SYBASE®
An SAP® Company

C SDK ガイド

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01750-01-0500-01
改訂： 2011 年 12 月

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

目次

Aleri Streaming Platform からの移行 1
エンティティのライフサイクルとアクセス・モード 3
パブリッシュ 9
サブスクライブ 15
フェールオーバの処理 19
API リファレンス 21
索引 23

Aleri Streaming Platform からの移行

Sybase® Event Stream Processor (ESP) の SDK インターフェイスは、Aleri Streaming Platform (ASP) の SDK インターフェイスとは異なります。Event Stream Processor の SDK が改善され、柔軟性およびパフォーマンスが向上し、クラスタ環境で実行しているプロジェクトに対応できるようになりました。

クラスタとプロジェクト
プロジェクトをクラスタで実行できるようになったため、プロジェクトにコマンドとコントロール・ホストおよびコントロール・ポートを使用してアクセスする必要がなくなりました。プロジェクトには、一意の ID が付けられており、その ID は、通常、クラスタ情報、ワークスペース名、プロジェクト名で構成される URI で示されます。SDK によって、内部的に URI が物理アドレスに解決されます。ESP のプロジェクト・オブジェクトは、ASP のプラットフォーム・オブジェクトにほぼ対応しています。Pub/Sub API は ESP サーバの API とは異なります。

注意： スタンドアロン・プロジェクトに接続する方法はありますが、今後のリリースで削除される予定の機能であるため、使用しないでください。

ESP SDK には、クラスタの構成および監視を行う新しい機能があります。ASP Pub/Sub API には、これらの機能に相当するものはありません。

アクセス・モード
ASP Pub/Sub では、プラットフォーム・オブジェクトとパブリッシャ・オブジェクトには、同期メソッド呼び出しを使用してアクセスしていました。サブスクライバ・オブジェクトには、コールバック・ハンドラが必要でしたが ESP では、この点が変更されました。すべてのエンティティ (サーバ、プロジェクト、パブリッシャ、サブスクライバ) には、直接メソッドを呼び出すか、コールバック ハンドラを使用してアクセスできます。また、ESP では、セレクト・アクセスという 3 つ目のメソッドが導入されました。

直接アクセス・モードは、以前のプラットフォーム・オブジェクトとパブリッシャ・オブジェクトが ASP で呼び出される方法と似ています。各オブジェクトの呼び出しは、タスクが完了するか、エラーが発生するまでブロックします。ESP では、サブスクライバの操作でこのモードも使用できます。

コールバック・モードでは、ハンドラ関数を登録すると、SDK でユーザ指定のイベントが発生した場合に、指定の関数が呼び出されます。ASP では、これはサブスクライバを操作する唯一の方法でした。ESP では、他のエンティティにもこのメソッドを使用できます。

セレクト・アクセス・モードでは、セレクタに複数のエンティティを登録することができます。また、単一のスレッドが、それらのエンティティのいずれかに関するイベントを待つように設定できます。この機能は、単一のスレッドで複数のファイル記述子の監視を行う select／poll メカニズムに似ています。

自動再接続とモニタリング
ASP では、Pub/Sub API によってホット／アクティブ・モードで作業している場合のピアへの自動接続がサポートされていました。ESP でも自動接続がサポートされていますが、コールバック・アクセス・モードまたはセレクト・アクセス・モードの場合、その他の機能が利用できるようになりました。追加された機能には、クラスタまたはプロジェクトが停止したかどうかを確認する機能、オプションでバックエンドを監視して再起動する機能があります。

パブリッシュ
直接アクセス・モードでは、SDK でパブリッシュのスループットを向上するためにバックグラウンド・スレッドが早く進むように設定できるようになりました。ASP では、上記のようなタスクは、Pub/Sub ユーザが行う必要がありました。

ASP では、メッセージは一時的な記憶領域 (ベクトル) でフォーマットされており、その記憶域には、Pub/Sub API を呼び出してバッファを作成する前にデータを入力する必要がありました。ESP では、直接バッファにデータが書き込まれるため、この操作は必要ありません。ESP SDK でメッセージを作成するとき、ユーザがブロックまたはローの開始を指定し、その後順にデータが入力されます。フィールドはスキーマに表示されるのと同じ順序で入力する必要があります。

