C SDK Guide

SAP Sybase Event Stream Processor 5.1 SP04

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

In the SAP Sybase Event Stream Processor C SDK, all entities exposed by the SDK have a common life cycle and multiple 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. 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 EspProject, EspPublisher, and EspSubscriber 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 operations on an entity 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 `ESPAPICALL int32_t esp_publisher_set_callback(EspPublisher * publisher, uint32_t events, PUBLISHER_CALLBACK_T callback, void * user_data, EspError * error);` or `ESPAPICALL int32_t esp_subscriber_set_callback(EspSubscriber * subscriber, uint32_t events, SUBSCRIBER_CALLBACK_T callback, void * user_data, EspError * error);` multiple times. Each registered handler gets the same events.

The following example shows how an EspProject could be accessed 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:

```
    EspProjectOptions * options = esp_project_options_create(error);

    int rc = esp_project_options_set_access_mode(options,
CALLBACK_ACCESS, error);

    const char * temp = "esp://host.domain.com/workspace/project";
    EspUri * uri = esp_uri_create_string(temp, error);

    // Create credentials to authenticate with project. Assume
cluster is setup to use user password authentication
    EspCredentials * creds =
esp_credentials_create(ESP_CREDENTIALS_USER_PASSWORD, error);
    esp_credentials_set_user(creds, "user", error);
    esp_credentials_set_password(creds, "password", error);

    EspProject * project = esp_project_get(uri, creds, options,
error);

    // If you are not going to reuse the credentials, you need to free
them
    esp_credentials_free(creds, error);

    rc = esp_project_set_callback(project, ESP_PROJECT_EVENT_ALL,
project_callback, NULL, error);

    rc = esp_project_connect(project, error);

    // The callback handler
    void project_callback(const EspProjectEvent * event, void * data)
    {
        EspProject * project = NULL;
        const EspError * error = NULL;
        int rc;
        uint32_t type;
        rc = esp_project_event_get_type(event, &type, NULL);

        switch (type) {
            case ESP_PROJECT_EVENT_CONNECTED:
               project = esp_project_event_get_project(event, NULL);
                break;
            case ESP_PROJECT_EVENT_DISCONNECTED:
               project = esp_project_event_get_project(event, NULL);
              esp_project_close(project, NULL);          // you can
call close inside a callback
                break;
            case ESP_PROJECT_EVENT_CLOSED:
            case ESP_PROJECT_EVENT_STALE:
            case ESP_PROJECT_EVENT_UPTODATE:
                break;
            case ESP_PROJECT_EVENT_ERROR:
                error = esp_project_event_get_error(event, NULL);
                break;
        }
    }
```

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

The following example uses a single selector to multiplex different entities.

```
// Assuming the EspServer, EspProject, EspPublisher, EspSubscriber
have been created with the correct options
// Not doing error checking, etc for clarity

    EspSelector * selector = esp_selector_create("server-select",
error);
    rc = esp_server_select_with(server, selector,
ESP_SERVER_EVENT_ALL, error);
    EspList * list = esp_list_create(ESP_LIST_EVENT_T, error);

    rc = esp_server_connect(m_server, error);

    uint32_t type;
    const void * ev;
    int c;
    int done = 0;

    while (!done)
    {
        esp_list_clear(list, error);
        rc = esp_selector_select(selector, list, error);

        c = esp_list_get_count(list, error);
        for (int i = 0; i < c; i++)
        {
            ev = esp_list_get_event(list, i, error);

            int cat = esp_event_get_category(ev, error);

            switch ( cat ) {
                case ESP_EVENT_SERVER:
                    srvevent = (EspServerEvent*) ev;
                  esp_server_event_get_type(srvevent, &type, error);
                    switch (type) {
                        // process server events
                        case ESP_SERVER_EVENT_CONNECTED:
                            break;
                        // .....
                    }
                default:
                    break;

                case ESP_EVENT_PROJECT:
                    prjevent = (EspProjectEvent*) ev;
                 esp_project_event_get_type(prjevent, &type, error);
                    switch (type) {
                        // process project events
                        case ESP_PROJECT_EVENT_CONNECTED:
                            break;
                    }
                case ESP_EVENT_PUBLISHER:
                    {
                        pubevent = (EspPublisherEvent*) ev;
                      esp_publisher_event_get_type(pubevent, &type,
error);
                        switch (type) {
                            case ESP_PUBLISHER_EVENT_CONNECTED:
                                break;
                        }
                    }
                    break;

                case ESP_EVENT_SUBSCRIBER:
                    {
                        subevent = (EspSubscriberEvent*) ev;
                     esp_subscriber_event_get_type(subevent, &type,
error);
                        switch (type) {
                            case ESP_SUBSCRIBER_EVENT_CONNECTED:
                                break;
                        }
                        break;
                    }
            }
        }
    }
```

