SAS® Event Stream Processing 5.2: パブリッシュ/サブスクライブ API

パブリッシュ/サブスクライブ API の概要

SAS Event Stream Processing では、C、Java および Python 用のパブリッシュ/サブスクライブアプリケーションプログラミングインターフェイス(API)が提供されます。これらの API を使用して、以下を実行します。

- 実行中のイベントストリームプロセッサプロジェクトのソースウィンドウにイベントストリームをパブリッシュする
- 同じマシンまたはネットワーク上の別のマシンからプロジェクトのウィンドウイベントストリームをサブスクライブする

これらの API は、TCP/IP ネットワーキングをサポートしています。したがって、パブリッシュおよびサブスクライブ・アプリケーションは、イベントストリーム処理エンジンまたはエンジン・プラットフォーム自体へのネットワーク・アクセスを持つ任意のマシン上で実行できます。これらの API は、SAS Event Stream Processing でサポートされているすべてのアーキテクチャで 使用できます。

パブリッシュ/サブスクライブ API は、クロスプラットフォームでの使用を処理します。たとえば、バイトオーダーのエンディアンが異なる場合でも、 Java API を使用して Linux 上で実行されるエンジンのイベントストリームにサブスクライブすることができます。

永続性のためにイベントストリームをデータベースに継続的にロードできるように、イベントストリームにサブスクライブすることもできます。この場合、イベントストリームプロセッサーのデータベース コネクタまたはアダプタを使用します。

注: API は、アプリケーションと他のネットワークアプリケーション、クライアント、データフィードとの間のプラットフォーム間の接続性とエンディアンの互換性を提供します。API は IPv4 に準拠しています。

C パブリッシュ/サブスクライブ API の使用

エンジンのパースペクティブからの C パブリッシュ/サブスクライブ API

C++モデリング API を使用してエンジンインスタンスのパブリッシュ/サブスクライブを有効にするには、次のように、dfESPengine::initialize()コールの pubsub_ENABLE()パラメータにポート番号を指定する必要があります。

```
dfESPengine *engine;
engine = dfESPengine::initialize(argc, argv, "engine",
pubsub_ENABLE(33335));

if (engine == NULL) {
    cerr <<"Error: dfESPengine::initialize() failed\n";
    return 1;
}
```

注: ESP サーバーでは、コマンドラインで"–pubsub port"を指定するか、エンジンエレメントのポート属性を使用してポートを指定します。

注: パブリッシュ/サブスクライブポートを 1 つしか使用できない場合は、プロジェクトパブリッシュ/サブスクライブポートとして設定します。エンジンパブリッシュ/サブスクライブポートとして構成しないでください。

クライアントは、ポート番号(この例では 33335)を使用してパブリッシュ/サブスクライブ接続を確立できます。パブリッシュ/サブスクライブが不要な場合は、そのパラメータに pubsub_DISABLE を使用します。

パブリッシュ/サブスクライブが必要で、クライアントを認証する必要がある場合は、次のいずれかを使用します。

- pubsub_ENABLE_OAUTH(port, clientId) (ここで、*clientId* は CF UAA OAuth サーバークライアント ID です)
- pubsub_ENABLE_SASLOGONOAUTH(port, sasLogonURL) (ここで、*sasLogonURL* は SASLogon サービスの基本サービス URL です)
- pubsub_ENABLE_KERBEROS(port, serviceName) (ここで、*serviceName* は Kerberos サービスプリンシパル名のサービス名部分です。)

ESP サーバーの場合は、**-auth oauth://*clientId*** または**-auth saslogon://*sasLogonURL*** または**-auth kerberos://*serviceName*** を指定します。

詳細は、"TCP/IP 接続での認証の有効化" (*SAS Event Stream Processing: セキュリティ*)を参照してください。

プロジェクトのパブリッシュ/サブスクライブ機能を初期化するには、engine->startProjects()を呼び出す前に project->setPubSub()を呼び出します。

例えば:

```
project->setPubSub(dfESPproject::ps_AUTO);
engine->startProjects();
```

注: ESP サーバーと同等の機能は、 **プロジェクト**エレメントの **pubsub** 属性を使用してパブリッシュ/サブスクライブ機能を指定することです。

このコードは、ポート 33335 でサーバーリスナーソケットを開き、クライアントのサブスクライバとパブリッシャがパブリッシュ/サブスクライブサービス用にエンジンアプリケーションまたはサーバーに接続できるようにします。クライアントからのパブリッシュ/サブスクライブの接続要求が(後述のように)行われた後、パブリッシュ/サブスクライブ API がこの接続に使用する一時的なポートが返されます。

特定のポートの一時ポートを(セキュリティ上の目的で)オーバーライドするには、このプロジェクトへの接続に使用する優先ポートである第 2 パラメータを使用して project->setPubSub を指定します。

例えば:

```
project->setPubSub(dfESPproject::ps_AUTO, 33444);
```

注: ESP サーバーと同等の機能は、**プロジェクト**エレメントの**ポート**属性を使用してポートを指定することです。

project->setPubSub()の最初のパラメータはサブスクリプションサービスにのみ適用され、プロジェクトのウィンドウがクライアントのサブスクリプションをサポートする方法を指定します。ps_AUTO を指定すると、クライアントはプロジェクト内のすべてのウィンドウ出力イベントストリームをサブスクライブすることができます。

または、ps_MANUAL を指定して手動でウィンドウを有効にすることもできます。重要ではないプロジェクトでは、すべてのウィンドウを 自動的に有効にすることが全体のパフォーマンスに顕著な影響を与えるため、手動で任意のウィンドウを有効にします。ps_NONE を指定することもできます。これにより、すべてのウィンドウのサブスクライブを無効にします。

project->setPubSub()で ps_MANUAL を使用してウィンドウサブスクリプションを手動で有効にする場合は、それぞれのウィンドウに対して enableWindowSubs()を使用して、サブスクライブを次のように有効にします。

```
project->enableWindowSubs(dfESPwindow *w);
```

注: ESP サーバーは、**ウィンドウ**エレメントの **pubsub** 属性を使用して、パブリッシュ/サブスクライブを有効または無効にします。

ただし、setPubSub()で ps_AUTO または ps_NONE を指定した場合、その後の enableWindowSubs()の呼び出しは無視され、警告が生成されます。

注: クライアントは、現在実行中のプロジェクトの任意のソースウィンドウ(およびソースウィンドウのみ)にイベントストリームをパブリッシュできます。デフォルトでは、すべてのソースウィンドウがパブリッシュ 用に有効になっています。

クライアントの視点からの C パブリッシュ/サブスクライブ API

C パブリッシュ/サブスクライブ API を使用してエンジンのイベントストリームをサブスクライブしたりパブリッシュしたりするクライアントは、(C_dfESPpubsubInit()を使用して)クライアント上のサービスを最初に初期化する必要があります。次に、C_dfESPsubscriberStart()および C_dfESPpublisherStart()を使用しているパブリッシャを使用してサブスクリプションを開始し、次に C_dfESPpubsubConnect()を使用してアプリケーションまたはサーバーに接続する必要があります。

パブリッシャを実装するクライアントは、C_dfESPpublisherStart().に渡された URL で指定されたソースウィンドウにイベントブロックをパブリッシュするために、必要に応じて C_dfESPpublisherInject()を呼び出すことができます。

クライアントパブリッシュ/サブスクライブ API の詳細は次のとおりです。

- パブリッシャサービスとサブスクライバサービスを提供するには、クライアントアプリケーションにヘッダーファイル C_dfESPpubsubApi.h を含める必要があります。API 呼び出しに加えて、このファイルにはユーザー提供のコールバック関数のシグニチャも定義されています。その中には、サブスクライブされたイベントブロックハンドラーとパブリッシュ/サブスクライブエラーハンドラーの 2 つがあります。
- サブスクライブされたイベントブロックハンドラーは、サブスクライバクライアントによってのみ 使用されます。これは、アプリケーションまたはサーバーからの新しいイベントブロックが到着したときに呼び出されます。イベントブロックを処理した後、クライアントは C_dfESPeventblock_destroy()を呼び出して解放します。このユーザー定義のコールバックのシグネチャは、次のとおりです。"eb"は読み込んだイベントブロック、"schema"はクライアント処理のイベントのスキーマ、ctx はコール状態を含むオプションのコンテキストオブジェクトです。

  ```
  typedef void (*C_dfESPsubscriberCB_func)(C_dfESPeventblock eb,
    C_dfESPschema schema, void *ctx);
  ```

- 2 番目のコールバック関数 C_dfESPpubsubErrorCB_func()は、サブスクライバクライアントとパブリッシャクライアントの両方でオプションです。指定されている場合(つまりヌルでない場合)、クライアントサービス内の異常な切断のような異常なイベントが発生するたびに呼び出されます。これにより、クライアントはパ

ブリッシュ/サブスクライブ・サービスの ERROR を処理し、場合によってはリカバリできます。このコールバック関数のシグネチャは次のとおりです。

- failure は pubsubFail_APIFAIL、pubsubFail_THREADFAIL、または pubsubFail_SERVERDISCONNECT です。
- code は障害の特定のコードを提供します
- ctx は、コール状態を含むオプションのコンテキストオブジェクトです

```
typedef void (*C_dfESPpubsubErrorCB_func)(C_dfESPpubsubFailures
  failure, C_dfESPpubsubFailureCodes code);
```

- C_dfESPpubsubFailures および C_dfESPpubsubFailureCodes 列挙体は、C_dfESPpubsubFailures.h で定義されています。
- パブリッシャクライアントは、C_dfESPpublisherInject()API 関数を使用して、イベントブロックをアプリケーションまたはサーバーのソースウィンドウにパブリッシュします。イベントブロックは、C_dfESPpublisherStart()に渡された URL で指定された連続クエリおよびプロジェクトで実行されているソースウィンドウに挿入されます。クライアントは、各ウィンドウに対して C_dfESPpublisherStart()を 1 回呼び出してから、適切なクライアントオブジェクトを必要に応じて C_dfESPpublisherInject()に渡すことで、プロジェクト内の複数のウィンドウにイベントをパブリッシュできます。
- クライアントは、いつでも現在実行中のウィンドウ、連続したクエリ、およびプロジェクトをさまざまなレベルで検出するために、アプリケーションまたはサーバーにいつでも問い合わせることができます。この情報は、C_dfESPsubscriberStart()または C_dfESPpublisherStart()に渡す URL 文字列を作成するために後で使用される名前を表す文字列のリストの形式でクライアントに返されます。サポートされる照会のリストについては、関数の説明を参照してください。

C パブリッシュ/サブスクライブ API の関数

パブリッシュ/サブスクライブ API でクライアントのパブリッシュ/サブスクライブに提供される関数は、次のとおりです。シンプルな接続や、クライアントによる複数の接続やリカバリ処理を伴うより堅牢で複雑な接続に使用できます。

int C_dfESPpubsubInit(C_dfESPLoggingLevel *level*, const char **logConfigPath*)

パラメータ:

level
: ログレベル

logConfigPath
: ログ構成ファイルのフルパス名

戻り値:

1
: 成功

0
: 失敗 — エラーはログに書き出されます。

注: この関数は、クライアントのパブリッシャサービスとサブスクライバサービスを初期化し、C_dfESPpubsubSetPubsubLib()と C_dfESPpubsubSetTransportConfigFile()を除いて、他のクライアントコールを行う前に(1 回だけ)呼び出さなければなりません。

clientObjPtr C_dfESPpublisherStart(char *_serverURL_, C_dfESPpubsubErrorCB_func _errorCallbackFunction_, void *_ctx_)

