Estimating Memory Consumption

Overview

SAS Event Stream Processing is designed to optimize low latency and high throughput. Knowing how much memory an event consumes can help you size the computer resources required to run a SAS Event Stream Processing model.

An event is an instance of the dfESPevent class. This class maintains an event as a tuple of components: a meta event portion that consists of certain metadata (opcode, flag) and a reference-counted data event portion that contains the binary packed field data. Open DFESP_HOME/doc/html/index.html in a web browser to access information about this class.

Assume the following:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of fields in the event</td>
</tr>
<tr>
<td>PB</td>
<td>Packed binary size of all the fields in the event. Assume no padding and that strings are NULL-terminated.</td>
</tr>
<tr>
<td>16</td>
<td>Size of the event class, which consists of a meta event portion and a data event portion</td>
</tr>
<tr>
<td>32</td>
<td>Size of meta event memory consumption</td>
</tr>
<tr>
<td>[4 \ast (N+5) + PB]</td>
<td>Size of data event memory consumption</td>
</tr>
</tbody>
</table>

The total memory consumed by an event is the sum of these components: \(16 + 32 + [4 \ast (N+5) + PB]\).

The final memory allocation is rounded up to 16 bytes. That rounding affects only the size of the data event storage.

Additional overhead is required for indexes, and the amount of overhead depends on the compiler libraries (C++ and Boost). Minimally, you need an additional 32 bytes per element for an index type of pi_RBTREE and another 36 bytes per element for an index type of pi_HASH.
Memory Consumption for Two Stateful Source Windows That Feed a Stateful Join

Suppose that you have two Source windows, src_win1 and src_win2, that are stateful. These windows feed a stateful Join window, join_win. The Join window contains reference_counted copies of both Source windows in addition to its own set of indexes.

A reference-counted copy of an event can be included in the total memory consumption estimate for the window: \((16 + 32 + 0)\) per event.

Note: The data event portion is the same in the left and right indexes of the join as it is in the Source window, hence the 0 in the formula.

Let:

- \(NR1\) Number of events streaming through src_win1
- \(N1\) Number of fields in src_win1
- \(PB1\) Size of binary packed data for src_win1
- \(NR2\) Number of events streaming through src_win2
- \(N2\) Number of fields in src_win2
- \(PB2\) Size of binary packed data for src_win2
- \(NR3\) Number of events streaming through join_win
- \(N2\) Number of fields in the join
- \(PB3\) Size of binary packed data for the join

Assume that you use the basic formula to compute memory for the join windows and use pi_RBTREE indexes. You derive the following [memory for events] + [memory for indexes]:

- \(\text{src_win1} = [NR1 \times (16 + 32 + (4 \times (N1+5) + PB1))] + [NR1 \times 32]\)
- \(\text{src_win2} = [NR2 \times (16 + 32 + (4 \times (N2+5) + PB2))] + [NR2 \times 32]\)
- \(\text{join_left} = [NR1 \times (16 + 32 + 0)] + [NR1 \times 32]\)
- \(\text{join_right} = [NR2 \times (16 + 32 + 0)] + [NR2 \times 32]\)
- \(\text{join_win} = [NR3 \times (16 + 32 + (4 \times (N3+5) + PB3))] + [NR3 \times 32]\)

The total memory consumed by this stateful join is the sum of these components: src_win1 + src_win2 + join_left + join_right + join_win.

Additional Considerations

Consider these additional points when estimating memory consumption:
Setting `pubsub="auto"` at the project level significantly increases the memory used by the project. Each stateful window has a publish/subscribe index that is automatically created. These windows use reference–counted copies of the windows data, but they still consume significant memory.

To avoid needlessly increasing memory consumption, it is recommended that you set `pubsub="manual"`. You then can enable publish/subscribe for the windows that need to support subscriptions.

Adding aggregation, pattern matching, and secondary join indexes to a model complicates the estimation of memory consumption. Expect to consume more memory when deploying these features.