Starting and Stopping the C SDK

Start the C SDK before performing operations.

Initializing the C SDK prompts it to start its internal threads and register required resources. This call can be made from any thread, but it must be made before any other SDK functionality is used.

Example:

```
int rc;
EspError * error = esp_error_create();
rc = esp_sdk_start(error);
if (rc) {
}
```

Once the application using C SDK is ready to exit or its functionality is no longer needed, stop the C SDK. This stops its internal threads and releases any held resources.

Example:

```
rc = esp_sdk_stop(error);
if (rc) {
}
```

Multiple SDK start calls may be made. The C SDK requires corresponding number of stop calls to properly shutdown.

Publishing

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

The steps involved in publishing data are:

1. Create an EspPublisher for the project to publish to. You can create an EspPublisher directly or from a previously retrieved and connected EspProject object.
2. Create an EspMessageWriter for the stream to publish to. You can create multiple EspMessageWriters from a single EspPublisher.
3. Create an EspRelativeRowWriter.
4. Format the data buffer to publish using EspRelativeRowWriter methods.
5. Publish the data.

While EspPublisher is thread-safe, EspMessageWriter and EspRelativeRowWriter are not. Therefore, ensure that you synchronize access to the latter two.

The SDK provides a number of options to tune the behavior of an EspPublisher. Specify these options using EspPublisherOptions when creating the EspPublisher. 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 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.

When publishing is buffered, you can specify how the SDK batches rows in EspPublisherOptions. EXPLICIT_BLOCKING lets you control the batches by using start transaction and end block calls. AUTO_BLOCKING ignores these calls and batches rows internally. The default mode is NO_BLOCKING.

Sync mode settings control the publishing handshake between the SDK and the ESP project. By default, the SDK keeps sending 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.

There are certain considerations to keep in mind when using callback or select mode publishing. These modes are driven by the 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 ESP_PUBLISHER_EVENT_READY event.

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 ESP 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 ESP project sent.

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:

```
esp_publisher_options_set_access_mode(options, CALLBACK_ACCESS,
error);
esp_publisher_set_callback(publisher, events, callback, NULL,
error);
esp_publisher_connect(publisher, error);
```

The following code snippets show different ways of publishing data. The sample code provided here is for illustration only; it does not comprise a complete, working example.