サブスクライブ
ASP では、メッセージのデータをオブジェクトのコレクションとして利用できました。ESP SDK では、この手順は利用できず、標準のデータ型またはヘルパ・オブジェクト (Money、BigDatetime、Binary) として直接バッファを読み取るメソッドを利用できます。データ・フィールドには、ランダムにアクセスできます。

エンティティのライフサイクルとアクセス・モード

Sybase® Event Stream Processor の C SDK では、SDK で公開されているすべてのエンティティで共通のライフサイクルが採用され、複数のアクセス・モードが用意されています。

ユーザによる Event Stream Processor (ESP) SDK の操作は、SDK で使用されるエンティティを介して処理されます。主なエンティティには、サーバ、プロジェクト、パブリッシャ、サブスクライバがあります。これらのエンティティは SDK の機能に対応しています。たとえば、サーバ・オブジェクトは、クラスタの実行インスタンスを表します。プロジェクトは、クラスタに配備される単一のプロジェクトを、またパブリッシャ・オブジェクトは、実行プロジェクトへのデータのパブリッシュを扱います。

最初に取得するとき、エンティティはオープン状態であると見なされます。 エンティティがオープン状態であるとき、そのエンティティに関する特定の静的情報を取得できます。割り当てられたタスクを行うには、エンティティをクラスタ内の該当のコンポーネントに接続する必要があります。サーバはクラスタの実行インスタンスに接続し、EspProject、EspPublisher、EspSubscriber はすべてクラスタ内のプロジェクトの実行インスタンスに接続します。

エンティティが接続されると、クラスタ・コンポーネントで操作できます。エンティティの接続が解除されると、クラスタで操作できなくなりますが、SDK 内ではアクティブなままで、クラスタに再度接続できます。エンティティをクローズすると、操作できなくなり、SDK で再度要求されます。クローズされたエンティティを再度利用するには、エンティティのコピーを新たに取得します。

たとえば、プロジェクト・オブジェクトを取得して、クラスタ内のプロジェクトに接続できます。バックエンドのプロジェクトが停止すると、SDK のプロジェクトは切断イベントを受け取ります。この場合、手動で再接続するか、コールバック・モードを使用していて、再接続がサポートされる構成の場合、SDK で自動の再接続が試行されます。再接続できた場合、接続イベントが生成されます。ユーザがエンティティをクローズすると、エンティティはバックエンドのプロジェクトから切断され、SDK でプロジェクト・オブジェクトが再度要求されます。再接続するには、新規のプロジェクト・オブジェクトを取得する必要があります。

SDK によって、API で公開されるエンティティへのアクセスを非常に柔軟に構築できます。エンティティへのアクセスに使用できるモードは、次の 3 つです。

直接	同期
コールバック	非同期

セレクト	非同期

デフォルトのアクセス・モードは、直接モードです。エンティティの取得時は常にこのモードに設定されています。直接モードでは、エンティティのすべての操作は同期されます。 呼び出しが返されると、オペレーションが完了またはエラーが発生したと認識されます。

コールバック・アクセスでは、エンティティにハンドラ関数を登録します。エンティティに対するほとんどの呼び出しは、即座に返されます。リクエストが完了すると、該当のイベントが生成されます。SDK では、コールバック・メカニズムを実装するための内部スレッドが 2 つあります。更新スレッドは、現在登録されているすべてのエンティティを監視し、該当の更新のコールバックを確認します。更新がある場合、該当のイベントが生成され、ディスパッチ・スレッドのキューに追加されます。ディスパッチ・スレッドは、登録されているハンドラを呼び出し、ユーザ・コードによってそれらを処理します。

次の例は、EspProject にコールバック・モードでアクセスできる方法を示します。コールバック・モードを使用していて、コールバック・イベントを受け取る必要がある場合は、対象のエンティティの接続を呼び出す前にコールバック・ハンドラを登録します。

```
    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);


    EspProject * project = esp_project_get(uri, NULL, options,
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;
        }
    }
```

