Overview to the Publish/Subscribe APIs

SAS Event Stream Processing provides publish/subscribe application programming interfaces (APIs) for C, Java, and Python. Use these APIs to do the following:

- publish event streams into a running event stream processor project source window
- subscribe to a project's window event stream, either from the same machine or from another machine on the network

The APIs support TCP/IP networking. Thus, publish and subscribe applications can run on any machine with network access to an event stream processing engine, or on the engine platform itself. APIs are available on all architectures supported by SAS Event Stream Processing.

The publish/subscribe APIs handle cross-platform usage. For example, you can subscribe using the Java API to event streams in an engine that runs on Linux even though the byte order Endianness is different.

You can also subscribe to an event stream so that it can be continuously loaded into a database for persistence. In this case, you more likely would use an event stream processor database connector or adapter.

Connectors are in-process classes that publish or subscribe to and from event stream processing windows. Adapters are stand-alone executable files that publish and subscribe, potentially over a network.

Note: The APIs provide cross-platform connectivity and Endianness compatibility between the application and other networked applications, clients, and data feeds. The APIs are IPv4 compliant.
Using the C Publish/Subscribe API

The C Publish/Subscribe API from the Engine’s Perspective

To enable publish/subscribe for the engine instance using the C++ Modeling API, you must provide a port number to the `pubsub_ENABLE()` parameter in the `dfESPengine::initialize()` call as follows:

```c
dfESPengine *engine;
engine = dfESPengine::initialize(argc, argv, "engine",
    pubsub_ENABLE(33335));
if (engine == NULL) {
    cerr <<"Error: dfESPengine::initialize() failed\n";
    return 1;
}
```

Note: The XML server equivalent is to specify "--pubsub port" on the command line, or specify the port using the port attribute in the engine element.

Clients can use the port number (in this example 33335) to establish publish/subscribe connections. If publish/subscribe is not required, then use `pubsub_DISABLE` for that parameter.

If publish/subscribe is required and clients must be authenticated, use one of the following:

- `pubsub_ENABLE_OAUTH(port, clientId)`, where `clientId` is a CF UAA OAuth server client ID
- `pubsub_ENABLE_SASLOGONAUTH(port, sasLogonURL)`, where `sasLogonURL` is the base services URL for a SASLogon service
- `pubsub_ENABLE_KERBEROS(port, serviceName)`, where `serviceName` is the service name portion of the Kerberos Service Principal Name.

For the XML server, specify `-auth oauth://clientId` or `-auth saslogon://sasLogonURL` or `-auth kerberos://serviceName`.

For more information, see “Enabling Authentication on Socket Connections” in SAS Event Stream Processing: Security.

To initialize publish/subscribe capabilities for a project, `project->setPubSub()` is called before calling `engine->startProjects()`.

For example:

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

Note: The XML server equivalent is to specify the publish/subscribe capabilities using the `pubsub` attribute in the `project` element.

This code opens a server listener socket on port 33335 to enable client subscribers and publishers to connect to the engine application or server for publish/subscribe services. After the connection request is made for publish/subscribe by a client (as described below), an ephemeral port is returned, which the publish/subscribe API uses for this connection.

In cases when you need to override ephemeral ports for a specific port (for security purposes), specify `project->setPubSub` with a second parameter that is the preferred port to be used for the actual connections to this project.

For example:
The first parameter of `project->setPubSub()` applies only to subscription services and it specifies how windows in the project are enabled to support client subscriptions. Specifying `ps_AUTO` enables clients to subscribe to all window output event streams in the project.

Alternatively, you can enable windows manually by specifying `ps_MANUAL`. For non-trivial projects, enable the specific windows of interest manually because automatically enabling all windows has a noticeable impact on overall performance. You can also specify `ps_NONE`, which disables subscribing for all windows.

If you use `ps_MANUAL` in `project->setPubSub()` to specify manual enabling of window subscribes, then use `enableWindowSubs()` for each desired window to enable the subscribe as follows:

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

The XML server equivalent is to enable or disable pubsub using the `pubsub` attribute in the `window` element.

If, however, you specified `ps_AUTO` or `ps_NONE` in `setPubSub()`, then subsequent calls to `enableWindowSubs()` are ignored and generate a warning.

Clients that implement a publisher can then call `C_dfESPpublisherInject()` as needed to publish event blocks into the source window specified in the URL passed to `C_dfESPpublisherStart()`.

The specifics of the client publish/subscribe API are as follows.

- Your client application must include the header file `C_dfESPpubsubApi.h` to provide publisher and subscriber services. In addition to the API calls, this file also defines the signatures of the user-supplied callback functions, of which there are currently two: the subscribed event block handler and the publish/subscribe failure handler.

- The subscribed event block handler is used only by subscriber clients. It is called when a new event block from the application or server arrives. After processing the event block, the client is responsible for freeing it by calling `C_dfESPeventblock_destroy()`. The signature of this user-defined callback is as follows, where "eb" is the event block just read, "schema" is the schema of the event for client processing, and `ctx` is an optional context object containing call state:

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

- The second callback function, `C_dfESPpubsubErrorCB_func()`, is optional for both subscriber and publisher clients. If supplied (that is, no NULL), it is called for every occurrence of an abnormal event within the client services, such as an unsolicited disconnect. This enables the client to handle and possibly recover from publish/subscribe services errors. The signature for this callback function is below, where the following is true:

  - failure is either `pubsubFail_APIFAIL`, `pubsubFail_THREADFAIL`, or `pubsubFail_SERVERDISCONNECT`
  - code provides the specific code of the failure
ctx is an optional context object containing call state.

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

- The `C_dfESPpubsubFailures` and `C_dfESPpubsubFailureCodes` enums are defined in `C_dfESPpubsubFailures.h`.

- A publisher client uses the `C_dfESPpublisherInject()` API function to publish event blocks into a source window in the application or server. The event block is injected into the source window running in the continuous query and project specified in the URL passed to `C_dfESPpublisherStart()`. A client can publish events to multiple windows in a project by calling `C_dfESPpublisherStart()` once for each window and then passing the appropriate client object to `C_dfESPpublisherInject()` as needed.

- A client can query the application or server at any time to discover currently running windows, continuous queries, and projects in various granularities. This information is returned to the client in the form of a list of strings that represent names, which might subsequently be used to build URL strings to pass to `C_dfESPsubscriberStart()` or `C_dfESPpublisherStart()`. See the function description for a list of supported queries.

### Functions for the C Publish/Subscribe API

The functions provided for client publish/subscribe in the publish/subscribe API are as follows. You can use them for simple connections or for more robust and complex connections with multiple connections or recovery handling by the client.

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int C_dfESPpubsubInit(C_dfESPLoggingLevel level, const char *logConfigPath)</code></td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td><code>level</code></td>
</tr>
<tr>
<td>the logging level</td>
</tr>
<tr>
<td><code>logConfigPath</code></td>
</tr>
<tr>
<td>the full pathname to the log configuration file</td>
</tr>
<tr>
<td>Return values:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>success</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>failure — an error is written to the log</td>
</tr>
</tbody>
</table>
| Note: This function initializes client publisher and subscriber services, and must be called (only once) before making any other client calls, with the exception of `C_dfESPpubsubSetPubsubLib()`.

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clientObjPtr C_dfESPpublisherStart(char *serverURL, C_dfESPpubsubErrorCB_func errorCallbackFunction, void *ctx)</code></td>
</tr>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td><code>serverURL</code></td>
</tr>
<tr>
<td>string representing the destination host, port, project, continuous query, and window</td>
</tr>
<tr>
<td><code>serverURL</code> format</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery/window&quot;</td>
</tr>
<tr>
<td><code>errorCallbackFunction</code></td>
</tr>
<tr>
<td>either NULL or a user-defined function pointer for handling client service failures</td>
</tr>
<tr>
<td><code>ctx</code></td>
</tr>
<tr>
<td>optional context pointer for passing state into this call</td>
</tr>
</tbody>
</table>
### clientObjPtr C_dfESPpublisherStart(char *serverURL, C_dfESPpubsubErrorCB_func errorCallbackFunction, void *ctx)

**Return value:**

A pointer to a client object that is passed to all API functions described below or NULL if there was a failure (error written to the log).