The first example shows publishing in direct access mode with transaction blocks.

```
EspCredentials * creds =
esp_credentials_create(ESP_CREDENTIALS_USER_PASSWORD, error);
    esp_credentials_set_user(creds, “user”, error);
    esp_credentials_set_password(creds, “password”, error);

    // Create publisher with default options from an existing
EspProject
    publisher = esp_project_create_publisher(project, creds, error);
    esp_credentials_free(creds, error);
    // Connect the publisher
    int rc = esp_publisher_connect(publisher, error);

    // Retrieve the EspStream we want to publish to
    const EspStream * stream = esp_project_get_stream(project,
"Stream1", error);

    // Determine its schema
    const EspSchema * schema = esp_stream_get_schema(stream, error);

    // Create EspMessageWriter to publish to "Stream1"
    EspMessageWriter * writer = esp_publisher_get_writer(publisher,
stream, error);

    EspRelativeRowWriter * row_writer =
esp_message_writer_get_relative_rowwriter(writer, error);

    // Number of columns in "Stream1"
    int32_t numcols;
    esp_schema_get_numcolumns(schema, &numcols, error);

    int32_t intvalue = 10;
    bool inblock = false;

    // Your logic to determine how long to publish
    while (....) {
           // Your logic to determine whether to start a transaction
            if (!inblock) {
            esp_message_writer_start_transaction(writer, 0, NULL);
            inblock = true;
        }
        // Start a data row
        esp_relative_rowwriter_start_row(row_writer, NULL);
        int32_t coltype;

        for (int i = 0; i < numcols; ++i) {
            esp_schema_get_column_type(schema, i, &coltype, error);
            switch (coltype) {
                case ESP_DATATYPE_INTEGER:
                    esp_relative_rowwriter_set_integer(row_writer,
intvalue++, error);
                    break;
                // ...
                // Code to fill in other data types goes here
                // ...
              // NOTE - you must fill in all data fields, with NULLs
needed
                default:
                 esp_relative_rowwriter_set_null(row_writer, error);
                    break;
            }
        }
        // End the data row
        esp_relative_rowwriter_end_row(row_writer,
error);
```

```
       // Determine if the batch is to be ended, we code for 60 rows
per block
        if ((nrows % 60) == 0) {

            // End the batch started in
esp_message_writer_start_transaction()
            esp_message_writer_end_block(writer, error);

            // Publish the batch
            esp_publisher_publish(publisher, writer,
error);
            inblock = false;
        }
    }
    // Done with publishing
    esp_publisher_close(publisher, error);
```

This example shows publishing in callback access mode.

```
       // Create EspPublisherOptions
    int rc;
    EspPublisherOptions * options =
esp_publisher_options_create(error);

    // Set access mode
    rc = esp_publisher_options_set_access_mode(options,
CALLBACK_ACCESS, error);

    // Create EspPublisher using the options above from existing
EspProject
    publisher = esp_project_create_publisher(project, options,
error);

    // Free EspPublisherOptions
    esp_publisher_options_free(options, error);

    // Set callback handler
    rc = esp_publisher_set_callback(publisher,
ESP_PUBLISHER_EVENT_ALL, publish_callback,
    NULL, m_error);

    // Connect publisher
    rc = esp_publisher_connect(publisher, error);

    ...
    ...
    ...

    // Handler function
    void publish_callback(const EspPublisherEvent * event, void *
user_data)
    {
        EspPublisher * publisher = NULL;
        EspMessageWriter * mwriter = NULL;
        EspRelativeRowWriter * row_writer = NULL;
        EspProject * project = NULL;
        const EspStream * stream = NULL;
        const EspSchema * schema = NULL;

        EspError * error = esp_error_create();

        int rc;
        uint32_t type;

        publisher = esp_publisher_event_get_publisher(event, error);
        rc = esp_publisher_event_get_type(event, &type, error);

        switch (type)
        {
            case ESP_PUBLISHER_EVENT_CONNECTED:
                // EspProject, EspStream, EspSchema can be retrieved
from the EspPublisherEvent
                // if required
              project = esp_publisher_get_project(publisher, error);
                stream = esp_project_get_stream(project, "Stream1",
error);
                schema = esp_stream_get_schema(stream, error);
                break;

            case ESP_PUBLISHER_EVENT_READY:

                // Populate EspMessageWriter with data to publish

              rc = esp_publisher_publish(publisher, mwriter, error);
                break;

            case ESP_PUBLISHER_EVENT_DISCONNECTED:
                esp_publisher_close(publisher, error);
                break;

            case ESP_PUBLISHER_EVENT_CLOSED:
                break;
        }

        if (error)
            esp_error_free(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\cpp\SDK` (Windows)

`$ESP_HOME/examples/cpp/SDK` (Linux and Solaris)

Subscribing

The SDK provides various options for subscribing to a project.

Subscribing to data using the SDK involves the following steps:

1. Create an EspSubscriber object. This can be created directly or retrieved from EspProject.
2. Connect the EspSubscriber.
3. Subscribe to one or more streams. Call `esp_subscriber_subscribe(EspSubscriber * subscriber, const EspStream * stream, EspError * error);` for each stream you are connecting to.
4. In direct access mode, retrieve events using **esp_subscriber_get_next_event()**. In callback and select access modes, the event is generated by the SDK and passed back to user code.
5. For data events, retrieve EspMessageReader. This encapsulates a single message from the ESP project. It may consist of a single data row or a block with multiple data rows.
6. Retrieve one or more EspRowReaders. Use the methods in EspRowReader to read in individual fields.

The sample code provided here is for illustration only; it does not comprise a complete, working example. This example shows subscribing to a stream using direct access mode with default options:

```
EspError * error = esp_error_create();
esp_sdk_start(error);

EspUri * project_uri = esp_uri_create_string("esp://server:port//
default/vwap", error);

EspCredentials * creds =
esp_credentials_create(ESP_CREDENTIALS_USER_PASSWORD, error);
esp_credentials_set_user(creds, “user”, error);
esp_credentials_set_password(creds, “password”, error);

EspProject * project = esp_project_get(project_uri, creds, NULL,
error);

rc = esp_project_connect(project, error);

// Reusing credentials for the subscriber
EspSubscriber * subscriber = esp_project_create_subscriber(project,
creds, error);

// Now free credentials
esp_credentials_free(creds, error);

rc = esp_subscriber_connect(subscriber, error);
```

```
EspStream * stream = esp_project_get_stream(project, "Trades",
error);
rc = esp_subscriber_subsribe(subscriber, stream, error);

while (true) {
  EspSubscriberEvent * event =
esp_subscriber_get_next_event(subscriber, error);

  // process event data

  // delete event
  esp_subscriber_event_free(event);
}

esp_subscriber_close(subscriber, error);
esp_sdk_close();
```

If the event is an ESP_SUBSCRIBER_EVENT_DATA event, it contains field data. This is a typical example of reading data from a subscribe event:

```
// stream for this event
    const EspStream * stream = esp_subscriber_event_get_stream(event,
error);

    // get message reader
    EspMessageReader * reader =
esp_subscriber_event_get_reader(event, error);

    int rc = esp_message_reader_is_block(reader, &flag,
error);

    // get the stream schema if you do not have it
    const EspSchema * schema = esp_stream_get_schema(stream,
error);

    EspRowReader * row_reader;

    int32_t int_value;
    int numcolumns = 0, numrows = 0;
    int type;
    // need to know how many columns are there
    rc = esp_schema_get_numcolumns(schema, &numcolumns,
error);

    // loop until we finish all rows
    while ((row_reader = esp_message_reader_next_row(reader,
error)) != NULL) {