セレクト・アクセス・モードでは、単一のスレッドのさまざまなエンティティを多重化し、ファイル記述子を監視できます。この機能は、多くのシステムで使用されている select/poll メカニズムに似ています。 エンティティは、監視対象のイベントと一緒に EspSelector に登録されます。次に、**esp_selector_select(...)** を呼び出すと、監視対象の更新がバックグラウンドで発生するまでブロックします。 この関数では、EspEvent オブジェクトのリストが返されます。 まずイベントのカテゴリー (サーバ、プロジェクト、パブリッシャ、サブスクライバ) を決定し、次に適切なイベント・タイプを処理します。 セレクト・モードでは、SDK は 1 つのバックグラウンド更新スレッドを使用して更新を監視します。更新が検出されると、該当のイベントが作成され、EspSelector にプッシュされます。 その後、イベントは各スレッドで処理されます。

次は、1 つのセレクタを使用してさまざまなエンティティを多重化する例です。

```
// 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;
                    }
            }
        }
    }
```

パブリッシュ

SDK には、プロジェクトにデータをパブリッシュするためのオプションがいくつかあります。

データをパブリッシュする場合の手順は、次のとおりです。

1. パブリッシュ先のプロジェクトに対する EspPublisher を作成します。EspPublisher は、直接作成するか、以前に取得または接続した EspProject オブジェクトから作成できます。
2. パブリッシュ先のストリームに対する EspMessageWriter を作成します。 1 つの EspPublisher から複数の EspMessageWriter を作成できます。
3. EspRelativeRowWriter を作成します。
4. EspRelativeRowWriter メソッドを使用してパブリッシュするデータ・バッファをフォーマットします。
5. データをパブリッシュします。

EspPublisher はスレッドセーフですが、EspMessageWriter と EspRelativeRowWriter はスレッドセーフではありません。そのため、EspMessageWriter と EspRelativeRowWriter へのアクセスを同期するようにしてください。

SDK には、パブリッシャの動作を調整するさまざまなオプションがあります。EspPublisher を作成する場合は、EspPublisherOption を使用してこれらのオプションを指定します。作成後は、オプションは変更できません。パブリッシュでは、SDK のその他のエンティティと同様に、直接、コールバック、セレクトの各アクセス・モードも使用できます。

SDK ではアクセス・モードを利用できるほか、内部バッファリング機能がサポートされます。パブリッシュがバッファに保存されると、データはまず内部キューに書き込まれます。 このデータはパブリッシュ・スレッドで選択されて、プラットフォームに書き込まれます。バッファリング機能は、直接アクセス・モードの場合のみ利用できます。 バッファに保存されたパブリッシュで、直接アクセス・モードを使用することにより、最も効率的なスループットが得られる可能性があります。

その他に、バッチ・モードと同期モードの設定がパブリッシュに影響を及ぼします。バッチ・モードは、データ・ローがソケットに書き込まれる方法を指定します。 データ・ローは、個別に書き込むことも、エンベロープまたはトランザクション・バッチでクループ化することもできます。エンベロープは、個別のローをグループ化して、プラットフォームに書き込まれます。また、プラットフォームによってソケットから一括で読み取られます。これにより、ネットワーク・スループットが向上します。トランザクション・バッチは、エンベロープ・バッチと同様に、データをグループ化して、書き込みと読み取りが行われます。ただし、

トランザクション・バッチの場合、バッチ内のすべてのローの処理が完了した場合のみ、プラットフォームでグループが処理されます。ローの処理に失敗すると、バッチ全体がロールバックされます。

注意： シャインスルーを使用して、更新レコード内のデータが NULL である場合に、以前のデータを保持するようにするには、トランザクション・バッチを使用せず、個別にまたはエンベロープを使用してローをパブリッシュします。

同期モードの設定では、SDK とプラットフォームの間のパブリッシュ・ハンドシェイクを制御できます。 SDK は、デフォルトでは、データの受信確認なしでプラットフォームにデータを送信し続けます。 ただし、同期モードが true に設定されている場合は、SDK はプラットフォームからの受信確認を待ってから、データの次のバッチを送信します。この場合は、アプリケーション・レベルの配信は保証されますが、スループットが低下します。