---

**Reducing Model Memory Consumption**

**Overview**

SAS Event Stream Processing trades speed for memory. It can consume more memory than you expect in order to achieve high throughput rates and a consistent computational model. For more information, see “Estimating Memory Consumption”.

Without specific information about the size of each event or how many events are being retained by the ESP server, it is impossible to precisely compute memory consumption. Because SAS Event Stream Processing reference counts events, it is difficult to know whether the data portion of events, the index nodes that hold events, or both dominate memory consumption.

Even so, you can reduce memory consumption by not indexing events (use `pi_EMPTY` indices). The discussion that follows provides use cases with `pi_EMPTY` indexes even when incoming events are a mix of Insert, Upsert, Update, and Delete events.

**State Resolution for Updates, Upserts, and Deletes**

Stateful indexes store events and resolve Updates, Upserts, and Deletes into fully qualified events that can be used in downstream processing. A **fully qualified event** has the following characteristics:

- It has no ambiguity (Upsert is ambiguous) and it carries sufficient information for downstream processing.
- For an Update event, it is the old event and the new event.
- For a Delete event, it is the entire event that is being deleted.

**Using One-to-One Windows to Resolve the Event State**

The following figure depicts how to set up one-to-one windows (one input implies one output) to resolve the event state. The chain, which begins with a set of `pi_EMPTY` windows, terminates with a stateful window that resolves the state of all incoming events. The stateful window can use any other index type, such as `pi_HASH` or `pi_RBTREE`. This final stateful window is typically a Copy, Compute, or Functional window or the lookup side of a non-regenerate join.
The final stateful window that resolves all non-insert opcodes retains all current events (minus deleted events). This means that this final stateful window could be potentially unbounded. To maintain bounded finite memory consumption, do one of the following:

- Ensure that the cardinality of the total set of keys is finite (the number of Updates, Inserts, and Deletes does not need to be finite).
- Make the final stateful window a Copy window with retention. This forces a finite number of events to be retained.

What follows is a project that contains several pi_EMPTY Compute windows.

Here is the Source window:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<engine>
  <projects>
    <project name='project' pubsub='auto' threads='1'>
      <contqueries>
        <contquery name='cq' trace='copyW'>
          <windows>
            <window-source name='srcW' index='pi_EMPTY'>
              <schema>
                <fields>
                  <field name='ID' type='int64' key='true'/>
                  <field name='int1' type='int64'/>
                  <field name='int2' type='int64'/>
                  <field name='int3' type='int64'/>
                  <field name='int4' type='int64'/>
                  <field name='int5' type='int64'/>
                </fields>
              </schema>
            </windows>
          </window-source 'srcW' index='pi_EMPTY'>
          <connectors>
            <connector class='fs' name='pub'>
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
              </properties>
            </connector>
          </connectors>
        </contquery>
      </contqueries>
    </project>
  </projects>
</engine>
```
Here are the Compute windows:

```xml
<window-compute name='computeW1' index='pi_EMPTY'>
  <schema>
    <fields>
      <field name='ID' type='int64' key='true'/>
      <field name='int1' type='int64'/>
      <field name='int2' type='int64'/>
      <field name='int3' type='int64'/>
      <field name='int4' type='int64'/>
      <field name='int5' type='int64'/>
      <field name='comp1' type='int64'/>
    </fields>
  </schema>
  <output>
    <field-expr>int1</field-expr>
    <field-expr>int2</field-expr>
    <field-expr>int3</field-expr>
    <field-expr>int4</field-expr>
    <field-expr>int5</field-expr>
    <field-expr>int1+int2</field-expr>
  </output>
</window-compute>

<window-compute name='computeW2' index='pi_EMPTY'>
  <schema>
    <fields>
      <field name='ID' type='int64' key='true'/>
      <field name='int1' type='int64'/>
      <field name='int2' type='int64'/>
      <field name='int3' type='int64'/>
      <field name='int4' type='int64'/>
      <field name='int5' type='int64'/>
      <field name='comp1' type='int64'/>
      <field name='comp2' type='int64'/>
    </fields>
  </schema>
  <output>
    <field-expr>int1</field-expr>
    <field-expr>int2</field-expr>
    <field-expr>int3</field-expr>
    <field-expr>int4</field-expr>
    <field-expr>int5</field-expr>
    <field-expr>comp1</field-expr>
    <field-expr>int4+int5</field-expr>
  </output>
</window-compute>
```

The following Copy window, which is stateful, resolves all the Update, Upsert, and Delete events that stream through the chain:
The edges connect the windows to one another:

```xml
<edges>
  <edge source='srcW' target='computeW1'/>
  <edge source='computeW1' target='computeW2'/>
  <edge source='computeW2' target='copyW'/>
</edges>
```

Suppose that you run the following input events through the project:

- `i,n,0,1,2,3,4,5`
- `u,n,0,2,3,4,5,6`
- `p,n,0,3,4,5,6,7`
- `d,n,0,0,0,0,0,0`

The project produces the following output, as represented in XML:
<event opcode='insert' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>3</value>
<value name='comp2'>9</value>
<value name='int1'>1</value>
<value name='int2'>2</value>
<value name='int3'>3</value>
<value name='int4'>4</value>
<value name='int5'>5</value>
</event>

<event opcode='updateblock' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>5</value>
<value name='comp2'>11</value>
<value name='int1'>2</value>
<value name='int2'>3</value>
<value name='int3'>4</value>
<value name='int4'>5</value>
<value name='int5'>6</value>
</event>

<event opcode='delete' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>3</value>
<value name='comp2'>9</value>
<value name='int1'>1</value>
<value name='int2'>2</value>
<value name='int3'>3</value>
<value name='int4'>4</value>
<value name='int5'>5</value>
</event>

<event opcode='updateblock' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>7</value>
<value name='comp2'>13</value>
<value name='int1'>3</value>
<value name='int2'>4</value>
<value name='int3'>5</value>
<value name='int4'>6</value>
<value name='int5'>7</value>
</event>

<event opcode='delete' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>5</value>
<value name='comp2'>11</value>
<value name='int1'>2</value>
<value name='int2'>3</value>
<value name='int3'>4</value>
<value name='int4'>5</value>
<value name='int5'>6</value>
</event>

<event opcode='delete' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>7</value>
<value name='comp2'>13</value>
<value name='int1'>3</value>
<value name='int2'>4</value>
<value name='int3'>5</value>
<value name='int4'>6</value>
<value name='int5'>7</value>
</event>

<event opcode='delete' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>3</value>
<value name='comp2'>9</value>
<value name='int1'>1</value>
<value name='int2'>2</value>
<value name='int3'>3</value>
<value name='int4'>4</value>
<value name='int5'>5</value>
</event>

<event opcode='delete' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>5</value>
<value name='comp2'>11</value>
<value name='int1'>2</value>
<value name='int2'>3</value>
<value name='int3'>4</value>
<value name='int4'>5</value>
<value name='int5'>6</value>
</event>

<event opcode='delete' window='project/cq/copyW'>
<value name='ID'>0</value>
<value name='comp1'>7</value>
<value name='comp2'>13</value>
<value name='int1'>3</value>
<value name='int2'>4</value>
<value name='int3'>5</value>
<value name='int4'>6</value>
<value name='int5'>7</value>
</event>
Using a Stateful Local Join Index to Resolve the State

In the past, a streaming join lookup where the lookup table required maintenance through publishing Insert, Update, and Delete events required a large amount of memory. This lookup required a stateful Source window for the lookup side of the join. That stateful Source window fully resolved Insert, Update, and Delete events and fed them to the join. The insert Update and Delete events were applied to the local lookup index in the join. A considerable amount of memory was wasted because of the dual stateful index maintenance (the Source window and the join’s local lookup index).

Beginning with SAS Event Stream Processing 5.2, the local stateful index for the lookup side of the join can resolve Insert, Update, Upsert, and Delete events. This enables the windows that feed into the lookup side of the join to be pi_EMPTY but still contain Insert, Update, Upsert, and Delete events for lookup maintenance.

The following code shows a low memory join:

Note: You must maintain a finite number of events to ensure a bounded memory footprint. The local lookup index for the join maintains all events (minus deleted events) that stream into the lookup side of the join.

Here are the Source windows:

```xml
<Project name='pr_01' pubsub='auto' threads='3' disk-store-path='./'>
  <ContQueries>
    <ContQuery name='cq_01'>
      <Windows>
        <WindowSource name='stream_source' index='pi_EMPTY' insert-only='true'>
          <Schema>
            <Fields>
              <Field name='ID' type='int64' key='true'/>
              <Field name='matchID' type='int64'/>
            </Fields>
          </Schema>
        </WindowSource>
        <StatefulIndex name='pi_EMPTY' left='true' right='true'>
          <JoinWith pi_EMPTY PrimaryIndex />
          <InsertOnly />
          <Matched />
        </StatefulIndex>
      </Windows>
    </ContQuery>
  </ContQueries>
</Project>
```
Here is the Join window:
Here are the connector groups and edges:

```xml
<project-connectors>
  <connector-groups>
    <connector-group name="group1">
      <connector-entry connector='cq_01/join/sub' state='running'/>
      <connector-entry connector='cq_01/lookup_source/pub' state='finished'/>
    </connector-group>
    <connector-group name="group2">
      <connector-entry connector='cq_01/stream_source/pub' state='finished'/>
    </connector-group>
    <connector-group name="group3">
      <connector-entry connector='cq_01/lookup_source/pub1' state='finished'/>
    </connector-group>
    <connector-group name="group4">
      <connector-entry connector='cq_01/stream_source/pub1' state='finished'/>
    </connector-group>
  </connector-groups>
  <edges>
    <edge source='group1' target='group2'/>
    <edge source='group2' target='group3'/>
    <edge source='group3' target='group4'/>
  </edges>
</project-connectors>
```

Four input data files are orchestrated in the following way:
1 The first file contains data that is fed to the lookup side of the join.

   i,n,0, lookup string #0
   i,n,1, lookup string #1
   i,n,2, lookup string #2
   i,n,3, lookup string #3
   i,n,4, lookup string #4
   i,n,5, lookup string #5
   i,n,6, lookup string #6
   i,n,7, lookup string #7
   i,n,8, lookup string #8
   i,n,9, lookup string #9

2 The second file contains streaming data that is fed to the streaming side of the join, and join output is produced.

   i,n,1,1
   i,n,2,2
   i,n,3,9

Here is the initial output of the join:

   I,N, 1,1,lookup string #1
   I,N, 2,2,lookup string #2
   I,N, 3,9,lookup string #9

3 The third file contains a second set of data that is fed to the lookup side of the join. Join maintenance occurs; Inserts, Updates, Upserts, and Deletes are performed on the lookup table.

   u,n,0, NEW lookup string #0
   d,n,5,
   p,n,9, NEW lookup string #9
   d,n,17

4 The fourth file contains a second set of streaming data that is fed to the streaming side of the join. The lookups reflect the previously executed join maintenance.

   i,n,1,1
   i,n,2,2
   i,n,3,9
   i,n,4,5
   i,n,5,0

Here is the output of the join after lookup table maintenance has been applied:

   I,N, 1,1,lookup string #1
   I,N, 2,2,lookup string #2
   I,N, 3,9,NEW lookup string #9
   I,N, 4,5,
   I,N, 5,0,NEW lookup string #0

Using Shared Vectors and Shared Hash Table Data Structures

SAS Micro Analytic Service provides two ways to share data between the modules executing within a user context: shared vectors and shared hash tables. Shared vectors are collections of data values. The values within a vector can have a mix of data types. Shared hash tables are containers of vectors, where the vectors are
stored and accessed by using keys. Both of these structures are thread-safe and lock-free. You can use them to share data across DS2 methods within a SAS Event Stream Processing project.

For more information about these data structures, see *SAS Micro Analytic Service: Programming and Administration Guide*

The following query shows two Source windows streaming events into a Calculate window. Two DS2 methods that use a shared hash table run within a SAS Micro Analytic Service (MAS) module. The MAS module simulates a no-regenerate inner-join. The module runs within a Calculate window and produces joined events.

*Figure 3*  Continuous Query with a SAS Micro Analytic Service Module Sharing a Hash Table

Here is the XML code for the MAS module:

```
<mas-modules>
  <mas-module language="ds2" module="module_1"
    func-names='process_lookup,process_stream'>
    <code>
      <![CDATA[
        ds2_options sas;
        package module_1/overwrite=yes;

        dcl package masstate | st();
        dcl package logger logr('DF.ESP');

        method process_stream2 (varchar(16) _inOpcode,
                                   bigint lookupID,
                                   in_out varchar matchString,
                                   in_out varchar matchDescription,
                                   in_out varchar _outOpcode);
        dcl int rc;
        dcl varchar(32) key;
      ]]>
    </code>
  </mas-module>
</mas-modules>
```
key = lookupID;
_outOpcode='noop';

/* logr.log('d', 'key=$s', key); */
if (0 eq st.get(key)) then do;
    matchString=st.getString(key,0);
    matchDescription=st.getString(key,1);
    _outOpcode=_inOpcode;
end;
/* logr.log('d', 'inOpcode=$s, outOpcode=$s', _inOpcode, _outOpcode); */
end;

method process_lookup (bigint ID, varchar(32) value,
                      varchar(4096) description,
                      in_out varchar _outOpcode);
dcl int rc;
dcl varchar(32) key;

_outOpcode='noop';
key = ID;
rc = st.createVector(key,2); /* Create a new vector */
rc = st.setString(key, 0, value); /* Insert the events data */
rc = st.setString(key, 1, description);
rc = st.put(key); /* Put the vector to the Hash */
rc = st.deleteVector(key); /* Delete the vector */
end;
endpackage;

1 When using a shared hash table, you use the package MASSTATE to create, share, retrieve, and delete data.

2 The method process_stream is designed to insert received events into a shared hash table. It can also handle Update and Delete events by looking at the _inOpcode.

3 Default to noop. That is, produce no event.

4 Set the Opcode upon a match, producing an event.

5 Uncomment this line to write debugging messages to the log.

6 The method process_lookup is designed to look up a string through a lookup ID in a shared hash table. If no match is found, then the _outOpcode is noop, so no event is generated.

   The method as presented handles Insert events. You could modify it to examine the _inOpcode and add, update, or delete an entry from the shared hash table.

Here is the XML code for two Source windows:

<window-source name='src_win_lookup' index='pi_RBTREE'>
    <schema-string>ID*:int64,value:string,description:string</schema-string>
    <connectors>
        <connector class='fs' name='pub'>
            <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
            </properties>
        </connector>
    </connectors>
</window-source>
Here is the XML code for the Calculate window containing the MAS map for the MAS methods:

```xml
<window-calculate name='join' algorithm='MAS'>
  <schema-string>ID*:int64, matchID:int64, matchString:string, matchDescription:string, price:double, quant:int64</schema-string>
  <mas-map>
    <window-map module="module_1" revision="0" source="src_win_lookup" function="process_lookup"/>
    <window-map module="module_1" revision="0" source="src_win_stream" function="process_stream"/>
  </mas-map>
  <connectors>
    <connector class='fs' name='sub'>
      <properties>
        <property name='type'>sub</property>
        <property name='fstype'>csv</property>
        <property name='fsname'>result.csv</property>
        <property name='snapshot'>true</property>
      </properties>
    </connector>
  </connectors>
</window-calculate>
```

Here is the XML code for the edges between the Source windows and the Calculate window:

```xml
<edges>
  <edge source='src_win_lookup' target='join' role='data'/>
  <edge source='src_win_stream' target='join' role='data'/>
</edges>
```

Here is the XML code for the connector groups. You place this code outside the continuous query that contains the Source and Calculate windows.