        for (int i = 0; i < numcolumns; ++i) {
            // if column is null, skip
            rc = esp_row_reader_is_null(row_reader, i, &flag,
error);
            if ( flag )
                continue;
           rc = esp_schema_get_column_type(schema, i, &type, error);
            switch ( type ) {
                case ESP_DATATYPE_INTEGER:
                    rc = esp_row_reader_get_integer(row_reader, i,
&int_value, error);
                    break;
                case ESP_DATATYPE_LONG:
                    rc = esp_row_reader_get_long(row_reader, i,
&long_value, error);
                    break;
                case ESP_DATATYPE_FLOAT:
                    rc = esp_row_reader_get_float(row_reader, i,
&double_value, error);
                // ...
                // other data types
                // ...
            }
        }
    }
```

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 `esp_publisher_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 `esp_subscriber_options_set_gd_session(EspSubscriberOptions * options, char * session_name, EspError * error)` and creating the Subscriber object.
2. Create and connect a Publisher object.
3. Check if streams have GD or GD with checkpoint enabled by calling `esp_stream_is_gd_enabled(const EspStream * stream, EspError * error)` and `esp_stream_is_checkpoint_enabled(const EspStream * stream, EspError * error)`.
4. Retrieve active and inactive GD sessions by calling `esp_project_get_active_gd_sessions(EspProject * Project, EspList * gd_sessions, EspError * error)` and `esp_project_get_inactive_gd_sessions(EspProject * Project, EspList * gd_sessions, EspError * error)`.
5. Retrieve the checkpoint sequence number for the last checkpointed data by calling `esp_subscriber_event_get_checkpoint_sequence_number(const EspSubscriberEvent * event, int64_t * seq_val, EspError * error)`.
6. Tell the server that the subscriber has committed messages up to a given sequence number and no longer needs them by calling `esp_publisher_commit_gd(EspPublisher * publisher, char * gd_name, int32_t stream_ids[], int64_t seq_nos[], int32_t len, EspError * error)`.
7. Cancel the GD session by closing the subscriber or by calling `esp_project_cancel_gd_subscriber_session(EspProject * project, char * gd_session, EspError * error)`.

Example

```
    int rc = esp_project_connect(project, g_error);
    if (rc != 0) { print_error_and_exit(g_error, __LINE__); }

    EspSubscriberOptions *options =
esp_subscriber_options_create(g_error);

   // This will set the GD session. Provide a unique GD session name
(GD999 in the example).
    // The GD session will only be created once
esp_project_create_subscriber(…) is called.
    esp_subscriber_options_set_gd_session(options, "GD999",
g_error);

    EspSubscriber * subscriber =
esp_project_create_subscriber(project, options, g_error);
    if (NULL == subscriber) { print_error_and_exit(g_error,
__LINE__); }

    // Here this example expects that the CCL has a window called
```

```
"In1" with an attached log store and
   // guaranteed delivery enabled. You can use stream name from your
CCL.
   const EspStream * stream = esp_project_get_stream(project, "In1",
g_error);
    if (NULL == stream) { print_error_and_exit(g_error, __LINE__); }

    // This call checks whether stream in the CCL has GD enabled.
    rc = esp_stream_is_gd_enabled(stream, g_error);
    if (rc == 0)
    {
        printf("%s\n", "stream has GD enabled!”);
    }

    // This call checks whether the checkpoint is set in the CCL
project.
    rc = esp_stream_is_checkpoint_enabled(stream, g_error);
    if (rc == 0)
    {
        printf("%s\n", "chkpointEnable...OK");
    }
    rc = esp_subscriber_connect(subscriber, g_error);
    if (rc != 0) { print_error_and_exit(g_error, __LINE__); }

    // This call checks whether it is a GD subscriber.
    rc = esp_subscriber_is_gd_enabled(subscriber, g_error);
    if (rc == 0)
    {
        printf("%s\n", "GD subscriber!");
    }

    EspList * gd_sessions  = esp_list_create(ESP_LIST_STRING_T,
g_error);

    // This gets all the active gd_sessions and populates the
gd_sessions list.
    rc =  esp_project_get_active_gd_sessions(project, gd_sessions,
g_error);

    const char * gds = esp_list_get_string(gd_sessions, 0, g_error);
    printf("%s", gds);

    while (!done)
    {
         event = esp_subscriber_get_next_event(subscriber, g_error);
          if (esp_error_get_code(g_error) != 0)
{ print_error_and_exit(g_error, __LINE__); }

          rc = esp_subscriber_event_get_type(event, &eventType,
g_error);
          if (rc != 0)
          {
                esp_subscriber_event_free(event, g_error);
                print_error_and_exit(g_error, __LINE__);
          }
          switch(eventType)
          {
                case ESP_SUBSCRIBER_EVENT_DATA:
                      // Process events
                      break;