コールバック・モードまたはセレクト・モードでパブリッシュを実行する場合、注意すべき点があります。これらのモードは、ESP_PUBLISHER_EVENT_READY イベントによって実行されます。このイベントは、パブリッシャで、より多くのデータを受信する準備ができていることを示します。 これによって、ユーザはデータのパブリッシュまたはコミットの発行を行えますが、1 つの ESP_PUBLISHER_EVENT_READY イベントに対して実行できるアクションは 1 つのみです。

同期モードでのパブリッシュの場合は、スループットは向上しますが、アプリケーション・レベルの配信は保証されません。TCP でもアプリケーション・レベルの配信は保証されないので、クライアントが終了すると、TCP バッファ内のデータは失われる可能性があります。そのため、同期モードでパブリッシュする場合は、クライアントが終了する前にコミットを実行します。

一般的には、パブリッシュ・コールからのリターン・コードは、ローの送信が成功したかどうかを示します。プラットフォームでの処理中に発生したエラー (重複データの挿入など) は返されません。 パブリッシュ・コールからのリターン・コードの意味は、アクセス・モードと、同期転送または非同期転送のどちらを選択したかによって異なります。

コールバック・アクセス・モードまたはセレクト・アクセス・モードを使用する場合、リターン・コードでは、SDK でデータ・キューを作成できるかどうかのみが示されます。データがソケットに実際に書き込まれたかどうかは、該当のイベントによって示されます。コールバック・アクセス・モードおよびセレクト・アクセス・モードでは、現在同期パブリッシュは実行できません。

直接アクセス・モードを使用している場合、使用されている転送の種類によってパブリッシュ・コールからのリターン・コードの内容が決定します。パブリッシュが非同期モードで実行されている場合、リターン・コードでは、データがソケットに書き込まれたことのみが示されます。 パブリッシュが同期モードで実行

されている場合、パブリッシュ・コールからのリターン・コードでは、プラットフォームから送信された応答コードが示されます。

パブリッシュ・コールでは、プラットフォームでの処理中に発生したエラー (重複データの挿入など) は返されません。

すべてのエンティティと同様に、コールバック・モードでパブリッシャを実行し、受信通知が必要な場合、イベントのトリガの前にコールバック・ハンドラを登録します。次に例を示します。

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

次のコードは、データをパブリッシュするさまざまな方法を示します。

1 つ目の例は、直接アクセス・モードでトランザクション・ブロックを使用してパブリッシュする方法を示します。

```
    publisher = esp_project_create_publisher(project, NULL, error);
                  // create publisher with default options from an
existing EspProject
    int rc = esp_publisher_connect(publisher, error);
                 // connect the publisher
    const EspStream * stream = esp_project_get_stream(project,
"Stream1", error);
                // retrieve EspStream we want to publish to
    const EspSchema * schema = esp_stream_get_schema(stream, error);
                // determine its schema
    EspMessageWriter * writer = esp_publisher_get_writer(publisher,
stream, error);
                // create EspMessageWriter to publish to "Stream1"
    EspRelativeRowWriter * row_writer =
esp_message_writer_get_relative_rowwriter(writer, error);


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


    int32_t intvalue = 10;
    bool inblock = false;


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

```
        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
is needed
                default:
                 esp_relative_rowwriter_set_null(row_writer, error);
                    break;
            }
        }
        esp_relative_rowwriter_end_row(row_writer,
error);                  // end the data row


        if ((nrows % 60) == 0) {
                    // determine if the batch is to be ended, we code
for 60 rows per block
            esp_message_writer_end_block(writer, error);
                        // end the batch started in
esp_message_writer_start_transaction()
            esp_publisher_publish(publisher, writer,
error);               // publish the batch
            inblock = false;
        }
    }
   esp_publisher_close(publisher, error);                        //
done with publishing
```

この例は、コールバック・アクセス・モードでパブリッシュする方法を示します。

```
    int rc;
    EspPublisherOptions * options =
esp_publisher_options_create(error);
                // create EspPublisherOptions
    rc = esp_publisher_options_set_access_mode(options,
CALLBACK_ACCESS, error);
                // set access mode
    publisher = esp_project_create_publisher(project, options,
error);
               // create EspPublisher using the options above from
existing EspProject
   esp_publisher_options_free(options, error);                   //
free EspPublisherOptions
    rc = esp_publisher_set_callback(publisher,
ESP_PUBLISHER_EVENT_ALL, publish_callback,
    NULL, m_error);   // set callback handler
```

```
   rc = esp_publisher_connect(publisher, error);                //
connect publisher


    ...
    ...
    ...