パラメータ:

serverURL
宛先ホスト、ポート、プロジェクト、連続問合せ、およびウィンドウを表す文字列

serverURL 形式
"dfESP://host:port/project/contquery/window"

errorCallbackFunction
クライアントサービスの障害を処理するためのヌルまたはユーザー定義の関数ポインタ

ctx
この呼び出しに状態を渡すためのオプションのコンテキストポインタ

戻り値:

以下に説明するすべての API 関数に渡されるクライアントオブジェクトへのポインタ、または失敗(ログに書き込まれた ERROR)があった場合はヌル。

注: この関数は、特定のパブリッシャクライアント接続の接続パラメータを検証して保持します。

clientObjPtr C_dfESPGDpublisherStart()

パラメータ:

C_dfESPpublisherStart()と同じパラメータと戻り値。追加必須パラメーター: C_dfESPGDpublisherCB_func タイプの保証配信コールバック関数ポインター。追加必須パラメータ: このパブリッシャの保証された配信設定ファイルのファイル名。

clientObjPtr C_dfESPsubscriberStart(char *_serverURL_, C_dfESPsubscriberCB_func _callbackFunction_, C_dfESPpubsubErrorCB_ func _errorCallbackFunction_, void *_ctx_)

パラメータ:

serverURL
宛先ホスト、ポート、プロジェクト、連続クエリ、およびエンジン内のウィンドウを表す文字列。また、クライアントのスナップショット要件を指定します。クライアントが増分更新の前にウィンドウの現在のスナップショット状態を受け取ると、”true”になります。

クライアントの折り畳み要件の指定はオプションです。デフォルトでは false です。true の場合、UPDATE_BLOCK イベントは UPDATE イベントに変換され、サブスクライバの出力をパブリッシュ可能にします。

serverURL 形式
"dfESP://host:port/project/contquery/window?snapshot=true |false <?collapse=true | false><?rmretdel=true | false>”

callbackFunction
受信したイベントブロックを処理するためのユーザ定義関数へのポインタ。この関数は、C_dfESPeventblock_destroy()を呼び出してイベントブロックを解放する必要があります。

errorCallbackFunction
サブスクリプションサービスの失敗を処理するためのヌルまたはユーザー定義の関数ポインタ

ctx
この呼び出しの状態を解析するためのオプションのコンテキストポインタ

戻り値:

以下に説明するすべての API 関数に渡されるクライアントオブジェクトへのポインタ。失敗した場合はヌル(ログに書き込まれた ERROR)。

注: この関数は、特定のサブスクライバクライアント接続の接続パラメータを検証して保持します。

clientObjPtr C_dfESPGDsubscriberStart()

パラメータ:

C_dfESPsubscriberStart()と同じパラメータと戻り値。追加必須パラメータ: このサブスクライバの保証された配信設定ファイルのファイル名。

int C_dfESPpubsubConnect(clientObjPtr *client*)

パラメータ:

client
: C_dfESPsubscriberStart() または C_dfESPpublisherStart()または C_dfESPGDsubscriberStart()または C_dfESPGDpublisherStart()によって返されたクライアントオブジェクトへのポインタ。

戻り値:

1
: 成功

0
: 失敗 — エラーはログに書き出されます。

注: この関数は、アプリケーションまたはサーバーとの接続を確立しようとします。

int C_dfESPpubsubDisconnect(clientObjPtr *client*, int *block*)

パラメータ:

client
: C_dfESPsubscriberStart()または C_dfESPpublisherStart()または C_dfESPGDsubscriberStart()または C_dfESPGDpublisherStart()によって返されたクライアントオブジェクトへのポインタ。

block
: キューに入れられたすべてのイベントが処理されるのを待つ場合は 1 に設定し、それ以外の場合は 0 に設定します

戻り値:

1
: 成功

0
: 失敗 — エラーはログに書き出されます。

注: この関数は、渡されたクライアントオブジェクトに関連付けられている接続を閉じます。

int C_dfESPpubsubStop(clientObjPtr *client*, int *block*)

パラメータ:

client
: C_dfESPsubscriberStart() または C_dfESPpublisherStart()または C_dfESPGDsubscriberStart()または C_dfESPGDpublisherStart()によって返されたクライアントオブジェクトへのポインタ。

block
: キューに入れられたすべてのイベントが処理されるのを待つ場合は 1 に設定し、それ以外の場合は 0 に設定します

戻り値:

1
: 成功

0
: 失敗 — エラーはログに書き出されます。

int C_dfESPpubsubStop(clientObjPtr *client*, int *block*)

注: この関数は、クライアントセッションを停止し、渡されたクライアントオブジェクトを削除します。

int C_dfESPpublisherInject(clientObjPtr *client*, C_dfESPeventblock *eventBlock*)

パラメータ:

client
C_dfESPpublisherStart()または C_dfESPGDsubscriberStart()によって返されたクライアントオブジェクトへのポインタ

eventBlock
エンジンに注入するイベントブロック。ブロックは、渡されたクライアントオブジェクトに関連付けられたソースウィンドウ、連続クエリ、およびプロジェクトに挿入されます。

戻り値:

1
成功

0
失敗 — エラーはログに書き出されます。

注: この関数は、イベントをエンジンにパブリッシュすることによってクライアントパブリッシャ関数を実装します。イベント・ブロックは、イベントストリームプロセッサのオブジェクト C API で提供される他の追加機能を使用して構築することができます。

C_dfESPstringV C_dfESPpubsubQueryMeta(char **queryURL*)

パラメータ:

queryURL
エンジンに送信されるクエリを表す文字列。

戻り値:

クエリへのレスポンスを構成する名前のリストを表す文字列のベクトル、またはエラー(ログに書き込まれた ERROR)があった場合はヌルです。リストの終わりは空の文字列で示されます。呼び出し元は、C_dfESPstringV_free()を呼び出してベクトルを解放します。

注: この関数は、一般的なイベントストリームプロセッサのメタデータ問合せメカニズムを実装します。このメカニズムにより、クライアントは、現在エンジンで実行されているプロジェクト、連続クエリ、ウィンドウ、ウィンドウスキーマ、およびウィンドウのエッジを検出できます。このメカニズムは、クライアントによって実行される他のアクティビティとの 依存関係も相互作用もありません。この関数は、独立したソケットを開いてクエリを送信し、クエリ応答を受信するとソケットを閉じます。

サポートされている *queryURL* の形式	
"dfESP://host:port?get=projects"	現在実行中のプロジェクトの名前を返します。
“dfESP://host:port?get=projects_pubsubonly”	パブリッシュ/サブスクライブが有効になっている現在実行中のプロジェクトの名前を返します。
"dfESP://host:port?get=queries"	現在実行中のプロジェクトで連続クエリの名前を返します。

サポートされている *queryURL* の形式	
"dfESP://host:port/project?get=windows_sourceonly"	プロジェクトが実行中の場合、指定されたプロジェクトのソースウィンドウの名前を返します
"dfESP://host:port/project?get=windows_derivedonly"	プロジェクトが実行中の場合、指定されたプロジェクトの派生ウィンドウの名前を返します。
"dfESP://host:port?get=queries_pubsubonly"	現在実行中のプロジェクトでパブリッシュ/サブスクライブが可能なウィンドウを含む連続クエリーの名前を返します
"dfESP://host:port?get=windows"	現在実行中のプロジェクトのウィンドウの名前を返します。
"dfESP://host:port?get=windows_pubsubonly"	現在実行中のプロジェクトのパブリッシュ/サブスクライブ可能なウィンドウの名前を返します。
"dfESP://host:port?get=windowsandtypes"	現在実行中のプロジェクトのウィンドウの名前を返します。返される各ウィンドウ名の後にコロンとウィンドウタイプが続きます
"dfESP://host:port/project/contquery?get=windows_sourceonly"	プロジェクトが実行中の場合、指定された連続したクエリおよびプロジェクト内のソースウィンドウの名前を返します
"dfESP://host:port/project/contquery?get=windows_derivedonly"	プロジェクトが実行されている場合、指定された連続したクエリおよびプロジェクト内の派生ウィンドウの名前を返します
"dfESP://host:port/project?get=windows"	実行中の場合、指定されたプロジェクト内のウィンドウの名前を返します。
"dfESP://host:port/project?get=windowsandtypes"	実行中の場合、指定されたプロジェクト内のウィンドウの名前を返します。返される各ウィンドウ名の後にコロンとウィンドウタイプが続きます
"dfESP://host:port/project?get=windows_pubsubonly"	実行中の場合、指定されたプロジェクトのパブリッシュ/サブスクライブ可能なウィンドウの名前を返します。

サポートされている *queryURL* の形式	
"dfESP://host:port/project?get=queries"	実行中の場合、指定されたプロジェクト内の連続するクエリの名前を返します
"dfESP://host:port/project?get=queries_pubsubonly"	実行中の場合、指定されたプロジェクトにパブリッシュ/サブスクライブ可能なウィンドウを含む連続クエリの名前を返します
"dfESP://host:port/project/contquery?get=windows"	実行中の場合、指定された連続したクエリおよびプロジェクト内のウィンドウの名前を返します
"dfESP://host:port/project/contquery?get=windowsandtypes"	実行中の場合は、指定された連続クエリおよびプロジェクト内のウィンドウの名前を返します。返される各ウィンドウ名の後にコロンとウィンドウタイプが続きます。
"dfESP://host:port/project/contquery?get=windows_pubsubonly"	実行中の場合、指定された連続したクエリおよびプロジェクト内のパブリッシュ/サブスクライブ可能なウィンドウの名前を返します
"dfESP://host:port/project/contquery/window?get=schema"	ウィンドウスキーマのシリアル化されたバージョンである単一の文字列を返します。
"dfESP://host:port/project/contquery/window?get=edges"	すべてのウィンドウのエッジの名前を返します
"dfESP://host:port/project/contquery/window?get=rowcount"	現在ウィンドウ内にある行の数を返します。

C_dfESPstringV C_dfESPpubsubGetModel(char *_queryURL_)

パラメータ:

queryURL
エンジンに送信されるクエリを表す文字列。

queryURL のサポートされる形式は次のとおりです。

- "dfESP://host:port" - モデル内のすべてのウィンドウの名前とそのエッジを返します
- " dfESP://host:port/project " - プロジェクト内のすべてのウィンドウの名前とそのエッジを返します
- " dfESP://host:port/project/contquery " - 連続クエリ内のすべてのウィンドウの名前とそのエッジを返します

C_dfESPstringV C_dfESPpubsubGetModel(char *_queryURL_)

戻り値: クエリへの応答を表す文字列のベクトル。失敗があった場合はヌル(ログに書き込まれたERROR)。各文字列の形式は "*project*/*query*/*window*:*edge1*, *edge2*, ..."です。リストの終わりは空の文字列で示されます。呼び出し元は、C_dfESPstringV_free()を呼び出してベクトルを解放します。

注: この関数を使用すると、モデルまたはプロジェクトまたは連続クエリのウィンドウの完全なセットをウインドウのエッジと共に返すことによって、クライアントがエンジンを検出できます。クライアントによって実行される他のアクティビティとの 依存関係や相互作用はありません。クエリを送信するために独立したソケットを開き、クエリ応答を受信するとソケットを閉じます。

void C_dfESPpubsubShutdown()

