

                    // commitGD message for a single stream with the
corresponding last checkpointed sequence number
                    publisher.commit_gd("GD999", idArray, seqArray,
espError);
                    }
                    break;
                default:
                    break;
            }
        }

       // cancel GD session and disconnect subscriber and publisher
before exit
        project.cancel_gd_subscriber_session("GD999", espError);
```

```
        subscriber.disconnect(espError);
        server.disconnect(espError);
```

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 it 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.

  When the SDK reconnects, entities obtained from the project are no longer valid. This includes publishers, subscribers, message readers/writers, and row readers/writers. After reconnecting, recreate these objects from the project.

  In direct access mode, the SDK does not monitor the cluster for restarts. If a communication error occurs, the project object and all project-related entities are invalidated. Close the project, which also closes any elements it contains, then create a new project object and reconnect. The following example shows one way of doing this:

```
// encountered error
project.close(esperror);
project = server.get_project(workspace, pname, projoptions,
esperror);
rc = project.connect(esperror);
// if the project has been successfully restarted this will
succeed
if (!rc) {
// exit or loop
}
// create publisher or subscriber and proceed
```

- **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 given 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.

Examples

ESP includes several working examples for the .NET SDK.

PublisherExample	Demonstrates the basics of SDK use
PublisherAnySchema	Publishes using stream metadata
SubscriberCallback	Subscribes using the callback mechanism
SubscriberExample	Displays published data
SubscriberGdExample	Subscribes using the guaranteed delivery mechanism
UpdateShineThrough	Publishes updates using ShineThrough

These examples and a readme file with instructions for running them are located at `ESP_HOME\examples\net`.

API Reference

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

To access the API level documentation:

1. Navigate to `<Install_Dir>\ESP-5_1\doc\sdk\net`.
2. Launch `index.html`.

Index