**Note:** This function validates and retains the connection parameters for a specific publisher client connection.

---

### clientObjPtr C_dfESPGDpublisherStart()

**Parameters:**

Same parameters and return value as `C_dfESPpublisherStart()`. Additional required parameter: a Guaranteed Delivery callback function pointer of type `C_dfESPGDpublisherCB_func`. Additional required parameter: filename of this publisher’s guaranteed delivery configuration file.

---

### clientObjPtr C_dfESPsubscriberStart(char *serverURL, C_dfESPsubscriberCB_func callbackFunction, C_dfESPpubsubErrorCB_func errorCallbackFunction, void *ctx)

**Parameters:**

- `serverURL` string representing the destination host, port, project, continuous query, and window in the engine. Also specifies the client snapshot requirement - if “true” the client receives the current snapshot state of the window prior to any incremental updates.
  
  Specifying the client collapse requirement is optional. By default, it is false. When true, UPDATE_BLOCK events are converted to UPDATE events in order to make subscriber output publishable.
  
  `serverURL` format
  
  "dfESP://host:port/project/contquery/window?snapshot=true | false <?collapse=true | false><?rmretdel=true | false>"
  
- `callbackFunction` a pointer to a user-defined function for handling received event blocks. This function must call `C_dfESPeventblock_destroy()` to free the event block.
  
- `errorCallbackFunction` either NULL or a user-defined function pointer for handling subscription service failures
  
- `ctx` optional context pointer for parsing state into this call

**Return value:**

A pointer to a client object that is passed to all API functions described below, or NULL if there was a failure (error written to the log).

**Note:** This function validates and retains the connection parameters for a specific subscriber client connection.

---

### clientObjPtr C_dfESPGDsubscriberStart()

**Parameters:**

Same parameters and return value as `C_dfESPsubscriberStart()`. Additional required parameter: filename of this subscriber’s guaranteed delivery configuration file.
### C_dfESPpubsubConnect(clientObjPtr *client)

**Parameter:**
- *client* pointer to a client object returned by C_dfESPsubscriberStart() or C_dfESPpublisherStart() or C_dfESPGDsubscriberStart() or C_dfESPGDpublisherStart()

**Return values:**
- 1 success
- 0 failure — error written to the log

**Note:** This function attempts to establish a connection with the application or server.

---

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

**Parameters:**
- *client* pointer to a client object returned by C_dfESPsubscriberStart() or C_dfESPpublisherStart() or C_dfESPGDsubscriberStart() or C_dfESPGDpublisherStart()
- *block* set to 1 to wait for all queued events to be processed, else 0

**Return values:**
- 1 success
- 0 failure — error written to the log

**Note:** This function closes the connection associated with the passed client object.

---

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

**Parameters:**
- *client* pointer to a client object returned by C_dfESPsubscriberStart() or C_dfESPpublisherStart() or C_dfESPGDsubscriberStart() or C_dfESPGDpublisherStart()
- *block* set to 1 to wait for all queued events to be processed, else 0

**Return values:**
- 1 success
- 0 failure — error written to the log

**Note:** This function stops the client session and removes the passed client object.
```
int C_dfESPpublisherInject(clientObjPtr client, C_dfESPeventblock eventBlock)
```

**Parameters:**

- `client` pointer to a client object returned by `C_dfESPpublisherStart()` or `C_dfESPSubscriptionStart()`
- `eventBlock` the event block to inject into the engine. The block is injected into the source window, continuous query, and project associated with the passed client object.

**Return values:**

- 1 success
- 0 failure — error written to the log

**Note:** This function implements the client publisher function by publishing events into the engine. Event blocks can be built using other additional functions provided in the event stream processor objects C API.

```
C_dfESPstringV C_dfESPpubsubQueryMeta(char *queryURL)
```

**Parameter:**

- `queryURL` string representing the query to be posted to the engine.

**Return value:**

a vector of strings representing the list of names comprising the response to the query, or NULL if there was a failure (error written to the log). The end of the list is denoted by an empty string. The caller is responsible for freeing the vector by calling `C_dfESPstringV_free()`.

**Note:** This function implements a general event stream processor metadata query mechanism. This mechanism enables a client to discover projects, continuous queries, windows, window schema, and window edges currently running in the engine. This mechanism has no dependencies or interaction with any other activity performed by the client. The function opens an independent socket to send the query and closes the socket upon receiving the query reply.

**Supported formats of queryURL**

- "dfESP://host:port?get=projects" returns names of currently running projects
- "dfESP://host:port?get=projects_pubsubonly" returns names of currently running projects with publish/subscribe enabled
- "dfESP://host:port?get=queries" returns names of continuous queries in currently running projects
- "dfESP://host:port/project?get=windows_sourceonly" returns names of source windows in the specified project, if the project is running
- "dfESP://host:port/project?get=windows_derivedonly" returns names of derived windows in the specified project, if the project is running
<table>
<thead>
<tr>
<th>Supported formats of queryURL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dfESP://host:port?get=queries_pubsubonly&quot;</td>
<td>returns names of continuous queries containing publish/subscribe enabled windows in currently running projects</td>
</tr>
<tr>
<td>&quot;dfESP://host:port?get=windows&quot;</td>
<td>returns names of windows in currently running projects</td>
</tr>
<tr>
<td>&quot;dfESP://host:port?get=windows_pubsubonly&quot;</td>
<td>returns names of publish/subscribe enabled windows in currently running projects</td>
</tr>
<tr>
<td>&quot;dfESP://host:port?get=windowsandtypes&quot;</td>
<td>returns names of windows in currently running projects; each returned window name is followed by a colon and the window type</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery?get=windows_sourceonly&quot;</td>
<td>returns names of source windows in the specified continuous query and project, if the project is running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery?get=windows_derivedonly&quot;</td>
<td>returns names of derived windows in the specified continuous query and project, if the project is running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project?get=windows&quot;</td>
<td>returns names of windows in the specified project, if running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project?get=windowsandtypes&quot;</td>
<td>returns names of windows in the specified project, if running. Each returned window name is followed by a colon and the window type</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project?get=windows_pubsubonly&quot;</td>
<td>returns names of publish/subscribe-enabled windows in the specified project, if running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project?get=queries&quot;</td>
<td>returns names of continuous queries in the specified project, if running</td>
</tr>
</tbody>
</table>
**Supported formats of queryURL**

<table>
<thead>
<tr>
<th>queryURL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;dfESP://host:port/project?get=queries_pubsubonly&quot;</td>
<td>returns names of continuous queries containing publish/subscribe-enabled windows in the specified project, if running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery?get=windows&quot;</td>
<td>returns names of windows in the specified continuous query and project, if running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery?get=windowsandtypes&quot;</td>
<td>returns names of windows in the specified continuous query and project, if running. Each returned window name is followed by a colon and the window type.</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery?get=windows_pubsubonly&quot;</td>
<td>returns names of publish/subscribe-enabled windows in the specified continuous query and project, if running</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery/window?get=schema&quot;</td>
<td>returns a single string that is the serialized version of the window schema</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery/window?get=edges&quot;</td>
<td>returns the names of all the window’s edges</td>
</tr>
<tr>
<td>&quot;dfESP://host:port/project/contquery/window?get=rowcount&quot;</td>
<td>returns the number of rows currently in the window</td>
</tr>
</tbody>
</table>

**C_dfESPstringV C_dfESPpubsubGetModel(char *queryURL)**

**Parameter:**

- **queryURL**: string representing the query to be posted to engine.