パブリッシュ/サブスクライブサービスをシャットダウン

int C_dfESPpubsubPersistModel(char *_hostportURL_、const char *_persistPath_)

パラメータ:

hostportURL
"dfESP://host:port"という形式の文字列

persistpath
ターゲットプラットフォーム上の持続ファイルの絶対パス名または相対パス名

戻り値:

1
成功

0
失敗 — エラーはログに書き出されます。

注: この関数は、*hostportURL* のエンジンに現在の状態をディスクに保持するよう指示します。クライアントによって実行される他のアクティビティとの 依存関係や相互作用はありません。要求を送信するために独立したソケットを開き、要求戻りコードを受け取るとソケットを閉じます。

int C_dfESPpubsubQuiesceProject(char *_projectURL_、clientObjPtr _client_)

パラメータ:

projectURL
"dfESP://*ホスト*:*ポート*/*プロジェクト*"という形式の文字列

client
C_dfESPsubscriberStart()または C_dfESPpublisherStart()または C_dfESPGDsubscriberStart()または C_dfESPGDpublisherStart()によって返されたクライアントオブジェクトへのオプションのポインタ。ヌルでない場合は、指定したクライアントのキューに入れられたイベントが処理されてから、プロジェクトを静止します。

戻り値:

1
成功

0
失敗 - ログに書き込まれた ERROR。

注: この関数は、projectURL のエンジンに projectURL でプロジェクトを休止するよう指示します。この呼び出しは同期的です。つまり、プロジェクトが終了したときに戻ります。

int C_dfESPsubscriberMaxQueueSize(char *_serverURL_, int _maxSize_, int _block_)

パラメータ:

serverURL
次のいずれかの形式のサーバー URL の文字列。

- **"dfESP://*host*:*port*"**
- **"dfESP://*host*:*port*/*project*"**
- **"dfESP://*host*:*port*/*project*/*contquery*"**
- **"dfESP://*host*:*port*/*project*/*contquery*/*window*"**

maxsize
serverURL 内のウィンドウへの単一のサブスクライバのためにキューに入れることができるイベントブロックの最大数

block
キューサイズが *maxSize* を下回るのを待つ場合は 1 に設定し、そうでない場合はクライアントを切断します

戻り値:

1
成功

0
失敗 - ログに書き込まれた ERROR。

注: この関数を使用して、プロジェクト、クエリ、またはウィンドウ内のサブスクライバに送信されるイベントブロックをエンキューするために使用されるすべてのキューの最大サイズを構成します。これらのキューで消費されるメモリ量を制限するには、この値を使用します。block パラメータは、最大値に達したときの動作を指定します。

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

パラメータ:

psLib
クライアント/サーバーのトランスポートを表す番号

psLib のサポートされる値は次のとおりです。

- ESP_PSLIB_NATIVE (デフォルト)
- ESP_PSLIB_SOLACE - このモードでは、アプライアンス接続パラメータを提供するために、solace.cfg という名前のクライアント構成ファイルがカレントディレクトリに存在する必要があります。または、C_dfESPpubsubSetTransportConfigFile()を呼び出して、クライアント構成ファイルを指定します。
- ESP_PSLIB_TERVELA - このモードでは、アプライアンス接続パラメータを提供するために、**client.config** という名前のクライアント構成ファイルがカレントディレクトリに存在する必要があります。または、C_dfESPpubsubSetTransportConfigFile()を呼び出して、クライアント構成ファイルを指定します。
- ESP_PSLIB_RABBITMQ - このモードでは、Rabbit MQ サーバー接続パラメータを提供するために、**rabbitmq.cfg** という名前のクライアント構成ファイルがカレントディレクトリに存在する必要があります。または、C_dfESPpubsubSetTransportConfigFile()を呼び出して、クライアント構成ファイルを指定します。
- ESP_PSLIB_KAFKA - Kafka クラスタ接続パラメータを提供するために、**kafka.cfg** という名前のクライアント構成ファイルが現在のディレクトリに存在する必要があります。または、C_dfESPpubsubSetTransportConfigFile()を呼び出して、クライアント構成ファイルを指定します。

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Solace 設定ファイル形式

```
solace
{
SESSION_HOST = "10.37.150.244:55555"
SESSION_USERNAME = "pub1"
SESSION_PASSWORD = "pub1"
SESSION_VPN_NAME = "SAS"
SESSION_RECONNECT_RETRIES = "3"
SESSION_REAPPLY_SUBSCRIPTIONS = true
SESSION_TOPIC_DISPATCH = true
}
sas
{
buspersistence = false
queuename = "myqueue"
protobuf = false
protofile = "./GpbHistSimFactory.proto"
protomsg = "GbpTrade"
json = false
dateformat = "%Y-%m-%d %H:%M:%S"
passwordencrypted = false
}
```

注: passwordencrypted = true の場合、**SESSION_PASSWORD** の値は、この OpenSSL コマンドで生成された暗号化されたパスワードにする必要があります。echo "SESSION_PASSWORD" | openssl enc -e -aes-256-cbc -a -salt -pass pass:"SASespSOLclientUsedByUser=SESSION_USERNAME"

Tervela 設定ファイル形式

```
USERNAME          esp
PASSWORD          esp
PRIMARY_TMX       10.37.8.175
LOGIN_TIMEOUT     45000
GD_CONTEXT_NAME   tvaIF
GD_MAX_OUT        10000
PASSWORDENCRYPTED      0
```

注: PASSWORDENCRYPTED = 1 の場合、**PASSWORD** の値は、この OpenSSL コマンドで生成された暗号化されたパスワードにする必要があります。echo "PASSWORD" | openssl enc -e -aes-256-cbc -a -salt -pass pass:"SASespTVAclientUsedByUser=USERNAME"

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

RabbitMQ 設定ファイル形式

```
rabbitmq
{
host = "my.machine.com"
port = "5672"
exchange = "SAS"
userid = "guest"
password = "guest"
passwordencrypted = false
ssl = false
sslcacert = "./mycacert.pem"
sslkey = "./mykey.pem"
sslcert = "./mycert.pem"
}

sas
{
buspersistence = false
queuename = "subpersist"
protobuf = false
protofile = "./GpbHistSimFactory.proto"
protomsg = "GpbTrade"
json = false
noreplay = false
noautoack = false
dateformat = "%Y-%m-%d %H:%M:%S"
}
```

注: passwordencrypted = true の場合、password の値は、この OpenSSL コマンドで生成された暗号化されたパスワードにする必要があります。echo "password" | openssl enc -e -aes-256-cbc -a -salt -pass pass:"SASespRMQclientUsedByUser=userid"

注: buspersistence と queuename パラメータは、Rabbit Mq メッセージのサブスクライバまたはパブリッシャにとって異なる意味を持ちます。パブリッシャの場合、queuename は常に無視されます。If buspersistence = false の場合、メッセージは非永続配信モードで送信されます。それ以外の場合、配信モードは永続的です。サブスクライバの場合は、受信キューを作成するときに常に queuename が使用されます。buspersistence = false の場合、クライアントによって 作成されたすべてのキューおよびエクスチェンジは永続性がなく、自動的に削除されます。buspersistence = true の場合、すべての 交換およびキューは永続性があり、自動削除ではありません。デフォルトでは、noreplay パラメータは false です。true に設定すると、Rabbit MQ から受信したメッセージは、buspersistence が有効になっていても 肯定応答されます。デフォルトでは、noautoack パラメータは **false** に設定されています。**true** に設定すると、Rabbit MQ から受信したメッセージは、Rabbit MQ の autoack を通じて暗黙的に確認されるのではなく、明示的に確認されます。これは、受信したメッセージ処理で検出されたすべての ERROR が ack を抑制し、メッセージを Rabbit MQ キューに残すことを意味します。

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib *psLib*)

Kafka 設定ファイル形式

```
kafka
{
hostport = "kafkahost:9092"
partition = "0"
initialoffset = "largest"
groupid = "mygroup"
globalconfig = <librdkafka global config>
topicconfig = <librdkafka topic config>
metatopic= "mymetatopic"
}

sas
{
protobuf = false
protofile = "./GpbHistSimFactory.proto"
protomsg = "GpbTrade"
json = false
dateformat = "%Y-%m-%d %H:%M:%S"
hotfailover = "false"
numbufferedmsgs = "16000"
zookeeperhostport = "myhost:myport"
failovergroup = "mygroup"
}
```

戻り値:

1
: 成功

0
: 失敗

注: この関数呼び出しはオプションですが、呼び出された場合は C_dfESPpubsubInit()を呼び出す前に呼び出さなければなりません。ESP パブリッシュ/サブスクライブプロトコルを使用するデフォルトのピアツーピア TCP/IP ベースのソケット接続から、クライアントとエンジンの間で使用されるトランスポートを変更します。代わりに、ESP_PSLIB_SOLACE、ESP_PSLIB_TERVELA、ESP_PSLIB_RABBITMQ、または ESP_PSLIB_KAFKA を指定して、クライアントの TCP/IP ピアが Solace アプライアンス、Tervela アプライアンス、Rabbit MQ サーバ、または Kafka クラスタであることを示すことができます。このモードでは、エンジンが Solace、Tervela、Rabbit MQ、または Kafka コネクタを実行して、対応する逆クライアントをアプライアンスに提供する必要があります。アプライアンスで使用されるトピック名は、パブリッシュ/サブスクライブクライアントとコネクタによって 調整され、アプライアンスを介してイベントブロックを正しくルーティングします。

注: Solace、Tervela、Rabbit MQ、または Kafka トランスポートを使用する場合、次のパブリッシュ/サブスクライブ API 関数はサポートされていません。

C_dfESPpubsubGetModel()
C_dfESPGDpublisherStart()
C_dfESPGDpublisherGetID()
C_dfESPGDsubscriberStart()
C_dfESPGDsubscriberAck()
C_dfESPpubsubSetBufferSize()
C_dfESPsubscriberMaxQueueSize()
C_dfESPpubsubPingHostPort()

C_dfESPpubsubSetTransportConfigFile(const char **transportCfgFile*)

パラメータ:

transportCfgFile
: トランスポート構成ファイルのフルパスを指定します。

C_dfESPpubsubSetTransportConfigFile(const char *_transportCfgFile_)

戻り値:

1
成功

0
失敗

C_dfESPGDsubscriberAck(clientObjPtr _client_, CdfESPeventblock _eventblock_)

パラメータ:

- C_dfESPGDsubscriberStart()によって返されたクライアントオブジェクトへの *client* ポインタ
- *eventBlock* は、パブリッシャに返信されるイベントブロックへのポインタです。この関数が戻る前に、イベントブロックを解放してはいけません。

戻り値:

- 1 = success
- 0 = 失敗

C_dfESPGDpublisherCB_func()

パラメータ:

C_dfESPGDpublisherStart()に渡される保証された配信パブリッシャコールバック関数のシグネチャ。この関数は、発行されたイベントブロックの保証された配信ステータスをパブリッシャに返すために API によって呼び出されます。

- パラメータ 1: READY または ACK または NACK (承認済みまたは未承認)。
- パラメータ 2: 64 ビットイベントブロック ID
- パラメータ 3: C_dfESPGDpublisherStart()に渡されたユーザコンテキストポインタ

戻り値:

空

C_dfESPGDpublisherGetID()

戻り値:

64 ビットのイベントブロック ID。イベントブロックに書き込まれる順番に固有の ID を取得するためにパブリッシャから呼び出され、保証された配信対応のパブリッシュクライアントにそれらを注入することがあります。

int C_dfESPpubsubSetBufferSize(clientObjPtr _client_, int32_t _mbytes_)

パラメータ:

client
C_dfESPsubscriberStart()、C_dfESPpublisherStart() 、C_dfESPGDsubscriberStart()、または C_dfESPGDpublisherStart()によって返されたクライアントオブジェクトへのポインタ。

mbytes
1MB 単位の読み書きバッファサイズ

戻り値:

1
成功

0
失敗

int C_dfESPpubsubSetBufferSize(clientObjPtr *client*, int32_t *mbytes*)

注: この関数呼び出しはオプションですが、呼び出された場合は C_dfESPsubscriberStart()、C_dfESPpublisherStart()、C_dfESPGDsubscriberStart()、または C_dfESPGDpublisherStart()の後と C_dfESPpubsubConnect()の前に呼び出さなければなりません。ソケットの読み取りおよび書き込み操作に使用されるバッファのサイズを変更します。デフォルトでは、このサイズは 16MB です。

int C_dfESPpubsubPingHostPort(char **serverURL*)

パラメータ:

serverURL
"dfESP://host:port*host:port*"という形式の文字列

戻り値:

1
ポートはパブリッシュされており、パブリッシュ/サブスクライブポートです

0
ポートが開いていないか、パブリッシュ/サブスクライブポートではありません

注: この関数は、実行中のエンジンに ping を実行して、指定されたポートが開いているかどうかを判断します。また、パブリッシュポートがパブリッシュ/サブスクライブポートであることを確認するために、マジックナンバーを交換して検証します。

int C_dfESPpubsubSetTokenLocation(char **tokenLocation*)

パラメータ:

tokenLocation
トークンを含むファイルの絶対パスと filename

戻り値:

1
成功

0
失敗

注: この関数は、パブリッシュ/サブスクライブサーバーによる認証に必要な OAuth トークンを含むローカルファイルシステム内のファイルの場所を設定します。

protobuffObjPtr C_dfESPpubsubInitProtobuff(char **protoFile*, char **msgName*, C_dfESPschema *C_schema*, char **dateFormat*, C_dfESPeventcodes *defaultOpcode*)

パラメータ:

protoFile
メッセージ定義を含む Google の.proto ファイルへのパス。

msgName
Google .proto ファイル内のターゲットメッセージ定義の名前。

C_schema
ウィンドウスキーマへのポインタ。

dateFormat
CSV コンバージョンの日付形式。ESP_DATETIME および ESP_TIMESTAMP フィールド値を、エポック以降の秒数(ESP_DATETIME)またはマイクロ秒(ESP_TIMESTAMP)として解釈するには、ヌルに設定します。

defaultOpcode
シリアライズされた protobuf から構築されたイベントに挿入する演算コード。

protobuffObjPtr C_dfESPpubsubInitProtobuff(char *_protoFile_, char *_msgName_, C_dfESPschema _C_schema_, char *_dateFormat_, C_dfESPeventcodes _defaultOpcode_)

戻り値: protobuff オブジェクトへのポインタで、他のすべての protobuff 関数に渡されます。失敗した場合はヌルを返します。

C_dfESPeventblock C_dfESPprotobuffToEb(protobuffObjPtr _protobuff_, void *_serializedProtobuff_)

パラメータ:

protobuff
C_dfESPpubsubInitProtobuff()によって作成されたオブジェクトへのポインタ

serializedProtobuff
ソケット上で受信されたシリアル化された protobuff へのポインタ

戻り値: イベントブロックポインタ

void *C_dfESPebToProtobuff(protobuffObjPtr _protobuff_, C_dfESPeventblock _C_eb_, int32_t _index_)

パラメータ:

protobuff
C_dfESPpubsubInitProtobuff()によって作成されたオブジェクトへのポインタ

C_eb
イベントブロックポインタ

index
イベントブロックで変換するイベントのインデックス

戻り値: protobuff をシリアライズするポインタ

void C_dfESPdestroyProtobuff(protobuffObjPtr _protobuff_, void *_serializedProtobuff_)

パラメータ:

protobuff
C_dfESPpubsubInitProtobuff()によって作成されたオブジェクトへのポインタ

serializedProtobuff
ソケット上で受信されたシリアル化された protobuff へのポインタ

jsonObjPtr C_dfESPpubsubInitJson(C_dfESPschema _C_schema_, char *_dateFormat_ C_dfESPeventcodes _defaultOpcode_)

パラメータ:

C_schema
ウィンドウスキーマへのポインタ

dateFormat
CSV コンバージョンの日付形式。ESP_DATETIME および ESP_TIMESTAMP フィールド値を、エポック以降の秒数(ESP_DATETIME)またはマイクロ秒(ESP_TIMESTAMP)として解釈するには、ヌルに設定します。

defaultOpcode
演算コードフィールドを含まない JSON から構築されたイベントに挿入する演算コード

戻り値: 他のすべての JSON API 関数に渡される JSON オブジェクトへのポインタ。失敗した場合はヌルを返します。

C_dfESPeventblock C_dfESPjsonToEb(jsonObjPtr *json*, const char **serializedJson*, int32_t *maxEvents*)

パラメータ:

json
C_dfESPpubsubInitJson()によって作成されたオブジェクトへのポインタ

serializedJson
ソケットで受信されたシリアル化された JSON へのポインタ

maxEvents
作成されるイベントの数をこの値に制限します。値 0 は無制限を意味します。

戻り値: イベントブロックポインタ

void *C_dfESPebToJson(jsonObjPtr *json*, C_dfESPeventblock *C_eb*)

パラメータ:

json
C_dfESPpubsubInitJson()によって作成されたオブジェクトへのポインタ

C_eb
イベントブロックポインタ

戻り値: シリアル化された JSON へのポインタ

C_dfESPeventblockV C_dfESPxmlToEb(xmlObjPtr *xml*, const char *serializedXml, int32_t *maxEvents*)

パラメータ:

xml
C_dfESPpubsubInitXml()によって作成されたオブジェクトへのポインタ

serializedXml
ソケットで受け取ったシリアル化された XML へのポインタ

maxEvents
処理されるイベントの最大数。値 0 は無制限を意味します。

戻り値: イベントブロックポインタ

xmlObjPtr C_dfESPpubsubInitXml(C_dfESPschema *C_schema*, char **dateFormat*, C_dfESPeventcodes *defaultOpcode*, uint32_t *doublePrecision*)

パラメータ:

C_schema
ウィンドウスキーマへのポインタ

dateFormat
CSV コンバージョンの日付形式。ESP_DATETIME および ESP_TIMESTAMP フィールド値を、エポック以降の秒数(ESP_DATETIME)またはマイクロ秒(ESP_TIMESTAMP)として解釈するには、ヌルに設定します。

defaultOpcode
演算コードフィールドを含まない XML からビルドされたイベントに挿入する演算コード

doublePrecision
倍精度変数の ASCII 表現の小数点以下の桁数です。製品のデフォルト値を使用するには、値 6 を渡します。

xmlObjPtr C_dfESPpubsubInitXml(C_dfESPschema *C_schema*, char **dateFormat*, C_dfESPeventcodes *defaultOpcode*, uint32_t *doublePrecision*)	
戻り値:	XML オブジェクトへのポインタ。このポインタは、他のすべての XML API 関数に渡されます。障害が発生した場合、値はヌルです。

void *C_dfESPebToXml(xmlObjPtr *xml*, C_dfESPeventblock *C_eb*)	
パラメータ:	*xml* C_dfESPpubsubInitXml()によって作成されたオブジェクトへのポインタ *C_eb* イベントブロックポインタ
戻り値:	直列化された XML へのポインタ

AC ライブラリには、クライアント開発者がアプリケーションまたはサーバーからのイベントストリーム処理オブジェクトを分析および操作できるようにする一連の関数が用意されています。これらの関数は、C++ Modeling API で提供されるメソッドの小さなサブセットを囲む C ラッパーのセットです。これらのラッパーを使用すると、クライアント開発者は C++ではなく C を使用できます。これらのオブジェクトの例は、イベント、イベントブロック、およびスキーマです。これらの呼び出しの小さなサンプリングは、次のとおりです。完全な呼び出しについては、**$DFESP HOME/doc/html**l にある API リファレンスのドキュメントを参照してください。

イベントブロックのサイズを取得するには:	C_ESP_int32_t eventCnt = C_dfESPeventblock_getSize(eb);
イベントブロックからイベントを抽出するには:	C_dfESPevent ev = C_dfESPeventblock_getEvent(eb, eventIndx);
オブジェクト(この場合はスキーマの文字列表現)を作成するには:	C_ESP_utf8str_t schemaCSV = C_dfESPschema_serialize(schema);
オブジェクト(この場合は文字列のベクトル)を解放するには:	C_dfESPstringV_free(metaVector);

Java パブリッシュ/サブスクライブ API の使用

Java パブリッシュ/サブスクライブ API の概要

SAS Event Stream Processing と C パブリッシュ/サブスクライブ API は SAS ロギングライブラリを使用しますが、Java パブリッシュ/サブスクライブ API は **java.util.logging** パッケージの Java ロギング API を使用します。ログレベルと Java ロギングの詳細については、そのパッケージを参照してください。

Java パブリッシュ/サブスクライブ API は 2 つのパッケージで提供されています。これらのパッケージは、次のパブリックインターフェイスを定義します。

- **com.sas.esp.api.pubsub**

  - **com.sas.esp.api.pubsub.clientHandler**
  - **com.sas.esp.api.pubsub.clientCallbacks**
- **com.sas.esp.api.server**
  - **com.sas.esp.api.server.datavar**
  - **com.sas.esp.api.server.event**
  - **com.sas.esp.api.server.eventblock**
  - **com.sas.esp.api.server.library**
  - **com.sas.esp.api.server.schema**

クライアントは、イベントストリームプロセッサアプリケーションまたはサーバーをいつでも照会して、現在実行中のウィンドウ、連続クエリ、およびプロジェクトをさまざまな粒度で検出できます。この情報は、名前を表す文字列のリストの形式でクライアントに返されます。このリストは、subscriberStart()または publisherStart()に渡す URL 文字列を作成するために使用できます。

注: 注:Java API に渡されるイベントストリーム発行 URL が認証用に?username パラメータを含む場合、クラスパスに **cas-client-*.jar** を含める必要があります。

Java パブリッシュ/サブスクライブ API のパラメータと使用法は、C パブリッシュ/サブスクライブ API の同等の呼び出しと同じです。

C API リファレンスと Java インターフェースリファレンスは、**$DFESP_HOME/doc/html** にあります。

高レベルのパブリッシュ/サブスクライブメソッドの使用

次の高レベルパブリッシュ/サブスクライブメソッドは、次のインタフェース参照で定義されています。**com.sas.esp.api.pubsub.clientHandler**。