              // This event type indicates that the event is a check
point event. Check point events are received only
                // if CCR file has "auto-cp-trans" set.
               // You can issue GD commit at any time after you have
processed the events and can recover it without the ESP server.
             // This example assumes that CCL project is check point
enabled.
                case ESP_SUBSCRIBER_CHECKPOINT:
                      printf("<checkpoint-is-received/>\n");
                      int64_t seq_val;

                    // Get the check point sequence number from the
check point event.
                      rc =
esp_subscriber_event_get_checkpoint_sequence_number(event,
&seq_val, g_error);
                      if (rc == 0)
                      {
                             printf("SeqNo # %ld\n", seq_val);
                             int32_t ids[1];
                             esp_stream_get_id(stream, &ids[0],
g_error);
                             printf("stream id = %d\n", ids[0]);
                             int64_t sq[1];
                             sq[0] = seq_val;
                             printf("seq = %d\n", sq[0]);
                         // Do extra steps to make sure that all the
events up to this seq_val are consumed
                           // by third party applications or safely
logged so that they can be recovered without
                             // ESP server if needed.
                          // Code for logging or handing off to 3rd
party applications goes here…

                         // The following call sends commit message
to server indicating that it no longer needs
                          // events up to seq_val. In this scenario
(GD + check point), you should not commit sequence
                         // numbers higher than the sequence number
received from the server to safely recover data when
                             // server is restarted.
                            rc = esp_publisher_commit_gd(publisher,
"GD999", ids, sq, 1, g_error);
                             printf("rc = %d\n",
rc);
                      }
                      break;
                default:
                      break;
          }
          esp_subscriber_event_free(event, g_error);
    }

    // This cancels the GD subscriber session. When you close a 
subscriber, this call is internally called and GD session is
    // cancelled. This call is here to show how to use the API.
    rc = esp_project_cancel_gd_subscriber_session(project, "GD999", 
g_error);

    if (rc == 0) { printf("cancelled.... SUCCESS!\n"); }
```

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\cpp\SDK` (Windows)

`$ESP_HOME/examples/cpp/SDK` (Linux and Solaris)

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. So, if any one manager in the cluster goes down, the SDK tries to reconnect to another instance. The SDK returns an error only if connections to all known instances fail. 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 an EspServer instance even if all known manager instances are down. Instead, it generates an ESP_SERVER_EVENT_STALE event. If it manages to reconnect after a (configurable) number of attempts, it generates an ESP_SERVER_EVENT_UPTODATE. Otherwise, it disconnects and generates an 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 EspProject instances monitor the cluster for project restarts and then reconnect. This works only in callback or select modes. An ESP_PROJECT_EVENT_STALE event is generated when the SDK detects that the project has gone down. If it is able to reconnect, it generates an ESP_PROJECT_EVENT_UPTODATE event. Otherwise, it generates an 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 communication error
  esp_project_close(project, error);
  project = esp_project_get(uri, creds, NULL, error);
  rc = esp_project_connect(project, error);
  // 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 such active-active deployments. When connected to an active-active deployment, if the currently connected instance goes down, EspProject 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 EspProject, 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 an 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 an ESP_SUBSCRIBER_EVENT_SYNC_START event, followed by the data events, and finally an 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 C SDK.

PublisherExample	Demonstrates the basics of SDK use
PublisherAnySchemaExample	Publishes using stream metadata
SubscriberCallbackExample	Subscribes using the callback mechanism
SubscriberExample	Displays published data
Subscriber_gd_example	Subscribes using the guaranteed delivery mechanism
UpdateShineThroughExample	Publishes updates using ShineThrough

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

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/c`.
2. Launch `index.html`.

Index