Supported formats of queryURL are as follows:

- "dfESP://host:port" – returns names of all windows in the model and their edges
- "dfESP://host:port/project" – returns names of all windows in the project and their edges
- "dfESP://host:port/project/contquery" – returns names of all windows in the continuous query and their edges

**Return value:**

A vector of strings representing the response to the query, or NULL if there was a failure (error written to the log). The format of each string is "project/query/window: edge1, edge2, ...". The end of the list is denoted by an empty string. The caller is responsible for freeing the vector by calling **C_dfESPstringV_free()**.
C_dfESPstringV C_dfESPpubsubGetModel(char *queryURL)

Note: This function allows a client to discover an engine by returning the complete set of windows in the model or project or continuous query, along with the window’s edges. It has no dependencies or interaction with any other activity performed by the client. It opens an independent socket to send the query and closes the socket upon receiving the query reply.

void C_dfESPpubsubShutdown()

Shutdown publish/subscribe services

int C_dfESPpubsubPersistModel(char *hostportURL, const char *persistPath)

Parameters:

- **hostportURL**: string in the form "dfESP://host:port"
- **persistPath**: the absolute or relative pathname for the persist file on the target platform

Return values:

- 1: success
- 0: failure — error written to the log

Note: This function instructs the engine at the **hostportURL** to persist its current state to disk. It has no dependencies or interaction with any other activity performed by the client. It opens an independent socket to send the request and closes the socket upon receiving the request return code.

int C_dfESPpubsubQuiesceProject(char *projectURL, clientObjPtr client)

Parameters:

- **projectURL**: string in the form "dfESP://host:port/project"
- **client**: optional pointer to a client object returned by **C_dfESPSsubscriberStart()** or **C_dfESPpublisherStart()** or **C_dfESPGDsubscriberStart()** or **C_dfESPGDpublisherStart()**. If not null, wait for the specified client’s queued events to be processed before quiescing the project.

Return values:

- 1: success
- 0: failure — error written to the log

Note: This function instructs the engine at the **projectURL** to quiesce the project in **projectURL**. This call is synchronous, meaning that when it returns the project has been quiesced.
int C_dfESPsubscriberMaxQueueSize(char *serverURL, int maxSize, int block)

Parameters:

serverURL
String for the server URL in one of the following forms:
- "dfESP://host:port"
- "dfESP://host:port/project"
- "dfESP://host:port/project/contquery"
- "dfESP://host:port/project/contquery/window"

maxsize
the maximum number of event blocks that can be queued for any single subscriber to a window in the serverURL

block
set to 1 to wait for queue size to fall below maxSize, else disconnect the client

Return values:

1
success

0
failure - error written to the log

Note: Use this function to configure the maximum size of all queues used to enqueue event blocks sent to subscribers in a project, query, or window. Use it to limit the amount of memory consumed by these queues. The block parameter specifies the behavior when the maximum is hit.

int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib psLib)

Parameters:

psLib
Number representing the client/server transport

Supported values of psLib are as follows:
- ESP_PSLIB_NATIVE (default)
- ESP_PSLIB_SOLACE — In this mode a client configuration file named solace.cfg must be present in the current directory to provide appliance connectivity parameters.
- ESP_PSLIB_TERVELA — In this mode, a client configuration file named client.config must be present in the current directory to provide appliance connectivity parameters.
- ESP_PSLIB_RABBITMQ — In this mode, a client configuration file named rabbitmq.cfg must be present in the current directory to provide Rabbit MQ server connectivity parameters.
- ESP_PSLIB_KAFKA — In this mode, a client configuration file named kafka.cfg must be present in the current directory to provide Kafka cluster connectivity parameters.
int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib psLib)

Solace configuration file format:

```plaintext
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
}
```

Note: If `passwordencrypted = true`, the value in `SESSION_PASSWORD` must be the encrypted password generated by this OpenSSL command:
```
    echo "SESSION_PASSWORD" | openssl enc -e -aes-128-cbc -a -salt -pass pass:"SASespSOLclientUsedByUser=SESSION_USERNAME"
```

Tervela configuration file format:

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

Note: If `PASSWORDENCRIPTED = 1`, the value in `PASSWORD` must be the encrypted password generated by this OpenSSL command:
```
    echo "PASSWORD" | openssl enc -e -aes-128-cbc -a -salt -pass pass:"SASespTVAcclientUsedByUser=USERNAME"
```
int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib psLib)

RabbitMQ configuration file format:

```ruby
rabbitmq
{
host = "my.machine.com"
port = "5672"
exchange = "SAS"
userid = "guest"
password = "guest"
}

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

Note: If `passwordencrypted = true`, the value in `password` must be the encrypted password generated by this OpenSSL command: `echo "password" | openssl enc -e -aes-128-cbc -a -salt -pass pass:"SASespRMQclientUsedByUser=userid"

Note: The `buspersistence` and `queue` parameters mean different things for subscribers or publishers of Rabbit MQ messages. For a publisher, `queue` is always ignored. If `buspersistence = false`, messages are sent in non-persistent delivery mode. Otherwise, delivery mode is persistent. For a subscriber, `queue` is always used when creating the receive queue. If `buspersistence = false`, all queues and exchanges created by the client are non-durable and auto-delete. If `buspersistence = true`, all exchanges and queues are durable and not auto-delete. The `noreplay` parameter is false by default. When set to true, messages received from Rabbit MQ are acknowledged even when `buspersistence` is enabled. By default, the `noautoack` parameter is set to `false`. When set to `true`, messages received from Rabbit MQ are explicitly acknowledged instead of implicitly acknowledged through the Rabbit MQ `autoack`. This means that any errors detected in received message processing suppress the `ack` and leave the message on the Rabbit MQ queue.
int C_dfESPpubsubSetPubsubLib(C_dfESPpsLib psLib)

Kafka configuration file format:

```c
kafka
{
    hostport = "kafkahost:9092"
    partition = "0"
    initialoffset = "largest"
    groupid = "mygroup"
}
```