方法	説明
boolean init(Level level)	パブリッシュ/サブスクライブサービスを初期化します
dfESPclient publisherStart(String serverURL, clientCallbacks userCallbacks, Object ctx)	サイトパブリッシャを開始します。
dfESPclient subscriberStart(String serverURL, clientCallbacks userCallbacks, Object ctx)	サブスクライバを開始します
boolean connect(dfESPclient client)	イベントストリームプロセッサアプリケーションまたはサーバーに接続します
boolean publisherInject((dfESPclient client, dfESPeventblock eventblock)	イベントブロックをパブリッシュします
ArrayList< String > *queryMeta* (String *queryURL*)	クエリモデルのメタデータ
ArrayList< String > getModel(String *queryURL*)	クエリモデルウィンドウとそのエッジ
boolean disconnect (dfESPclient client, boolean block)	イベントストリームプロセッサから切断します
boolean stop (dfESPclient client, boolean block)	サブスクライバまたはパブリッシャを停止します

方法	説明
void shutdown ()	パブリッシュ/サブスクライブサービスをシャットダウン
boolean setBufferSize(dfESPclient client, int mbytes)	デフォルトのソケット読み書きバッファサイズを変更する
dfESPclient GDsubscriberStart (String serverURL, clientCallbacks userCallbacks, Object ctx, String configFile)	保証された配信サブスクライバを開始します
dfESPclient GDpublisherStart (String serverURL, clientCallbacks userCallbacks, Object ctx, String configFile)	保証配信パブリッシャを開始します
long GDpublisherGetID()	順次配信されるイベントブロックに書き込むための ID を順次取得
boolean GDsubscriberAck(dfESPclient client, dfESPeventblock eventblock)	保証された配信通知をトリガします
boolean persistModel(String hostportURL, String persistPath)	実行中のエンジンに現在の状態をディスクに保存するよう指示します
boolean quiesceProject(String projectURL, dfESPclient client)	実行中のエンジンにモデル内の特定のプロジェクトを休止するよう指示します。
boolean subscriberMaxQueueSize(String serverURL, int maxSize, boolean block)	サブスクライバに送信されたイベントブロックをエンキューするために使用される、イベントストリーム処理サーバー内のキューの最大サイズを構成します。キューのサイズが **maxSize** を下回るのを待つために **block** を 1 に設定し、そうでなければクライアントを切断します。
boolean pingHostPort(String hostportURL)	実行中のエンジンに ping を実行して、指定されたパブリッシュ/サブスクライブポートが開いているかどうかを確認します。また、オープンポートがパブリッシュ/サブスクライブポートであることを確認するために、マジックナンバーを交換して確認します。
boolean setTokenLocation(String tokenLocation)	パブリッシュ/サブスクライブサーバーによる認証に必要な OAuth トークンを含むローカルファイルシステム内のファイルの場所を設定します。
void setTransportConfigFile(String xportCfgFile)	ローカルファイルシステム内のパブリッシュ/サブスクライブトランスポート構成ファイルの場所を設定します。使用されている代替のトランスポートライブラリに応じて、デフォルト値は**"kafka.cfg"** 、 **"rabbitmq.cfg"** 、 **"solace.cfg"** 、または**"client.config"**です。

詳細は、 **$DFESP_HOME/doc/html/index.html** を参照してください。**クライアントハンドラ**については、**クラス**ページを検索してください。

Google プロトコルバッファをサポートするメソッドの使用

次のメソッドは、Google プロトコルバッファをサポートしています。これらは、このインタフェース参照で定義されています: **com.sas.esp.api.pubsub.protobufInterface**。

方法	説明
boolean init(String fileDescriptorSet, String msgName, dfESPschema schema, EventOpcodes defaultOpcode)	Google プロトコルバッファをサポートするライブラリを初期化します。
dfESPeventblock protobufToEb(byte[] serializedProtobuf)	protobuf メッセージをイベントブロックに変換します。
byte[] ebToProtobuf(dfESPeventblock eb, int index)	イベントブロック内のイベントを protobuf メッセージに変換します。

詳細は、“Google プロトコルバッファのパブリッシュ/サブスクライブ API サポート”を参照してください。

ユーザー提供のコールバック関数の使用

com.sas.esp.api.pubsub.clientCallbacks インタフェースリファレンスは、ユーザー提供のコールバック関数のシグネチャを定義します。現在、次の 3 つの機能があります。

- サブスクライブされたイベントブロックハンドラ
- パブリッシュ/サブスクライブ失敗ハンドラ
- 保証配信 ACK-NACK ハンドラ

サブスクライブされたイベントブロックハンドラは、サブスクライバクライアントによってのみ使用されます。これは、アプリケーションまたはサーバーからの新しいイベントブロックが到着したときに呼び出されます。イベントブロックを処理した後、クライアントは eventblock_destroy()を呼び出して解放します。

このユーザー定義のコールバックのシグネチャは次のとおりです。

```
void com.sas.esp.api.pubsub.clientCallbacks.dfESPsubscriberCB_func
   (dfESPeventblock eventBlock, dfESPschema schema, Object ctx)
```

- eventBlock は読み込まれたイベントブロックです
- schema は、クライアント処理用のイベントのスキーマです。
- ctx は、コール状態を維持するためのオプションのコンテキストポインタです。

パブリッシュ/サブスクライブクライアント ERROR 処理の 2 番目のコールバック関数は、サブスクライバクライアントとパブリッシャクライアントの両方でオプションです。指定されている場合(つまりヌルでない場合)、クライアントサービス内の異常な切断のような異常なイベントが発生するたびに呼び出されます。これにより、クライアントは正常に処理し、場合によってはパブリッシュ/サブスクライブサービス ERROR から回復することができます。このコールバック関数のシグネチャは次のとおりです。

- failure は pubsubFail_APIFAIL、pubsubFail_THREADFAIL、または pubsubFail_SERVERDISCONNECT です。
- code は、失敗の特定のコードを提供します。
- ctx は、状態データ構造体へのオプションのコンテキストポインタです。

```
void com.sas.esp.api.pubsub.clientCallbacks.dfESPpubsubErrorCB_func
   (clientFailures failure, clientFailureCodes code, Object ctx)
```

clientFailures および client FailureCodes は、インタフェース参照 **com.sas.esp.api.pubsub.clientFailures** および **com.sas.esp.api.pubsub.clientFailureCodes** で定義されています。

保証された配信 ACK-NACK ハンドラは、特定のイベントブロックのステータスを提供するために、またはすべてのサブスクライバが接続され、パブリッシュが開始できることをパブリッシャに通知するために呼び出されます。このコールバック関数のシグネチャは次のとおりです。

```
void com.sas.esp.api.pubsub.clientCallbacks.dfESPGDpublisherCB_func
  (clientGDStatus eventBlockStatus, long eventBlockID, Object ctx)
```

どこで

- eventBlockStatus は、ESP_GD_READY、ESP_GD_ACK、または ESP_GD_NACK です。
- eventBlockID は、パブリッシュ前にイベントブロックに書き込まれた ID です。
- ctx は、状態データ構造体へのオプションのコンテキストポインタです。

Java クライアント用の代替トランスポートライブラリの使用

代替のトランスポートライブラリを使用すると、Java パブリッシュ/サブスクライブクライアントアプリケーションは、クライアントへの直接 TCP/IP 接続以外のメカニズムを使用してイベントブロックを送受信できます。

- Rabbit MQ Java ライブラリを使用すると、Rabbit MQ サーバーを介して送受信できます。
- Solace Java ライブラリを使用すると、Solace アプライアンスを介して送受信できます。
- Tervela Java ライブラリを使用すると、Tervela アプライアンスを介して送受信できます。
- Kafka Java ライブラリは、Kafka クラスタを介して送受信できます。

これらのライブラリは、Java の C パブリッシュ/サブスクライブ API メソッドと同等のものを使用して、代替トランスポートを置き換えます。エンジンが Rabbit MQ、Solace、Tervela、または Kafka を使用して 1+N-Way フェイルオーバー用に構成されている場合、Java パブリッシュ/サブスクライブクライアントは対応するクライアントライブラリを使用してフェイルオーバーを success させる必要があります。

Java パブリッシュ/サブスクライブクライアントアプリケーションでこれらのライブラリの 1 つを置き換えるには、次のように、クラスパスの **dfx-esp-api.jar** の前に対応する JAR filename を挿入します。

- Rabbit MQ の場合、JAR ファイルは **dfx-esp-rabbitmq-api.jar** です。
- Solace の場合、JAR ファイルは **dfx-esp-solace-api.jar** です。
- Tervela の場合、JAR ファイルは **dfx-esp-tervela-api.jar** です。
- Kafka の場合、JAR ファイルは **dfx-esp-kafka-api.jar** です。

Rabbit MQ ライブラリを使用する場合は、ネイティブ Rabbit MQ Java クライアントライブラリ(**rabbitmq-client.jar**)もインストールする必要があります。http://www.rabbitmq.com/java-client.html から入手してください。次に、**$DFESP_HOME/lib** から **commons-configuration-1.10.jar** 、**commons-lang-2.6.jar** 、および **commons-logging-1.2.jar** と一緒に **rabbitmq-client.jar** をクラスパスに追加します。

Kafka ライブラリを使用している場合、ネイティブの Kafka Java クライアントライブラリ(**kafka-clients-*.jar**)もインストールする必要があります。現在、バージョン 0.9.x のみがサポートされています。http://kafka.apache.org/downloads.html で入手してください。次に、 **$DFESP_HOME/lib** から **commons-configuration-1.10.jar** 、**commons-lang-2.6.jar** 、および **commons-logging-1.2.jar** と一緒に **kafka-clients-*.jar** をクラスパスに追加します。

1+N-Way フェイルオーバーをサポートする Java パブリッシャアダプタに Kafka ライブラリを使用する場合は、Apache Zookeeper Java クライアントライブラリもインストールする必要があります。http://zookeeper.apache.org/releases.html で入手し、クラスパスに追加します。

現在の作業ディレクトリには、次のような対応する構成ファイルも含まれている必要があります。

- RabbitMQ の場合、このファイルの名前は **rabbitmq.cfg** でなければなりません。

- Solace の場合は、 **solace.cfg** という名前にする必要があります。
- Tervela のの場合、**client.config** という名前にする必要があります。
- Kafka の場合、**kafka.cfg** という名前にする必要があります。

または、setTransportConfigFile()メソッドを使用して、別のディレクトリにある構成ファイルの絶対パスを指定することもできます。

RabbitMQ の設定ファイルの例を次に示します。

```
{
rabbitmq =
{
host = "my.machine.com";
port = "5672";
exchange = "SAS";
userid = "guest";
password = "guest";
passwordencrypted = false;
}
sas =
{
buspersistence = false;
queuename = "subpersist";
protobuf = false;
descfile = "./GpbHistSimFactory.desc";
protomsg = "GpbTrade";
noreplay = false;
}
}
```

注: passwordencrypted = true の場合、**password** の値は、この OpenSSL コマンドで生成された暗号化されたパスワードでなければなりません。

```
echo "password" | openssl enc -e -aes-256-cbc -a -salt
-pass pass:"SASespRMQclientUsedByUser=userid"
```

buspersistence パラメータと queuename パラメータは、パブリッシャとサブスクライバにとって異なる意味を持ちます。

- パブリッシャの場合、queuename は常に無視されます。buspersistence = false の場合、メッセージは非永続配信モードで送信されます。それ以外の場合、配信モードは永続的です。
- サブスクライバにとって、buspersistence = false は、クライアントによって作成されたすべてのキューおよびエクスチェンジが耐久性がなく、自動削除され、queuename パラメータが無視されることを意味します。buspersistence = true の場合、すべての交換とキューは永続的で自動削除ではなく、耐久性のある受信キューの queuename は固定されます。