    // 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);


    }
```

サブスクライブ

SDK には、プロジェクトをサブスクライブするためのさまざまなオプションがあります。

SDK を使用してデータをサブスクライブする手順は、次のとおりです。

1. EspSubscriber オブジェクトを作成します。このオブジェクトは、直接作成するか、EspProject から取得できます。
2. EspSubscriber に接続します。
3. ストリームをサブスクライブします。
4. 直接アクセス・モードでは、**esp_subscriber_get_next_event()** を使用してイベントを取得します。コールバック・アクセス・モードとセレクト・アクセス・モードでは、SDK によってイベントが生成されて、ユーザ・コードに再び渡されます。
5. データ・イベントの場合は、EspMessageReader を取得します。これにより、プラットフォームからの 1 つのメッセージがカプセル化されます。カプセルには、単一のデータ・ローか、複数のデータ・ローから成るトランザクション／エンベロープ・ブロックが含まれている場合があります。
6. EspRowReader を 1 つ以上取得します。 EspRowReader のメソッドを使用して、個々のフィールドのデータを読み取ります。

次の例は、デフォルト・オプションで直接アクセス・モードを使用してストリームをサブスクライブする方法を示します。

```
EspError * error = esp_error_create();
esp_sdk_start(error);
EspUri * project_uri = esp_uri_create_string("esp://server:port//
default/vwap", error);
EspProject * project = esp_project_create(project_uri, NULL, NULL,
error);
rc = esp_project_connect(project, error);
EspSubscriber * subscriber = esp_project_create_subscriber(project,
NULL, 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();
```

イベントが ESP_SUBSCRIBER_EVENT_DATA イベントの場合は、フィールド・データがあります。これは、サブスクライブ・イベントからデータを読み取る一般的な例です。

```
   const EspStream * stream = esp_subscriber_event_get_stream(event,
error);
          // stream for this event
    EspMessageReader * reader =
esp_subscriber_event_get_reader(event, error);
          // get message reader
    int rc = esp_message_reader_is_block(reader, &flag,
error);
          // you can check if this a block
    const EspSchema * schema = esp_stream_get_schema(stream,
error);
          // get the stream schema if you do not have it
    EspRowReader * row_reader;