```c
sas
    globalconfig = <librdkafka global config>
    topicconfig = <librdkafka topic config>
    {
        protobuf = false
        protofile = "/path/to/protobuf.proto"
        protomsg = "MyProto"
        json = false
        dateformat = "Y-m-d H:M:S"
    }
```

Return values:

- 1 = success
- 0 = failure

Note: This function call is optional, but if called it must be called before calling `C_dfESPpubsubInit()`. It modifies the transport used between the client and the engine from the default peer-to-peer TCP/IP based socket connection that uses the ESP publish/subscribe protocol. Instead, you can specify `ESP_PSLIB_SOLACE`, `ESP_PSLIB_TERVELA`, `ESP_PSLIB_RABBITMQ`, or `ESP_PSLIB_KAFKA` to indicate that the client’s TCP/IP peer is a Solace appliance, a Tervela appliance, a RabbitMQ server, or a Kafka cluster. This mode requires that the engine runs a Solace, Tervela, RabbitMQ, or Kafka connector to provide the corresponding inverse client to the appliance. The topic names used by the appliance are coordinated by the publish/subscribe client and connector to correctly route event blocks through the appliance.

Note: When using the Solace, Tervela, RabbitMQ, or Kafka transports, the following publish/subscribe API functions are not supported:

- `C_dfESPpubsubGetModel()`
- `C_dfESPpublisherStart()`
- `C_dfESPpublisherGetID()`
- `C_dfESPsubscriberStart()`
- `C_dfESPsubscriberAck()`
- `C_dfESPpubsubSetBufferSize()`
- `C_dfESPsubscriberMaxQueueSize()`
- `C_dfESPpubsubPingHostPort()`

---

### C_dfESPGDsubscriberAck(clientObjPtr client, CdfESPeventblock eventblock)

**Parameters:**

- `client` pointer to a client object returned by `C_dfESPGDsubscriberStart()`
- `eventBlock` pointer to the event block being acknowledged back to the publisher. The event block must not be freed before this function returns.

**Return values:**

- 1 = success
- 0 = failure
C_dfESP GDpublisherCB_func()

Parameters:
- Signature of the guaranteed delivery publisher callback function passed to C_dfESPGDpublisherStart(). This function is invoked by the API to return the guaranteed delivery status of a published event block back to the publisher.
  - Parameter 1: READY or ACK or NACK (acknowledged or not acknowledged).
  - Parameter 2: 64-bit event block ID
  - Parameter 3: the user context pointer passed to C_dfESPGDpublisherStart()

Return value: Void

C_dfESPGDpublisherGetID()

Return value: 64-bit event block ID. Might be called by a publisher to obtain sequentially unique IDs to be written to event blocks before injecting them to a guaranteed delivery-enabled publish client.

int C_dfESPpubsubSetBufferSize(clientObjPtr client, int32_t mbytes)

Parameters:
- client pointer to a client object returned by C_dfESPsubscriberStart(), C_dfESPpublisherStart(), C_dfESPGDsubscriberStart(), or C_dfESPGDpublisherStart()
- mbytes the read and write buffer size, in units of 1MB

Return values:
- 1 success
- 0 failure

Note: This function call is optional, but if called it must be called after C_dfESPsubscriberStart(), C_dfESPpublisherStart(), C_dfESPGDsubscriberStart(), or C_dfESPGDpublisherStart() and before C_dfESPpubsubConnect(). It modifies the size of the buffers used for socket Read and Write operations. By default this size is 16MB

int C_dfESPpubsubPingHostPort(char *serverURL)

Parameters:
- serverURL A string with the format "dfESP://host:port"

Return values:
- 1 port is open and it is a publish/subscribe port
- 0 port is not open or is not a publish/subscribe port

Note: This function pings a running engine to determine whether the specified port is open. It also exchanges and verifies a magic number in order to confirm that the open port is a publish/subscribe port.
int C_dfESPpubsubSetTokenLocation(char *tokenLocation)

Parameters:  
    tokenLocation  
    full path and filename of the file that contains the token

Return values:  
    1  
    success  
    0  
    failure

Note:  This function sets the location of the file in the local file system that contains the OAuth token that is required for authentication by the publish/subscribe server.

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

Parameters:  
    protoFile  
    Path to the Google .proto file that contains the message definition.  
    msgName  
    Name of the target message definition in the Google .proto file.  
    C_schema  
    Pointer to the window schema.  
    dateFormat  
    Date format for CSV conversions. Set to null to interpret ESP_DATETIME and ESP_TIMESTAMP field values as an integer number of seconds (ESP_DATETIME) or microseconds (ESP_TIMESTAMP) since epoch.  
    defaultOpcode  
    The opcode to insert into events that are built from a serialized protobuf.

Return value:  
    A pointer to a protobuff object, which is passed to all other protobuff API functions. NULL when there is a failure.

C_dfESPeventblock C_dfESPprotobuffToEb(protobuffObjPtr protobuff, void *serializedProtobuff)

Parameters:  
    protobuff  
    pointer to the object created by C_dfESPpubsubInitProtobuff()  
    serializedProtobuff  
    pointer to serialized protobuff received on a socket

Return value:  
    Event block pointer
### void *C_dfESPebToProtobuff(protobuffObjPtr protobuff, C_dfESPeventblock C_eb, int32_t index)

**Parameters:**
- `protobuff`: pointer to the object created by `C_dfESPpubsubInitProtobuff()`
- `C_eb`: event block pointer
- `index`: index of the event to convert in the event block

**Return value:** pointer to serialized protobuff

### void C_dfESPdestroyProtobuff(protobuffObjPtr protobuff, void *serializedProtobuff)

**Parameters:**
- `protobuff`: pointer to the object created by `C_dfESPpubsubInitProtobuff()`
- `serializedProtobuff`: pointer to serialized protobuff received on a socket

### jsonObjPtr C_dfESPpubsubInitJson(C_dfESPschema C_schema, char *dateFormat C_dfESPeventcodes defaultOpcode)

**Parameters:**
- `C_schema`: pointer to the window schema
- `dateFormat`: Date format for CSV conversions. Set to null to interpret ESP_DATETIME and ESP_TIMESTAMP field values as an integer number of seconds (ESP_DATETIME) or microseconds (ESP_TIMESTAMP) since epoch.
- `defaultOpcode`: the opcode to insert into events that are built from JSON that contain no opcode field

**Return value:** A pointer to a JSON object, which is passed to all other JSON API functions. NULL when there is a failure.

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

**Parameters:**
- `json`: pointer to the object created by `C_dfESPpubsubInitJson()`
- `serializedJson`: pointer to a serialized JSON received on a socket
- `maxEvents`: limit the number of events created to this value. A value of 0 means no limit.

**Return value:** event block pointer
void *C_dfESPebToJson(jsonObjPtr json, C_dfESPeventblock C_eb)

Parameters:

json  
pointer to the object created by C_dfESPpubsubInitJson()

C_eb  
event block pointer

Return value:  
pointer to a serialized JSON

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

Parameters:

xml  
pointer to the object created by C_dfESPpubsubInitXml()

serializedXml  
pointer to serialized XML received on a socket

maxEvents  
the maximum number of events processed. A value of 0 means no limit.

Return value:  
event block pointer

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

Parameters:

C_schema  
pointer to the window schema

dateFormat  
date format for CSV conversions. Set to null to interpret ESP_DATETIME and ESP_TIMESTAMP field values as an integer number of seconds (ESP_DATETIME) or microseconds (ESP_TIMESTAMP) since epoch.

defaultOpcode  
the opcode to insert into events that are build from XML that contain no opcode field

Return value:  
a pointer to an XML object. This pointer is passed to all other XML API functions. The value is null when there is a failure.

void *C_dfESPebToXml(xmlObjPtr xml, C_dfESPeventblock C_eb)

Parameters:

xml  
pointer to the object created by C_dfESPpubsubInitXml()

C_eb  
event block pointer
A C library provides a set of functions to enable client developers to analyze and manipulate the event stream processing objects from the application or server. These functions are a set of C wrappers around a small subset of the methods provided in the C++ Modeling API. With these wrappers, client developers can use C rather than C++. Examples of these objects are events, event blocks, and schemas. A small sampling of these calls follows. For the full set of calls, see the API reference documentation available at $DFESP HOME/doc/html.

To get the size of an event block:

```c
C_ESP_int32_t eventCnt = C_dfESPeventblock_getSize(eb);
```

To extract an event from an event block:

```c
C_dfESPevent ev = C_dfESPeventblock_getEvent(eb, eventIndx);
```

To create an object (a string representation of schema in this case):

```c
C_ESP_utf8str_t schemaCSV = C_dfESPschema_serialize(schema);
```

To free an object (a vector of strings in this case):