デフォルトでは、noreplay パラメータは false です。true に設定すると、buspersistence が有効になっていても、受信したメッセージが肯定応答されます。

Solace の設定ファイルの例を次に示します。

```
{
solace =
{
session = ( "host", "10.37.150.244:55555",
"username", "sub1", "password",
"sub1", "vpn_name", "SAS");
context = ( "CONTEXT_TIME_RES_MS", "50",
"CONTEXT_CREATE_THREAD", "1" );
}
```

```
sas=
{
buspersistence = false;
queuename = "myqueue";
protobuf = false;
descfile = "./GpbHistSimFactory.desc";
protomsg = "GpbTrade";
passwordencrypted = false;
}
}
```

注: passwordencrypted = true の場合、**session.password** の値は、この OpenSSL コマンドで生成された暗号化されたパスワードでなければなりません。

```
echo "session.password" | openssl enc -e -aes-256-cbc -a -salt
-pass pass:"SASespSOLclientUsedByUser=session.username"
```

Tervela の設定ファイルの例を以下に示します。

```
USERNAME esp
PASSWORD esp
PRIMARY_TMX 10.37.8.175
LOGIN_TIMEOUT 45000
GD_CONTEXT_NAME tvaIF
GD_MAX_OUT 10000
PASSWORDENCRYPTED       0
```

注: PASSWORDENCRYPTED = 1 の場合、**PASSWORD** の値はこの OpenSSL コマンドで生成された暗号化されたパスワードでなければなりません。

```
echo "PASSWORD" | openssl enc -e -aes-256-cbc -a -salt
-pass pass:"SASespTVAclientUsedByUser=USERNAME"
```

Kafka の設定ファイルの例を以下に示します:

```
{
kafka =
{
hostport = "kafkahost:9092"
partition = "0"
initialoffset = "largest"
groupid = "mygroup"
metatopic = "mymetatopic"
}
sas =
{
protobuf = false;
descfile = "./GpbHistSimFactory.desc";
protomsg = "GpbTrade";
hotfailover = false;
numbufferedmsgs = "16000"
zookeeperhostport = "myhost:myport"
failovergroup = "mygroup"
}
}
```

Python パブリッシュ/サブスクライブ API の使用

SAS Event Stream Processing Python パブリッシュ/サブスクライブ API は、Python 3.4.*x* を使用して開発され、テストされました。他のバージョンでは、テストされておらずサポートされていません。

API は、Python の ctypes 外部関数ライブラリを使用して実装されています。このライブラリは Python で C パブリッシュ/サブスクライブ API をラップします。したがって、Python パブリッシュ/サブスクライブ API 定義は、C パブリッシュ/サブスクライブ API を反映します。関数は、C 関数と同じ名前を持ちますが、名前から "C_dfESPpubsub"が削除されています。

API は、pubsubApi.py と modelingApi.py という 2 つのファイルで定義され、文書化されています。これらのファイルは、Linux システムの場合は**$DFESP_HOME/lib**、Windows システムの場合は**%DFESP_HOME%\lib** にあります。

Python で書かれたパブリッシャとサブスクライバのクライアントの例は、Linux システムでは**$DFESP_HOME/examples**、Windows システムでは**%DFESP_HOME%\examples** にあります。

Init()関数は、次の処理を行います。

- 必要な C API ライブラリを読み込みます
- Python の関数ポインタを設定します
- ESP ロガーインスタンスを作成します
- Python Init()関数に渡されるロギングレベルとロギング設定ファイルパスを使用して C API C_dfESPpubsubInit()関数を呼び出します
- C_dfESPpubsubInit()が返す戻り値を返します。

modelingApi.py には、C API 関数をラップしない次の余分な関数が含まれています。

- getLogger()
- arrayGetString(instance, i)
- arrayGetInt32(instance, i)

getLogger()は、Init()によって作成された Python ロガーインスタンスを返します。Python クライアントは次のようなロガーインスタンスを取得できます:

logger = logging.getLogger()

logger.addHandler(modelingApi.getLogger())

ロガーは標準の ESP ロギング機能への内部関数ポインタを定義しているため、ログメッセージは ESP サーバーまたは C クライアントによって 記録されたメッセージのように見えます。logger.level(string)を呼び出すことによって、さまざまなレベルでメッセージを記録できます。*level* は **debug**、**info**、**warning**、**error**、または **critical** です。これらのレベルは、SAS Stream Processing ロギングレベル DEBUG、INFO、WARN、ERROR、および FATAL にそれぞれマップされます。

arrayGet*関数は、C 関数によって構築され、SchemaGetNames()などの Python 関数によって返された配列内のエントリを抽出するために使用されます。この抽出は、QueryMeta()によって返されたものなど、C 関数によって構築されたベクトルの処理とは異なります。このようなベクトルのエントリは、StringVGet()を呼び出すことで取得できます。

Google プロトコルバッファのパブリッシュ/サブスクライブ API サポート

Google プロトコルバッファのパブリッシュ/サブスクライブ API サポートの概要

SAS Event Stream Processing では、Google プロトコルバッファをサポートするライブラリが提供されます。このライブラリは、バイナリ形式のイベントブロックとシリアル化された Google プロトコルバッファ(protobuf)の間の変換メソッドを提供します。

protobuf をイベントストリーム処理サーバーと交換するために、標準パブリッシュ/サブスクライブ API を使用するパブリッシュ/サブスクライブクライアントは、C_dfESPpubsubInitProtobuff()を呼び出して、Google プロトコルバッファをサポートするライブラリをロードできます。次に、protobuf 形式のソースデータを持つパブリッシャクライアントは、C_dfESPprotobuffToEb()を呼び出して、C_dfESPpublisherInject()を呼び出す前にバイナリ形式のイベントブロックを作成できます。同様に、サブスクライバクライアントは、C_dfESPebToProtobuff()を呼び出して protobuf ハンドラに渡す前に、受け取ったイベントブロックを protobuf に変換することができます。

注: イベントストリーム処理パブリッシュ/サブスクライブ接続のサーバー側は、Google プロトコルバッファをサポートしていません。イベントブロックはバイナリ形式でのみ送受信し続けます。

SAS Event Stream Processing Java パブリッシュ/サブスクライブ API には、Java パブリッシュ/サブスクライブクライアント用の同等のメソッドを実装する protobuf JAR ファイルが含まれています。Google プロトコルバッファをサポートするライブラリを読み込むには、標準の Google プロトコルバッファランタイムライブラリをインストールしている必要があります。SAS Event Stream Processing ランタイム環境では、このライブラリを検出できる必要があります。Java の場合は、Google の protobufJAR ファイルをインストールし、実行時クラスパスに組み込む必要があります。

パブリッシュ/サブスクライブクライアント接続は、そのウィンドウのスキーマを使用して 1 つのウィンドウとイベントを交換します。したがって、protobuf 対応のクライアント接続では、**.proto** ファイルのメッセージブロックで定義されているように、単一の protobuf 固定メッセージ型を使用します。Google プロトコルバッファをサポートしているライブラリは、メッセージ定義を動的に解析するので、あらかじめコンパイルされたメッセージ固有のクラスは必要ありません。ただし、Java ライブラリは**.proto** ファイルの代わりに**.desc** ファイルを使用します。このため、対応する**.desc** ファイルを生成するために**.proto** ファイルで Google **protoc** コンパイラを実行する必要があります。

C クライアントの場合、**.proto** ファイルの名前と囲まれたメッセージは、両方とも C_dfESPpubsubInitProtobuff()呼び出しで Google プロトコルバッファをサポートするライブラリに渡されます。この呼び出しは、protobuf オブジェクトインスタンスを返します。このインスタンスは、その後、クライアントによってすべての 後続の protobuf 呼び出しに渡されます。このインスタンスは protobuf メッセージ定義に固有です。したがって、特定のウィンドウへのクライアント接続がアップしている限り有効です。クライアントが停止して再起動すると、新しい protobuf オブジェクトインスタンスを取得する必要があります。

Java クライアントの場合、プロセスは少し異なります。クライアントは dfESPprotobuf オブジェクトのインスタンスを作成し、その init()メソッドを呼び出します。その後の protobuf 呼び出しは、このオブジェクトのメソッドを使用して行われ、C++ protobuf オブジェクトで説明されているのと同じ有効範囲が適用されます。

バイナリイベントブロックと protobuf との間の変換は、protobuf メッセージ定義内のフィールドを、関連するパブリッシュ/サブスクライブウィンドウのスキーマ内のフィールドに一致させることによって行われます。protobuf メッセージ定義とウィンドウスキーマが互換性があることを確認してください。protobuf メッセージ定義にオプションのフィールドが含まれている場合は、それらがウィンドウスキーマに含まれていることを確認してください。受信した protobuf メッセージにオプションのフィールドがない場合、イベントの対応するフィールドはヌルに設定されます。逆に、protobuf を構築し、イベントのフィールドにヌルが含まれている場合、対応する protobuf フィールドは設定されていないため、**.proto** ファイルでオプションとして定義する必要があります。

イベントストリームプロセッサーと Google プロトコルバッファのデータ型のマッピングは、次のようにサポートされています。

イベントストリームプロセッサのデータ型	Google プロトコルバッファデータタイプ
ESP_DOUBLE	TYPE_DOUBLE TYPE_FLOAT
ESP_INT64	TYPE_INT64 TYPE_UINT64 TYPE_FIXED64 TYPE_SFIXED64 TYPE_SINT64
ESP_INT32	TYPE_INT32 TYPE_UINT32 TYPE_FIXED32 TYPE_SFIXED32 TYPE_SINT32 TYPE_ENUM
ESP_UTF8STR	TYPE_STRING
ESP_DATETIME	TYPE_INT64
ESP_TIMESTAMP	TYPE_INT64
他のマッピングは現在サポートされていません。	

プロトコルバッファメッセージ内のネストされたフィールドと繰り返しフィールドをイベントブロックに変換する

それらがサポートされている場合は、protobuf のメッセージフィールドを繰り返すことができます。TYPE_MESSAGE のメッセージフィールドはネストすることができ、場合によっては繰り返すこともできます。protobuf メッセージをイベントブロックに変換する際には、次のポリシーに従って、これらのすべてのケースがサポートされます。

- ネストされたメッセージを含む protobuf メッセージは、対応するスキーマが protobuf メッセージの平坦な表現であることを必要とします。たとえば、最初のメッセージが 4 つのフィールドを持つネストされたメッセージで、2 つ目がネストされていない、3 つ目が 2 つのフィールドを持つネストされたメッセージである 3 つのフィールドを含む protobuf メッセージは、4 + 1 + 2 = 7 フィールドのスキーマを必要とします。ネストの深さは制限されません。
- 単一の protobuf メッセージは、ネストされたメッセージフィールドが繰り返されない限り、単一のイベントを含むイベントブロックに常に変換されます。
- protobuf メッセージが繰り返される非メッセージタイプのフィールドを持つ場合、そのフィールドのすべてのエレメントは、イベントのカンマで区切られた単一の文字列フィールドに集められます。このため、protobuf メッセージの繰り返しフィールドに対応するスキーマフィールドは、protobuf メッセージのフィールドタイプに関係なく、タイプ ESP_UTF8STR でなければなりません。

- 繰り返されるネストされたメッセージフィールドを含む protobuf メッセージは、常に複数のイベントを含むイベントブロックに変換されます。反復されるすべてのネストされたメッセージフィールド内の各エレメントに対して 1 つのイベントがあります。