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


    while ((row_reader = esp_message_reader_next_row(reader,
error)) != NULL) {
            // loop until we finish all rows
         for (int i = 0; i < numcolumns; ++i) {
            rc = esp_row_reader_is_null(row_reader, i, &flag,
error);
            // if column is null, skip
            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
                // ...
            }
        }
    }
```

フェールオーバの処理

SDK では、さまざまな状況で、完全に透過的なフェールオーバまたは自動のフェールオーバがサポートされます。

- **クラスタのフェールオーバ** – バックエンド・コンポーネントとの接続に使用される URI には、クラスタ・マネージャの指定内容のリストを含めることができます。SDK によって、これらへの接続は透過的に維持されます。 したがって、クラスタ・マネージャのいずれか 1 つが停止した場合、SDK は別のインスタンスへの接続を試みます。 既知のすべてのインスタンスへの接続に失敗すると、SDK はエラーを返します。コールバック・アクセス・モードまたはセレクト・アクセス・モードで実行している場合、SDK を構成して、接続が切断されるまでの許容範囲のレベルを 1 つ上げることができます。この場合、既知のすべてのマネージャ・インスタンスが停止しても、EspServer インスタンスは切断されず、ESP_SERVER_EVENT_STALE イベントが生成されます。一定の回数 (指定可能) の試行の後、再接続できた場合、ESP_SERVER_EVENT_UPTODATE が生成されます。それ以外の場合は、接続は切断され、ESP_SERVER_EVENT_DISCONNECTED イベントが生成されます。
- **プロジェクトのフェールオーバ** – Event Stream Processor クラスタでは、プロジェクトにフェールオーバを設定できます。構成設定により、クラスタでは、プロジェクトが終了したことが検出されると、プロジェクトが再開されます (ただし、ユーザが明示的にプロジェクトを終了した場合は、プロジェクトは再開されません)。この機能に対応するため、EspProject インスタンスを使用してクラスタでプロジェクトの再開と再接続を監視できます。この機能は、コールバック・モードまたはセレクト・モードの場合にのみ使用できます。プロジェクトが停止したことを SDK が検出すると、ESP_PROJECT_EVENT_STALE イベントが生成されます。再接続できた場合、ESP_PROJECT_EVENT_UPTODATE イベントが生成されます。それ以外の場合は、ESP_PROJECT_EVENT_DISCONNECTED イベントが生成されます。
- **アクティブ／アクティブの配備** – プロジェクトをアクティブ／アクティブ・モードで配備することができます。このモードでは、クラスタは 1 つのプロジェクトで、プライマリ・インスタンスとセカンダリ・インスタンスの 2 つのインスタンスを開始できます。プライマリ・インスタンスにパブリッシュされたデータは、セカンダリ・インスタンスに自動的にミラーリングされます。SDK はこのようなアクティブ／アクティブの配備をサポートしています。アクティブ／アクティブで配備されていると、現在接続されているインスタンスが停止した場合に、EspProject は別のインスタンスへの接続を試みます。フェールオーバとは異なり、これは透過的に行われます。そのため、再接続で

きた場合でも、それは明示的に示されません。EspProject のほかに、パブリッシュとサブスクライブでもこのモードはサポートされています。アクティブ／アクティブで配備されたプロジェクトにサブスクライブしている場合は、インスタンスが停止しても SDK はサブスクリプションを切断せず、ESP_SUBSCRIBER_EVENT_DATA_LOST イベントが生成されます。その後、ピア・インスタンスへの再接続が試行されます。再接続できた場合、同じストリームを再度サブスクライブします。次に、サブスクライブ・クライアントは、ESP_SUBSCRIBER_EVENT_SYNC_START イベントに続けて、データ・イベントを受け取り、最後に ESP_SUBSCRIBER_EVENT_SYNC_END イベントを受け取ります。クライアントはこのシーケンスを使用して、必要に応じて、データ面での一貫性を維持できます。パブリッシュ中に再接続もできますが、この機能は同期モードでのパブリッシュでのみ利用できます。同期モード以外の場合、SDK では、データの一貫性を保証できません。パブリッシュ中の再接続は、透過的に行われ、外部のユーザ・イベントは生成されません。

API リファレンス

メソッド、関数、その他のプログラミング・ビルディング・ブロックの詳細については、API のマニュアルをダウンロードして参照してください。

C API のマニュアルをダウンロードして、ローカル・マシンにインストールしてください。

ダウンロードが完了したら、次の手順に従ってください。

1. ローカル・マシンのロケーションを指定し、そこにファイルを抽出します。
2. ファイルを抽出したロケーションを参照します。
3. `index.html` を開いて、API マニュアルを参照します。

索引

あ

アクセス・モード
- コールバック 3
- セレクト 3
- 直接 3

く

クラスの詳細 21

こ

コールバック・アクセス・モード
- 例 9

さ

サブスクライブ
- ストリーム 15
- 概要 15
- 直接モード 15
- 例 15

は

パブリッシュ
- スループットの向上 9
- プロジェクト 9
- モード 9
- 例 9

パブリッシュのモード
- バッチ・モード 9
- 同期モード 9

ふ

フェールオーバ
- アクティブ／アクティブ 19
- クラスタ 19
- プロジェクト 19

プロジェクト
- パブリッシュ 9

め

メソッドの詳細 21

り

リファレンス
- クラス 21
- メソッド 21
- 関数 21