```c
C_dfESPstringV_free(metaVector);
```

### Using the Java Publish/Subscribe API

#### Overview to the Java Publish/Subscribe API

SAS Event Stream Processing and its C publish/subscribe API use the SAS logging library, whereas the Java publish/subscribe API uses the Java logging APIs in the `java.util.logging` package. Please refer to that package for log levels and specifics about Java logging.

The Java publish/subscribe API is provided in two packages. These packages define the following public interfaces:

- `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`

A client can query the Event Stream Processor application or server at any time to discover currently running windows, continuous queries, and projects in various granularities. This information is returned to the client in
the form of a list of strings that represent names. This list can be used to build URL strings to pass to subscriberStart() or publisherStart().

Note: If an event stream publishing URL that is passed to the Java API contains a ?username parameter for authentication, you must include cas-client-*.jar in your classpath.

The parameters and usage for the Java publish/subscribe API are the same as for the equivalent calls for the C publish/subscribe API.

The C API references and Java interface references are available at $DFESP_HOME/doc/html.

Using High-Level Publish/Subscribe Methods

The following high-level publish/subscribe methods are defined in the following interface reference: com.sas.esp.api.pubsub.clientHandler.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean init(Level level)</td>
<td>Initialize publish/subscribe services</td>
</tr>
<tr>
<td>dfESPclient publisherStart(String serverURL, clientCallbacks userCallbacks, Object ctx)</td>
<td>Start a publisher.</td>
</tr>
<tr>
<td>dfESPclient subscriberStart(String serverURL, clientCallbacks userCallbacks, Object ctx)</td>
<td>Start a subscriber</td>
</tr>
<tr>
<td>boolean connect(dfESPclient client)</td>
<td>Connect to the Event Stream Processor application or server</td>
</tr>
<tr>
<td>boolean publisherInject((dfESPclient client, dfESPeventblock eventblock)</td>
<td>Publish event blocks</td>
</tr>
<tr>
<td>ArrayList&lt;String&gt; queryMeta(String queryURL)</td>
<td>Query model metadata</td>
</tr>
<tr>
<td>ArrayList&lt;String&gt; getModel(String queryURL)</td>
<td>Query model windows and their edges</td>
</tr>
<tr>
<td>boolean disconnect (dfESPclient client, boolean block)</td>
<td>Disconnect from the event stream processor</td>
</tr>
<tr>
<td>boolean stop (dfESPclient client, boolean block)</td>
<td>Stop a subscriber or publisher</td>
</tr>
<tr>
<td>void shutdown ()</td>
<td>Shutdown publish/subscribe services</td>
</tr>
<tr>
<td>boolean setBufferSize(dfESPclient client, int mbytes)</td>
<td>Change the default socket read and write buffer size</td>
</tr>
<tr>
<td>dfESPclient GDsubscriberStart (String serverURL, clientCallbacks userCallbacks, Object ctx, String config_file)</td>
<td>Start a guaranteed delivery subscriber</td>
</tr>
<tr>
<td>dfESPclient GDpublisherStart (String serverURL, clientCallbacks userCallbacks, Object ctx, String config_file)</td>
<td>Start a guaranteed delivery publisher</td>
</tr>
</tbody>
</table>
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long GDpublisherGetID()</td>
<td>Get a sequentially unique ID to write to an event block to be published using guaranteed delivery</td>
</tr>
<tr>
<td>boolean GDsubscriberAck(dfESPclient client, dfESPeventblock eventblock)</td>
<td>Trigger a guaranteed delivery acknowledgment</td>
</tr>
<tr>
<td>boolean persistModel(String hostportURL, String persistPath)</td>
<td>Instruct a running engine to persist its current state to disk</td>
</tr>
<tr>
<td>boolean quiesceProject(String projectURL, dfESPclient client)</td>
<td>Instruct a running engine to quiesce a specific project in the model.</td>
</tr>
<tr>
<td>boolean subscriberMaxQueueSize(String serverURL, int maxSize, boolean block)</td>
<td>Configure the maximum size of the queues in the event stream processing server that are used to enqueue event blocks sent to subscribers. Set <code>block</code> to 1 to wait for queue size to fall below <code>maxSize</code>, else disconnect the client.</td>
</tr>
<tr>
<td>boolean pingHostPort(String hostportURL)</td>
<td>Pings a running engine to see whether the specified publish/subscribe port is open. Also, exchanges and verifies a magic number in order to confirm that the open port is a publish/subscribe port.</td>
</tr>
<tr>
<td>boolean setTokenLocation(String tokenLocation)</td>
<td>Sets the location of the file in the local file system that contains the OAuth token required for authentication by the publish/subscribe server.</td>
</tr>
</tbody>
</table>

For more information, see `$DFESP_HOME/doc/html/index.html`. Search the Classes page for `client handler`.

### Using Methods That Support Google Protocol Buffers

The following methods support Google Protocol Buffers. They are defined in this interface reference: `com.sas.esp.api.pubsub.protobufInterface`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean init(String fileDescriptorSet, String msgName, dfESPschema schema, EventOpcodes defaultOpcode)</td>
<td>Initialize the library that supports Google Protocol Buffers.</td>
</tr>
<tr>
<td>dfESPeventblock protobufToEb(byte[] serializedProtobuf)</td>
<td>Convert a protobuf message to an event block.</td>
</tr>
<tr>
<td>byte[] ebToProtobuf(dfESPeventblock eb, int index)</td>
<td>Convert an event in an event block to a protobuf message.</td>
</tr>
</tbody>
</table>

For more information, see Publish/Subscribe API Support for Google Protocol Buffers.
Using User-supplied Callback Functions

The `com.sas.esp.api.pubsub.clientCallbacks` interface reference defines the signatures of the user-supplied callback functions. There currently are three functions:

- the subscribed event block handler
- the publish/subscribe failure handler
- the guaranteed delivery ACK-NACK handler

The subscribed event block handler is used only by subscriber clients. It is called when a new event block from the application or server arrives. After processing the event block, the client is responsible for freeing it by calling `eventblock_destroy()`.

The signature of this user-defined callback is as follows:

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

- `eventBlock` is the event block just read
- `schema` is the schema of the event for client processing
- `ctx` is an optional context pointer for maintaining call state

The second callback function for publish/subscribe client error handling is optional for both subscriber and publisher clients. If supplied (that is, not NULL), it is called for every occurrence of an abnormal event within the client services, such as an unsolicited disconnect. This enables the client to gracefully handle and possibly recover from publish/subscribe services errors. The signature for this callback function is below where

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

- `failure` is either `pubsubFail_APIFAIL`, `pubsubFail_THREADFAIL`, or `pubsubFail_SERVERDISCONNECT`.
- `code` provides the specific code of the failure.
- `ctx` is an optional context pointer to a state data structure.

The guaranteed delivery ACK-NACK handler is invoked to provide the status of a specific event block, or to notify the publisher that all subscribers are connected and publishing can begin. The signature for this callback function is as follows:

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

- `eventBlockStatus` is either `ESP_GD_READY`, `ESP_GD_ACK`, or `ESP_GD_NACK`
- `eventBlockID` is the ID written to the event block prior to publishing
- `ctx` is an optional context pointer to a state data structure

Using Alternative Transport Libraries for Java Clients

Alternative transport libraries enable a Java publish/subscribe client application to send and receive event blocks through a mechanism other than a direct TCP/IP connection to the client:

- Rabbit MQ Java libraries enable sending and receiving through the Rabbit MQ server.
Solace Java libraries enable sending and receiving through the Solace appliance.

Tervela Java libraries enable sending and receiving through the Tervela appliance.

Kafka Java libraries enable sending and receiving through the Kafka cluster.

These libraries enable the Java equivalent of the C publish/subscribe API method, substituting an alternate transport. When the engine is configured for 1 + N-Way Failover using Rabbit MQ, Solace, Tervela, or Kafka, Java publish/subscribe clients must use the corresponding client library to guarantee successful failover.

To substitute one of these libraries in your Java publish/subscribe client application, insert the corresponding JAR filename in front of `dfx-esp-api.jar` in your classpath, as shown here:

- For Rabbit MQ, the JAR file is `dfx-esp-rabbitmq-api.jar`.
- For Solace, the JAR file is `dfx-esp-solace-api.jar`.
- For Tervela, the JAR file is `dfx-esp-tervela-api.jar`.
- For Kafka, the JAR file is `dfx-esp-kafka-api.jar`.

When you are using the Rabbit MQ library, you must also install the native Rabbit MQ Java client libraries (`rabbitmq-client.jar`) on your system. Obtain them at http://www.rabbitmq.com/java-client.html. Then add `rabbitmq-client.jar` to your classpath along with `commons-configuration-1.10.jar`, `commons-lang-2.6.jar`, and `commons-logging-1.2.jar` from `$DFESP_HOME/lib`.

When you are using the Kafka library, you must also install the native Kafka Java client libraries (`kafka-clients-*.jar`) on your system. Obtain them at http://kafka.apache.org/downloads.html. Then add `kafka-clients-*.jar` to your classpath along with `commons-configuration-1.10.jar`, `commons-lang-2.6.jar`, and `commons-logging-1.2.jar` from `$DFESP_HOME/lib`.

Your current working directory must also include a corresponding configuration file as shown here:

- For RabbitMQ this file must be named `rabbitmq.cfg`.
- For Solace, it must be named `solace.cfg`.
- For Tervela, it must be named `client.config`.
- For Kafka, it must be named `kafka.cfg`.

Here is a sample configuration file for RabbitMQ:

```java
{
    rabbitmq =
    {
        host = "my.machine.com";
        port = "5672";
        exchange = "SAS";
        userid = "guest";
        password = "guest";
        passwordencrypted = false;
    }

    sas =
    {
        buspersistance = false;
        queuename = "subpersist";
        protobuf = true;
        protobuf = "./GpbHistSimFactory.desc";
        protomsg = "GpbTrade";
        noreplay = false;
    }
}
```
Note: If `passwordencrypted = true`, the value in `password` must be the encrypted password generated by this OpenSSL command:

```bash
    echo "password" | openssl enc -e -aes-128-cbc -a -salt
                        -pass pass:"SASespRMQclientUsedByUser=userid"
```

The `buspersistence` and `queuename` parameters mean different things for publishers and subscribers.

- For a publisher, `queuename` is always ignored. If `buspersistence = false`, messages are sent in non-persistent delivery mode. Otherwise, delivery mode is persistent.

- For a subscriber, `buspersistence = false` means that all queues and exchanges created by the client are non-durable and auto-delete and that the `queuename` parameter is ignored. If `buspersistence = true`, all exchanges and queues are durable and not auto-delete and the `queuename` in the durable receive queue is fixed.

The `noreplay` parameter is false by default. When set to true, received messages are acknowledged even when `buspersistence` is enabled.

Here is a sample configuration file for Solace:

```json
{
    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;
    }
}
```

Note: If `passwordencrypted = true`, the value in `session.password` must be the encrypted password generated by this OpenSSL command:

```bash
    echo "session.password" | openssl enc -e -aes-128-cbc -a -salt
                        -pass pass:"SASespSOLclientUsedByUser=session.username"
```

Here is a sample configuration file for Tervela:

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

Note: If `PASSWORDENCRYPTED = 1`, the value in `PASSWORD` must be the encrypted password generated by this OpenSSL command:

```bash
    echo "PASSWORD" | openssl enc -e -aes-128-cbc -a -salt
                        -pass pass:"SASespTVaclientUsedByUser=USERNAME"
```
Here is a sample configuration file for Kafka:

```json
{
  kafka = {
    hostport = "kafkahost:9092"
    partition = "0"
    initialoffset = "largest"
    groupid = "mygroup"
  }
  sas = {
    protobuf = false;
    descfile = "/GpbHistSimFactory.desc"
    protomsg = "GpbTrade"
  }
}
```

### Using the Python Publish/Subscribe API

The SAS Event Stream Processing Python publish/subscribe API was developed and tested using Python 2.7.x. Other versions might work, but they are untested and unsupported.

The API is implemented using the Python ctypes foreign function library. This library wraps the C publish/subscribe API in Python. Thus, the Python Publish/Subscribe API definition mirrors the C Publish/Subscribe API. Functions have the same names as their C counterparts, but with “C_dfESPpubsub” removed from the name.

The API is defined and documented in two files: pubsubApi.py and modelingApi.py. You can find these files in `$DFESP_HOME/lib` on Linux systems and in `%DFESP_HOME%\lib` on Windows systems.

You can find examples of publisher and subscriber clients written in Python in `$DFESP_HOME/examples` on Linux systems and in `%DFESP_HOME%\examples` on Windows systems.

The `Init()` function does the following:

- loads the required C API libraries
- sets the Python function pointers
- creates an ESP logger instance
- calls the C API `C_dfESPpubsubInit()` function with the logging level and logging configuration file path passed to the Python `Init()` function
- returns the return value returned by `C_dfESPpubsubInit()`

`modelingApi.py` contains the following extra functions that do not wrap a C API function:

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

`getLogger()` returns the Python logger instance created by `Init()`. A Python client can obtain the logger instance like this:

```python
logger = logging.getLogger()
logger.addHandler(modelingApi.getLogger())
```
The logger defines internal function pointers to the standard ESP logging functions, so logged messages look like messages logged by an ESP server or C client. Messages can be logged at various levels by calling `logger.level(string)`, where `level` is `debug`, `info`, `warning`, `error`, or `critical`. These levels map to SAS Stream Processing logging levels `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL` respectively.

The `arrayGet*` functions are used to extract entries in an array built by a C function and returned by a Python function, such as `SchemaGetNames()`. This extraction is different from processing a vector built by a C function, such as that returned by `QueryMeta()`. Entries in such a vector can be obtained by calling `StringVGet()`.

---

**Publish/Subscribe API Support for Google Protocol Buffers**

**Overview to Publish/Subscribe API Support for Google Protocol Buffers**

SAS Event Stream Processing provides a library to support Google Protocol Buffers. This library provides conversion methods between an event block in binary format and a serialized Google protocol buffer (protobuf).

To exchange a protobuf with an event stream processing server, a publish/subscribe client using the standard publish/subscribe API can call `C_dfESPpubsubInitProtobuff()` to load the library that supports Google Protocol Buffers. Then a publisher client with source data in protobuf format can call `C_dfESPprotobufToEb()` to create event blocks in binary format before calling `C_dfESPpublisherInject()`. Similarly, a subscriber client can convert a received events block to a protobuf by calling `C_dfESPebToProtobuff()` before passing it to a protobuf handler.

**Note:** The server side of an event stream processing publish/subscribe connection does not support Google Protocol Buffers. It continues to send and receive event blocks in binary format only.

The SAS Event Stream Processing Java publish/subscribe API contains a protobuf JAR file that implements equivalent methods for Java publish/subscribe clients. In order to load the library that supports Google Protocol Buffers, you must have installed the standard Google Protocol Buffers run-time library. The SAS Event Stream Processing run-time environment must be able to find this library. For Java, you must have installed the Google protobuf JAR file and have included it in the run-time class path.

A publish/subscribe client connection exchanges events with a single window using that window’s schema. Correspondingly, a protobuf enabled client connection uses a single fixed protobuf message type, as defined in a message block in a `.proto` file. The library that supports Google Protocol buffers dynamically parses the message definition, so no precompiled message-specific classes are required. However, the Java library uses a `.desc` file instead of a `.proto` file, which requires you to run the Google protoc compiler on the `.proto` file in order to generate a corresponding `.desc` file.

For C clients, the name of the `.proto` file and the enclosed message are both passed to the library that supports Google Protocol Buffers in the `C_dfESPpubsubInitProtobuff()` call. This call returns a protobuf object instance, which is then passed in all subsequent protobuf calls by the client. This instance is specific to the protobuf message definition. Thus, it is valid as long as the client connection to a specific window is up. When the client stops and restarts, it must obtain a new protobuf object instance.

For Java clients, the process is slightly different. The client creates an instance of a `dfESPprotobuf` object and then calls its `init()` method. Subsequent protobuf calls are made using this object’s methods, subject to the same validity scope described for the C++ protobuf object.

Conversion between a binary event block and a protobuf is accomplished by matching fields in the protobuf message definition to fields in the schema of the associated publish/subscribe window. Ensure that the protobuf message definition and the window schema are compatible. When the protobuf message definition...
contains optional fields, ensure that they are included in the window schema. If a received protobuf message is missing an optional field, the corresponding field in the event is set to null. Conversely, when building a protobuf and a field of an event contains null, the corresponding protobuf field is left unset, and therefore must be defined as optional in the .proto file.

The following mapping of event stream processor to Google Protocol Buffer data types are supported:

<table>
<thead>
<tr>
<th>Event Stream Processor Data Type</th>
<th>Google Protocol Buffer Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP_DOUBLE</td>
<td>TYPE_DOUBLE, TYPE_FLOAT</td>
</tr>
<tr>
<td>ESP_INT64</td>
<td>TYPE_INT64, TYPE_UINT64, TYPE_FIXED64, TYPE_SFIXED64, TYPE_SINT64</td>
</tr>
<tr>
<td>ESP_INT32</td>
<td>TYPE_INT32, TYPE_UINT32, TYPE_FIXED32, TYPE_SFIXED32, TYPE_SINT32, TYPE_ENUM</td>
</tr>
<tr>
<td>ESP_UTF8STR</td>
<td>TYPE_STRING</td>
</tr>
<tr>
<td>ESP_DATETIME</td>
<td>TYPE_INT64</td>
</tr>
<tr>
<td>ESP_TIMESTAMP</td>
<td>TYPE_INT64</td>
</tr>
</tbody>
</table>

Other mappings are currently unsupported.

Converting Nested and Repeated Fields in Protocol Buffer Messages to an Event Block

Provided that they are supported, you can repeat the message fields of a protobuf. A message field of TYPE_MESSAGE can be nested, and possibly repeated as well. All of these cases are supported when converting a protobuf message to an event block, observing the following policies:

- A protobuf message that contains nested messages requires that the corresponding schema be a flattened representation of the protobuf message. For example, a protobuf message that contains three fields where the first is a nested message with four fields, the second is not nested, and the third is a nested message with two fields requires a schema with 4 + 1 + 2 = 7 fields. Nesting depth is unbounded.

- A single protobuf message is always converted to an event block that contains a single event, provided that no nested message field is repeated.

- When a protobuf message has a non-message type field that is repeated, all the elements in that field are gathered into a single comma-separated string field in the event. For this reason, any schema field that corresponds to a repeated field in a protobuf message must have type ESP_UTF8STR, regardless of the field type in the protobuf message.
A protobuf message that contains nested message fields that are repeated is always converted to an event block that contains multiple events. There is one event for each element in every nested message field that is repeated.

**Converting Event Blocks to Protocol Buffer Messages**

Converting an event block to a protobuf is conceptually similar to converting nested and repeated fields in a protobuf to an event block, but the process requires more code. Every event in an event block is converted to a separate protobuf message. For this reason, the C\_dfESPebToProtobuff() library call takes an index parameter that indicates which event in the event block to convert. The library must be called in a loop for every event in the event block.

The conversion correctly loads any nested fields in the resulting protobuf message. Any repeated fields in the resulting protobuf message contain exactly one element, because event blocks do not support repeated fields.

**Note:** Event block to protobuf conversions support only events with the Insert opcode, because event opcodes are not copied to protobuf messages. Conversions of protobuf to event blocks use the opcode that is specified in the C\_dfESPpubsubInitProtobuff() function or Java\_init() function. When protobufs are invoked by a connector or adapter, the opcode is Insert unless the connector or adapter is configured to use Upsert.

**Support for Transporting Google Protocol Buffers**

Support for Google Protocol Buffers is available when you use the connectors and adapters that are associated with the following message buses:

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

These connectors and adapters support transport of a protobuf through the message bus, instead of binary event blocks. This allows a third-party publisher or subscriber to connect to the message bus and exchange a protobuf with an engine without using the publish/subscribe API. The protobuf message format and window schema must be compatible.

The connector or adapter requires configuration of the .proto file and message name through the protofile and protomsg parameters. The connector converts a protobuf to and from an event block using the SAS Event Stream Processing library that supports Google Protocol Buffers. In addition, the C and Java Solace publish/subscribe clients also support Google Protocol Buffers when configured to do so in the solace.cfg client configuration file. Similarly, C RabbitMQ and Kafka publish/subscribe clients support Google Protocol Buffers when they are configured to do so in the rabbitmq.cfg or kafka.cfg client configuration file. A protobuf enabled client publisher converts an event block to a protobuf to transport through the message bus to a third-party consumer of Google Protocol Buffers. Likewise, a protobuf-enabled client subscriber receives a protobuf from the message bus and converts it to an event block.
Publish/Subscribe API Support for JSON Messaging

Overview

SAS Event Stream Processing provides a C library to support JSON messaging. The library provides conversion methods between an event block in binary format and a serialized JSON message.

To exchange a JSON message with an event stream processing server, a publish/subscribe client that uses the standard publish/subscribe API can call `C_dfESPpubsubInitJson()` to load the library that supports JSON. Then a publisher client with source data in JSON format can call `C_dfESPjsonToEb()` to create event blocks in binary format. It can then call `C_dfESPpublisherInject()`.

Similarly, a subscriber client can convert a received events block to a JSON message by calling `C_dfESPebToJson()` before passing it to a JSON message handler.

Note: The server side of an event stream processing publish/subscribe connection does not support JSON messages. It continues to send and receive event blocks in binary format only.

A publish/subscribe client connection exchanges events with a single window using that window’s schema. Correspondingly, a JSON-enabled client connection exchanges messages with a fixed JSON schema. However, there is no static definition of this schema. A schema mismatch between a JSON message and the related window schema is detected only at run time.

The `C_dfESPpubsubInitJson()` call returns a JSON object instance, which is then passed in all subsequent JSON calls by the client. This object instance is valid only while the client connection to a specific window is up. When the client stops and restarts, it must obtain a new JSON object instance.

Fundamentally, a single JSON message maps to a single event. However, a JSON message can contain multiple events when you enclose them within a JSON array.

Converting Nested Fields in JSON Messages to an Event Block

The window schema must be a flattened representation of the JSON event schema, where window field names are a concatenation of any nested JSON tag names, separated by underscores. When the input JSON contains fields that are intentionally missing in the source window schema and should be ignored, call the `C_dfESPignoreMissingSchemaFields()` method before building event blocks. This prevents the library from logging errors.

Within a JSON event schema, unlimited nesting of arrays and objects is supported.

When a JSON event contains an array field, all the elements in that array are gathered into a single comma-separated string field in the event. For this reason, any schema field that corresponds to an array field in a JSON event must have type `ESP_UTF8STR`, regardless of the field type within the JSON event schema.

The event built from a JSON event always has the Insert opcode. The exception is when the JSON event contains a field named `opcode`. In that case, the value of that field is used to set the event opcode. When other JSON fields do not match fields in the source window schema, the inject operation fails.
Valid Values for the Opcode Field in a JSON Event

<table>
<thead>
<tr>
<th>Value</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Insert</td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>Update</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Delete</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>Upsert</td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>Safe delete</td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

By default, the event block is type= normal. When events in an event block array are contained within an additional array (that is, enclosed in an additional pair of brackets), the event block is type= transactional.

You can configure the JSON library to filter out JSON values that do not contain a configured substring. In this case, the corresponding event is silently dropped. You can enable such filtering by configuring the matchsubstrings parameter and supplying a comma separated string of key:substring pairs. For example, a configured value of "foo:bar" means that a JSON key "foo" that does not contain "bar" in its value does not generate an event stream processing event. When you configure multiple substrings per key, input JSON data must contain only one of the configured substrings in order to generate an event.

When no JSON field (or combination of fields) can easily be used as a key for the source window, add one or both of the following key fields to the source window schema:

```
"eventindex*:int64,adapterindex*:string"
```

When present in the source window schema, the library fills in these fields as follows:

- **eventindex**: a 64-bit atomic integer that is incremented for every event, and is guaranteed unique across multiple instances of the library in a process space.
- **adapterindex**: a GUID string that is guaranteed unique for instances of the library running in different process spaces.

The combination of these values allows multiple users of the library in different processes to inject events into a single source window without duplicating keys.

When you require static JSON values to be included in all injected events, pass it to the `C_dfESPaddStaticJson()` method. All subsequent events include fields built from that JSON.

**Converting Event Blocks to JSON Messages**

All JSON events that are created from an event contain an opcode field. Valid values are listed in Valid Values for the Opcode Field in a JSON Event.

Because the input is an event block, the resulting JSON message always has an array as its root object. Each array entry represents a single event.

Data variable type **ESP_MONEY** is not supported.
Data variable types `ESP_TIMESTAMP` and `ESP_DATETIME` are converted to a JSON string that contains the CSV representation of the field value.

**Support for Transporting JSON Messages**

Support for JSON messaging is available when you use the connectors and adapters associated with the following message buses:

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

These connectors and adapters support transport of JSON encoded messages through the message bus, instead of binary event blocks. This enables a third-party publisher or subscriber to connect to the message bus and exchange JSON messages with an engine without using the publish/subscribe API.

No message-format configuration is required. If the JSON schema and window schema are incompatible, a publisher that converts JSON to event blocks fails when an event block is injected. The connector converts JSON to and from an event block using the SAS Event Stream Processing library that supports JSON conversion.

C RabbitMQ, Solace Systems, and Kafka publish/subscribe clients support JSON when configured to do so in the `rabbitmq.cfg`, `solace.cfg`, or `kafka.cfg` client configuration files. A JSON-enabled client publisher converts an event block to a JSON message to transport through the message bus to a third-party consumer of JSON messages. Likewise, a JSON enabled client subscriber receives a JSON message from the message bus and converts it to an event block.

**Publish/Subscribe API Support for XML Messaging**

SAS Event Stream Processing provides a C library to support XML messaging. This library is analogous to the JSON messaging library described previously. It provides the following API functions:

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

These functions perform the same operations as the corresponding JSON functions in the JSON library.
Guaranteed Delivery

Overview to Guaranteed Delivery

The C, Java, and Python Publish/Subscribe APIs support guaranteed delivery between a single publisher and multiple subscribers. Guaranteed delivery assumes a model where each event block that is published into a source window generates exactly one event block in a subscribed window. This one block in, one block out principle must hold for all published event blocks. The guaranteed delivery acknowledgment mechanism is not aware of the event processing performed by the model.

When a publish or subscribe connection is started, a client is established to perform various publish/subscribe activities. When a publish connection is started, the number of guaranteed subscribers required to acknowledge delivery of its event blocks is specified. The time-out value used to generate negative acknowledgments upon non-receipt from all expected subscribers is also specified. Every event block injected by the publisher contains a unique 64-bit ID set by the publisher. This ID is passed back to the publisher from the publish client with every acknowledgment or negative acknowledgment in a publisher user-defined callback function. The function is registered when the publish client is started.

When a subscribe connection is started, the subscribe client is passed a set of guaranteed delivery publishers as a list of host and port entries. The client then establishes a TCP connection to each publisher on the list. This connection is then used only to transport acknowledgments specific to this publisher/subscriber pair. The subscriber calls a new Publish/Subscribe API function to trigger an acknowledgment.

Event blocks contain new host, port, and ID fields. All event blocks are uniquely identified by the combination of these fields. This enables subscribers to identify duplicate (that is, resent) event blocks.
Guaranteed Delivery Data Flow Diagram
Please note the following:

- Publishers and subscribers that do not use the guaranteed-delivery-enabled API functions are implicitly guaranteed delivery disabled.
- Guaranteed delivery subscribers can be mixed with non-guaranteed delivery subscribers.
- A guaranteed delivery-enabled publisher might wait to begin publishing until a READY callback has been received. This indicates that its configured number of subscribers have established their acknowledgment connections back to the publisher.
- Event blocks received by a guaranteed-delivery-enabled subscriber as a result of a snapshot generated by the engine are not acknowledged.
- Under certain conditions, subscribers receive duplicate event blocks. These conditions include the following:
  - A publisher begins publishing before all related subscribers have started. Any started subscriber can receive duplicate event blocks until the number of started subscribers reaches the number of required acknowledgments passed by the publisher.
  - A guaranteed delivery-enabled subscriber disconnects while the publisher is publishing. This triggers the same scenario described previously.
  - A slow subscriber causes event blocks to time-out, which triggers a negative acknowledgment to the publisher. In this case all subscribers related to the publisher receives any resent event blocks, including those that have already called `C_dfESPGDsubscriberAck()` for those blocks.
- If a guaranteed delivery-enabled subscriber fails to establish its acknowledgment connection, it retries at a configurable rate up to a configurable maximum number of retries.
- Suppose that a guaranteed delivery-enabled publisher injects an event block that contains an ID, and that the ID is present in the publish client's not acknowledged-ID list. In that case, the inject call is rejected by the publish client. The ID is cleared from the list when the publish client passes it to the ACK/NACK callback function of the new publisher.

### Guaranteed Delivery Success Scenario

In the context of guaranteed delivery, the publisher and subscriber are customer applications that are the endpoints in the data flow. The subscribe and publish clients are event stream processing code that implements the publish/subscribe API calls made by the publisher and subscriber.

The flow of a guaranteed delivery success scenario is as follows:

1. The publisher passes an event block to the publish client, where the ID field in the event block has been set by the publisher. The publish client fills in the host-port field, adds the ID to its unacknowledged ID list, and injects it to the engine.
2. The event block is processed by the engine and the resulting Inserts, Updates, or Deletes on subscribe windows are forwarded to all subscribe clients.
3. A guaranteed delivery-enabled subscribe client receives an event block and passes it to the subscriber by using the standard subscriber callback.
4. Upon completion of all processing, the subscribers call a new API function with the event block pointer to trigger an acknowledgment.
5. The subscribe client sends the event block ID on the guaranteed delivery acknowledgment connection that matches the host or port in the event block, completely bypassing the engine.
6. Upon receipt of the acknowledgment, the publish client increments the number of acknowledgments received for this event block. If that number has reached the threshold passed to the publish client at start-up, the
publish client invokes the new guaranteed delivery callback with parameters acknowledged and ID. It removes the ID from the list of unacknowledged IDs.

### Guaranteed Delivery Failure Scenarios

There are three failure scenarios for guaranteed delivery flows:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
</table>
| Event Block Time-out                  | - An event block-specific timer expires on a guaranteed-delivery-enabled publish client, and the number of acknowledgments received for this event block is below the required threshold.  
  - The publish client invokes the new guaranteed delivery callback with parameters `NACK` and `ID`. No further retransmission or other attempted recovery by the publish client or subscribe client is undertaken for this event block. The publisher most likely backs out this event block and resends.  
  - The publish client removes the ID from the list of unacknowledged IDs. |
| Invalid Guaranteed Delivery Acknowledged Connect Attempt | - A guaranteed-delivery-enabled publish client receives a connect attempt on its guaranteed delivery acknowledged server but the number of required client connections has already been met.  
  - The publish client refuses the connection and logs an error message.  
  - For any subsequent event blocks received by the guaranteed delivery-enabled subscribe client, an error message is logged. |
| Invalid Event Block ID                | - A guaranteed-delivery-enabled publisher injects an event block that contains an ID already present in the publish client’s unacknowledged ID list.  
  - The inject call is rejected by the publish client and an error message is logged. |

### Additions to the Publish/Subscribe API for Guaranteed Delivery

The publish/subscribe API provides the following methods to implement guaranteed delivery sessions:

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

For more information, see . For publish/subscribe operations without a guaranteed delivery version of the function, call the standard publish/subscribe API function.
## Configuration File Contents

The publish client and subscribe client reads a configuration file at start-up to get customer-specific configuration information for guaranteed delivery. The format of both of these files is as follows.

### Guaranteed Delivery-enabled Publisher Configuration File Contents

- **Local port number for guaranteed delivery acknowledgment connection server.**
- **Time-out value for generating negative acknowledgments, in seconds.**
- **Number of received acknowledgments required within time-out period to generate positive instead of negative acknowledgments.**

**File format:**

```
GDpub_port=<port>
GDpub_timeout=<timeout>
GDpub_numSubs=<number of subscribers generating acknowledged>
```

### Guaranteed Delivery-enabled Subscriber Configuration File Contents

- **List of guaranteed delivery-enabled publisher host or port entries. Each entry contains a host:port pair corresponding to a guaranteed delivery-enabled publisher from which the subscriber wishes to receive guaranteed delivery event blocks.**
- **Acknowledgment connection retry interval, in seconds.**
- **Acknowledgment connection maximum number of retry attempts.**

**File Format:**

```
GDsub_pub=<host:port>
GDsub_retryInt=<interval>
GDsub_maxRetries=<max>
```