イベントブロックをプロトコルバッファメッセージに変換する

イベントブロックを protobuf に変換することは、protobuf 内のネストされたフィールドと繰り返しフィールドをイベントブロックに変換することと概念的には似ていますが、プロセスにはより多くのコードが必要です。イベントブロック内のすべてのイベントは、別々の protobuf メッセージに変換されます。このため、C_dfESPebToProtobuff()ライブラリ呼び出しは、イベント・ブロック内のどのイベントを変換するかを示す索引パラメーターをとります。イベントブロック内のすべてのイベントに対して、ライブラリをループで呼び出す必要があります。

変換により、生成された protobuf メッセージのネストされたフィールドが正しくロードされます。イベントブロックは繰り返しフィールドをサポートしていないため、結果の protobuf メッセージの繰り返しフィールドには 1 つのエレメントしか含まれていません。

注: protobuf 変換へのイベントブロックは、イベント演算コードが protobuf メッセージにコピーされないため、Insert 演算コードを持つイベントのみをサポートします。protobuf からイベントブロックへの変換では、C C_dfESPpubsubInitProtobuff()関数または Java の init()関数で指定された演算コードを使用します。protobufs がコネクタまたはアダプタによって呼び出される場合、演算コードは、コネクタまたはアダプタが Upsert を使用するように構成されていない限り Insert です。

Google プロトコルバッファの転送のサポート

次のメッセージバスに関連付けられているコネクタとアダプタを使用すると、Google プロトコルバッファのサポートが利用できます。

- IBM WebSphere MQ
- Rabbit MQ
- Solace
- Tervela
- Tibco/RV
- Kafka

これらのコネクタとアダプタは、バイナリイベントブロックではなく、メッセージバスを介して protobuf の転送をサポートします。これにより、パブリッシュ/サブスクライブ API を使用せずに、サードパーティのパブリッシャまたはサブスクライバがメッセージバスに接続し、エンジンと protobuf を交換することができます。protobuf メッセージ形式とウィンドウスキーマは互換性がなければなりません。

コネクタまたはアダプタは、**protofile** および **protomsg** パラメータを介して**.proto** ファイルおよびメッセージ名の構成を必要とします。コネクタは、Google プロトコルバッファをサポートする SAS Event Stream Processing ライブラリを使用して、protobuf をイベントブロックとの間で変換します。さらに、C および Java Solace パブリッシュ/サブスクライブクライアントは、**solace.cfg** クライアント構成ファイルで構成されている場合、Google プロトコルバッファもサポートします。同様に、**rabbitmq.cfg** または **kafka.cfg** クライアント構成ファイルで C RabbitMQ および Kafka パブリッシュ/サブスクライブクライアントが Google プロトコルバッファをサポートするように構成されている場合、C RabbitMQ および Kafka パブリッシュ/サブスクライバクライアントもこれらをサポートします。protobuf 対応のクライアントパブリッシャは、イベントブロックを protobuf に変換し、メッセージバスを介して、Google プロトコルバッファのサードパーティコンシューマーに転送します。同様に、protobuf 対応のクライアントサブスクライバは、メッセージバスから protobuf を受信し、それをイベントブロックに変換します。

JSON メッセージングのパブリッシュ/サブスクライブ API サポート

概要

SAS Event Stream Processing では、JSON メッセージングをサポートする C ライブラリが提供されます。ライブラリは、バイナリ形式のイベントブロックとシリアライズされた JSON メッセージの間の変換メソッドを提供します。

JSON メッセージをイベントストリーム処理サーバーと交換するには、標準パブリッシュ/サブスクライブ API を使用するパブリッシュ/サブスクライブクライアントは、C_dfESPpubsubInitJson()を呼び出して JSON をサポートするライブラリをロードできます。JSON 形式のソースデータを持つパブリッシャクライアントは、C_dfESPjsonToEb()を呼び出して、バイナリ形式のイベントブロックを作成できます。その後、C_dfESPpublisherInject()を呼び出すことができます。

同様に、サブスクライバクライアントは、受け取ったイベントブロックを JSON メッセージハンドラに渡す前に C_dfESPebToJson()を呼び出して JSON メッセージに変換できます。

注: パブリッシュ/サブスクライブ接続を処理するイベントストリームのサーバー側では、JSON メッセージはサポートされません。イベントブロックはバイナリ形式でのみ送受信し続けます。

パブリッシュ/サブスクライブクライアント接続は、そのウィンドウのスキーマを使用して 1 つのウィンドウとイベントを交換します。これに対応して、JSON 対応のクライアント接続は、固定された JSON スキーマとメッセージを交換します。ただし、このスキーマの静的な定義はありません。JSON メッセージと関連ウィンドウスキーマの間のスキーマの不一致は、実行時にのみ検出されます。

C_dfESPpubsubInitJson()コールは、JSON オブジェクトインスタンスを返します。このインスタンスは、その後のすべての JSON 呼び出しでクライアントによって 渡されます。このオブジェクトインスタンスは、特定のウィンドウへのクライアント接続が起動している間のみ有効です。クライアントが停止して Restart すると、新しい JSON オブジェクトインスタンスを取得する必要があります。

基本的に、単一の JSON メッセージは単一のイベントにマッピングされます。ただし、JSON メッセージ内に複数のイベントを含めることができます。

JSON メッセージのネストされたフィールドをイベントブロックに変換する

ウィンドウスキーマは、JSON イベントスキーマのフラット化された表現でなければなりません。ここで、ウィンドウフィールド名は、ネストされた JSON タグ名の連結をアンダースコアで区切ったものです。入力 JSON に、ソースウィンドウスキーマに意図的に欠けているフィールドがあり、無視する必要がある場合は、イベントブロックを作成する前に C_dfESPignoreMissingSchemaFields()メソッドを呼び出します。これにより、ライブラリが ERROR をログに記録することが防止されます。

たとえば、バイトオーダーのエンディアンが 異なる場合でも、 JSON イベントスキーマ内では、配列とオブジェクトの無制限のネストがサポートされています。

JSON イベントに配列フィールドが含まれている場合、その配列内のすべてのエレメントは、イベント内のカンマで区切られた単一の文字列フィールドにまとめられます。このため、JSON イベントの配列フィールドに対応するスキーマフィールドは、JSON イベントスキーマ内のフィールドタイプに関係なく、タイプ ESP_UTF8STR でなければなりません。

JSON イベントから構築されたイベントには常に Insert 演算コードがあります。例外は、JSON イベントに opcode という名前のフィールドが含まれている場合です。その場合、そのフィールドの値はイベント演算コードを設定するために使用されます。他の JSON フィールドがソースウィンドウスキーマのフィールドと一致しない場合、注入操作は失敗します。

表 1 JSON イベントの演算コードフィールドの有効な値

値	演算コード
i **I**	Insert
u **U**	Update
d **D**	Delete
p **P**	Upsert
s **S**	Safe delete

デフォルトでは、イベントブロックは type= normal です。イベントブロック配列内のイベントが追加の配列内に含まれている場合(つまり、追加のかっこで囲まれている場合)、イベントブロックは type= transactional。

設定された部分文字列を含まない JSON 値を除外するように JSON ライブラリを設定することができます。この場合、対応するイベントは静かにドロップされます。このようなフィルタリングを有効にするには、matchsubstrings パラメータを設定し、カンマで区切られた ***key*:*substring*** ペアの文字列を指定します。たとえば、 "foo:bar"その値に "bar"を含まない JSON キー "foo"がイベントストリーム処理イベントを生成しないことを意味します。キーごとに複数の部分文字列を設定する場合、入力 JSON データには、イベントを生成するために構成された部分文字列のうちの 1 つのみが含まれている必要があります。

JSON フィールド(またはフィールドの組み合わせ)をソースウィンドウのキーとして簡単に使用できない場合は、次のキーフィールドのいずれかまたは両方をソースウィンドウスキーマに追加します。

"*eventindex:int64,*adapterindex**:*文字列*"**

ソースウィンドウスキーマに存在する場合、ライブラリは次のようにこれらのフィールドに入力します。

- *eventindex*: イベントごとにインクリメントされる 64 ビットの整数値。プロセス空間内の複数のライブラリにまたがって一意であることが保証されています。
- *adapterindex*: 異なるプロセス空間で実行されているライブラリのインスタンスに対して一意の GUID 文字列。

これらの値を組み合わせることにより、異なるプロセスのライブラリの複数のユーザーが、キーを複製することなくイベントを単一の Source ウィンドウに挿入できます。

注入されたすべてのイベントに静的な JSON 値を含める必要がある場合は、C_dfESPaddStaticJson() メソッドに渡します。後続のすべてのイベントには、その JSON から構築されたフィールドが含まれます。

イベントブロックを JSON メッセージに変換する

イベントから作成されたすべての JSON イベントには、opcode フィールドが含まれています。有効な値は、表 1 にリストされています。

入力はイベントブロックなので、結果として得られる JSON メッセージは常にルートオブジェクトとして配列を持ちます。各配列エントリは単一のイベントを表します。

データ変数型 ESP_MONEY はサポートされていません。

データ変数型 ESP_TIMESTAMP および ESP_DATETIME は、フィールド値の CSV 表現を含む JSON 文字列に変換されます。

JSON メッセージの転送のサポート

JSON メッセージングのサポートは、次のメッセージバスに関連付けられたコネクタとアダプタを使用すると利用できます。

- IBM WebSphere MQ
- RabbitMQ
- Solace
- Tervela
- Tibco/RV
- Kafka

これらのコネクタとアダプタは、バイナリイベントブロックではなく、メッセージバスを介して JSON エンコードされたメッセージの転送をサポートします。これにより、サードパーティのパブリッシャまたはサブスクライバは、パブリッシュ/サブスクライブ API を使用せずに、メッセージバスに接続し、JSON メッセージをエンジンと交換できます。

メッセージ形式構成は必要ありません。JSON スキーマとウィンドウスキーマが互換性がない場合、JSON をイベントブロックに変換するパブリッシャは、イベントブロックが挿入されると失敗します。コネクタは、JSON 変換をサポートする SAS Event Stream Processing ライブラリを使用して、JSON をイベントブロックとの間で変換します。

C RabbitMQ、Solace Systems、および Kafka パブリッシュ/サブスクライブクライアントは、**rabbitmq.cfg**、**solace.cfg**、または **kafka.cfg** クライアント構成ファイルで JSON をサポートするように構成されている場合、JSON をサポートします。JSON 対応クライアントパブリッシャは、イベントブロックを JSON メッセージに変換して、メッセージバスを介して、JSON メッセージのサードパーティコンシューマーに転送します。同様に、JSON 対応クライアントのサブスクライバは、メッセージバスから JSON メッセージを受け取り、それをイベントブロックに変換します。

XML メッセージのパブリッシュ/サブスクライブ API サポート

SAS Event Stream Processing では、XML メッセージングをサポートする C ライブラリが提供されます。このライブラリは、以前に説明した JSON メッセージングライブラリに似ています。以下の API 関数を提供します。

- C_dfESPpubsubInitXml()
- C_dfESPxmlToEb()
- C_dfESPebToXml()
- C_dfESPaddStaticXml()
- C_dfESPignoreMissingSchemaFields()

これらの関数は、JSON ライブラリの対応する JSON 関数と同じ操作を実行します。

保証配信

保証配信の概要

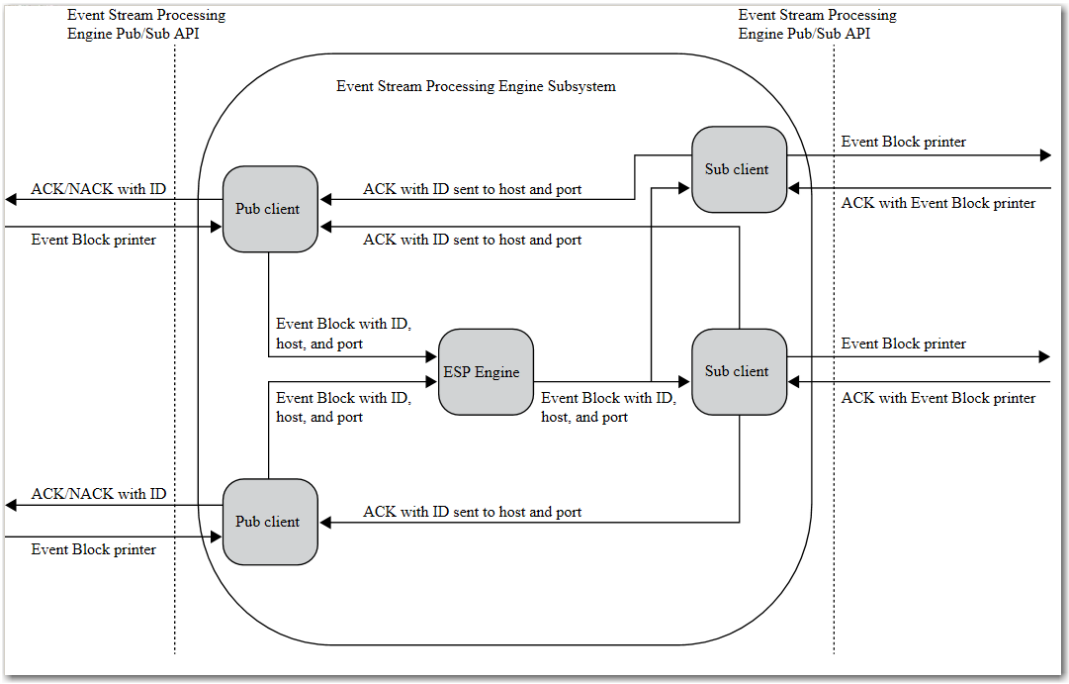
C、Java、および Python パブリッシュ/サブスクライブ API は、単一のパブリッシャと複数のサブスクライバ間の保証された配信をサポートします。保証配信では、ソースウィンドウにパブリッシュされる各イベントブロックが、サブスクライブされたウィンドウ内に正確に 1 つのイベントブロックを生成するモデルが想定されています。この 1 ブロックのブロックアウト原則は、パブリッシュされたすべてのイベントブロックに対して保持する必要があります。保証配信確認メカニズムは、モデルによって実行されるイベント処理を認識していません。

パブリッシュまたはサブスクライブ接続が開始されると、さまざまなパブリッシュ/サブスクライブ・アクティビティを実行するためにクライアントが確立されます。パブリッシュ接続が開始されると、そのイベントブロックの配信を確認するのに必要な保証されたサブスクライバの数が指定されます。予想されるすべてのサブスクライブからの非受信時に否定応答を生成するために使用されるタイムアウト値も指定されます。パブリッシャによって挿入されたすべてのイベントブロックには、パブリッシャによって設定された一意の 64 ビット ID が含まれています。この ID は、パブリッシャのユーザー定義のコールバック関数で、すべての 肯定応答または否定応答でパブリッシュクライアントからパブリッシャに返されます。この関数は、パブリッシュクライアントが開始されたときに登録されます。

サブスクライブ・コネクションが開始されると、サブスクライブ・クライアントは、一連の保証された配信パブリッシャをホストおよびポート・エントリのリストとして渡します。次に、クライアントは、リスト上の各パブリッシャとの TCP 接続を確立します。この接続は、このパブリッシャ/サブスクライバのペアに固有の肯定応答を転送するためにのみ使用されます。サブスクライバは、新しいパブリッシュ/サブスクライブ API 関数を呼び出して肯定応答をトリガします。

イベントブロックには新しいホスト、ポート、および ID フィールドが含まれています。すべてのイベントブロックは、これらのフィールドの組み合わせによって一意に識別されます。これにより、サブスクライバは重複(つまり再送信)イベントブロックを識別できます。

図 1 保証配信データフロー図



次の点に注意してください:

- 保証配信が可能な API 関数を使用しないパブリッシャおよびサブスクライバは、暗黙のうちに配信が無効になっていることを保証します。
- 保証配信サブスクライバは、保証されていない配信サブスクライバと混合することができます。
- 保証配信可能パブリッシャは、READY コールバックが受信されるまでパブリッシュを開始するのを待機する可能性があります。これは、設定されたサブスクライバ数がパブリッシャに肯定応答接続を確立したことを示します。
- エンジンによって生成されたスナップショットの結果として保証配信可能なサブスクライバによって受信されたイベントブロックは承認されません。
- 特定の条件下では、サブスクライバは重複したイベントブロックを受信します。これらの条件には、以下が含まれます。
  - 関連するすべてのサブスクライバが開始する前に、パブリッシャがパブリッシュを開始します。開始されたサブスクライバは、開始されたサブスクライバの数がパブリッシャが渡す必要な肯定応答の数に達するまで、重複するイベントブロックを受信できます。
  - 配信可能な保証付きサブスクライバは、パブリッシャがパブリッシュする間に切断されます。これにより、以前に説明したのと同じシナリオがトリガーされます。
  - 遅いサブスクライバは、イベントブロックのタイムアウトを引き起こし、パブリッシャへの否定応答をトリガします。この場合、パブリッシャに関連するすべてのサブスクライバは、それらのブロックに対してすでに C_dfESPGDsubscriberAck()を呼び出しているものを含む、再送信イベントブロックを受信します。
- 保証された配信可能なサブスクライバが肯定応答接続を確立できなかった場合、構成可能な最大再試行回数まで設定可能な速度で再試行します。
- 保証された配信可能パブリッシャが ID を含むイベントブロックを注入し、その ID がパブリッシュクライアントの非確認 ID リストに存在するとします。この場合、インジェクトコールはパブリッシュクライアントによって拒否されます。パブリッシュクライアントが新しいパブリッシャの ACK/NACK コールバック関数に ID を渡すと、ID はリストから消去されます。

保証配信成功のシナリオ

保証された配信のコンテキストでは、パブリッシャとサブスクライバは、データフローのエンドポイントである顧客アプリケーションです。サブスクライブおよびパブリッシュ・クライアントは、パブリッシャおよびサブスクライバによって発行/発行されるパブリッシュ/サブスクライブ API を実装するイベントストリーム処理コードです。

保証配信成功のシナリオの流れは次のとおりです。

1. パブリッシャはパブリッシュ・クライアントにイベント・ブロックを渡します。パブリッシュ・クライアントでは、イベント・ブロックの ID フィールドがパブリッシャによって設定されています。パブリッシュクライアントは、ホストポートフィールドに入力し、未確認の ID リストに ID を追加し、それをエンジンに注入します。
2. イベントブロックはエンジンによって処理され、サブスクライブウィンドウの挿入、更新、または削除はすべてのサブスクライブクライアントに転送されます。
3. 保証された配信対応サブスクライブクライアントは、イベントブロックを受信し、標準サブスクライバコールバックを使用してサブスクライバに渡します。
4. すべての処理が完了すると、サブスクライバは、イベントブロックポインタで新しい API 関数を呼び出して肯定応答をトリガします。
5. サブスクライブするクライアントは、イベントブロック ID を、イベントブロック内のホストまたはポートに一致する保証された配信肯定応答接続で送信し、エンジンを完全にバイパスします。

6 肯定応答を受信すると、パブリッシュクライアントは、このイベントブロックに対して受信した肯定応答の数を増分します。その番号が起動時にパブリッシュクライアントに渡されたしきい値に達した場合、パブリッシュクライアントはパラメータが確認され、ID で新しい保証された配信コールバックを呼び出します。承認されていない ID のリストから ID を削除します。

保証配信失敗のシナリオ

配信フローの保証には、次の 3 つの失敗シナリオがあります。

シナリオ	説明
イベントブロックのタイムアウト	■ イベントブロック固有のタイマーは、配信保証付きパブリッシュクライアントで期限切れになり、このイベントブロックで受信した肯定応答の数が必要なしきい値を下回っています。 ■ パブリッシュクライアントは、パラメータ NACK および ID を使用して、新しい保証型配信コールバックを呼び出します。このイベントブロックに対して、パブリッシュクライアントまたはサブスクライブクライアントによる更なる再送信やその他のリカバリの試みは行われません。パブリッシャはこのイベントブロックをバックアウトして再送信する可能性が最も高いです。 ■ パブリッシュクライアントは、承認されていない ID のリストから ID を削除します。
無効な保証された配信確認接続試行	■ 保証配信対応のパブリッシュクライアントは、保証された配信確認済みサーバーで接続試行を受け取りますが、必要なクライアント接続の数は既に満たされています。 ■ パブリッシュクライアントは接続を拒否し、ERROR メッセージを記録します。 ■ 保証された配信有効サブスクライブクライアントによって受信された後続のイベントブロックに対して、ERROR メッセージが記録されます。
無効なイベントブロック ID	■ 保証配信可能パブリッシャは、パブリッシュクライアントの未承認 ID リストに既に存在する ID を含むイベントブロックを挿入します。 ■ インジェクトコールはパブリッシュクライアントによって拒否され、ERROR メッセージが記録されます。

保証配信のためのパブリッシュ/サブスクライブ API への追加

パブリッシュ/サブスクライブ API は、保証された配信セッションを実装するための次のメソッドを提供します。

- C_dfESPGDpublisherStart()
- C_dfESPGDsubscriberStart()
- C_dfESPGDsubscriberAck()
- C_dfESPGDpublisherCB_func()
- C_dfESPGDpublisherGetID()

詳細については、を参照してください。機能の保証された配信バージョンのないパブリッシュ/サブスクライブ操作では、標準のパブリッシュ/サブスクライブ API 関数を呼び出します。

構成ファイルの内容

パブリッシュクライアントおよびサブスクライブクライアントは、起動時に構成ファイルを読み取り、保証された配信のためのクライアント固有の構成情報を取得します。これらのファイルの形式は次のとおりです。

配信可能なパブリッシャ設定ファイルの内容を保証	
保証された配信確認接続サーバーのローカルポート番号。	
否定応答を生成するためのタイムアウト値(秒単位)。	
タイムアウト期間内に必要な受信肯定応答の数で、否定応答の代わりに肯定応答を生成します。	
ファイル形式	GDpub_port=<port> GDpub_timeout=<timeout> GDpub_numSubs=<number of subscribers generating acknowledged>

保証された配信有効サブスクライバ構成ファイルの内容	
配信可能なパブリッシャホストまたはポートエントリの保証リスト。各エントリは、サブスクライバが保証された配信イベントブロックを受信したい保証された配信可能パブリッシャに対応するホスト:ポートのペアを含む。	
肯定応答接続の再試行間隔(秒単位)。	
肯定応答接続の最大再試行回数。	
ファイル形式	GDsub_pub=<host:port> GDsub_retryInt=<interval> GDsub_maxRetries=<max>