```xml
<project-connectors>
  <connector-groups>
    <connector-group name="group1">1
      <connector-entry connector='cq_01/join/sub' state='running'/>
      <connector-entry connector='cq_01/src_win_lookup/pub' state='finished'/>
    </connector-group>
    <connector-group name="group2">
      <connector-entry connector='cq_01/src_win_stream/pub' state='finished'/>
    </connector-group>
  </connector-groups>
</project-connectors>
```
A connector group is a container of connector-entry elements. Connector entries can specify the state of a connector.

Now suppose that you stream the following events into the Source window src_win_lookup:

\[i,n,10001,\text{sunw},"\text{Workstation manufacturer}"\]
\[i,n,20001,\text{ibm},"\text{From typewriters to mainframes}"\]

Then you stream the following events into the Source window src_win_stream:

\[i,n,1,10001,101.45,100\]
\[i,n,2,20001,23.5,1000\]
\[i,n,3,20001,10.1,80\]
\[i,n,4,10001,10.2,85\]

The joined data in XML is as follows:

\[
<\text{event opcode='insert' window='project_01/cq_01/join'>}
\langle\text{name='ID'}\rangle1\langle/\text{name}\rangle
\langle\text{name='matchDescription'}\rangle\text{Workstation manufacturer}\langle/\text{name}\rangle
\langle\text{name='matchString'}\rangle\text{sunw}\langle/\text{name}\rangle
\langle\text{name='price'}\rangle101.450000\langle/\text{name}\rangle
\langle\text{name='quant'}\rangle100\langle/\text{name}\rangle
\langle/\text{event}\rangle

<\text{event opcode='insert' window='project_01/cq_01/join'>}
\langle\text{name='ID'}\rangle2\langle/\text{name}\rangle
\langle\text{name='matchDescription'}\rangle\text{From typewriters to mainframes}\langle/\text{name}\rangle
\langle\text{name='matchString'}\rangle\text{ibm}\langle/\text{name}\rangle
\langle\text{name='price'}\rangle23.500000\langle/\text{name}\rangle
\langle\text{name='quant'}\rangle1000\langle/\text{name}\rangle
\langle/\text{event}\rangle

<\text{event opcode='insert' window='project_01/cq_01/join'>}
\langle\text{name='ID'}\rangle3\langle/\text{name}\rangle
\langle\text{name='matchDescription'}\rangle\text{From typewriters to mainframes}\langle/\text{name}\rangle
\langle\text{name='matchString'}\rangle\text{ibm}\langle/\text{name}\rangle
\langle\text{name='price'}\rangle20.100000\langle/\text{name}\rangle
\langle\text{name='quant'}\rangle80\langle/\text{name}\rangle
\langle/\text{event}\rangle

<\text{event opcode='insert' window='project_01/cq_01/join'>}
\langle\text{name='ID'}\rangle4\langle/\text{name}\rangle
\langle\text{name='matchDescription'}\rangle\text{Workstation manufacturer}\langle/\text{name}\rangle
\langle\text{name='matchString'}\rangle\text{sunw}\langle/\text{name}\rangle
\langle\text{name='price'}\rangle10.200000\langle/\text{name}\rangle
\langle\text{name='quant'}\rangle85\langle/\text{name}\rangle
\langle/\text{event}\rangle
Implementing 1+N-Way Failover

Overview to 1+N-Way Failover

SAS Event Stream Processing can use message buses to provide 1+N-way failover. You can configure a message bus connector or adapter to exchange CSV, JSON, or data in other text formats across the message bus. However, when enabled for 1+N-Way failover, messaging publishers and subscribers must exchange binary event blocks. This is because only binary event blocks contain the required message IDs.

When traversing a message bus, event blocks are mapped one-to-one to appliance messages. Each payload message contains exactly one event block. A payload appliance message encapsulates the event block and transports it unmodified.

This topic uses the terms “message” and “event block” interchangeably. The term active/standby identifies the state of any event stream processing engine in a 1+N cluster of event stream processing engine. The term primary/secondary identifies the state of a message bus with respect to another message bus in a redundant pair. The terms 1+N, failover, cluster, and combinations of these terms are used interchangeably.

The following diagram shows how an engine integrates with message buses to provide failover. It shows two separate message buses: one between publishers and event stream processing engines and a second between event stream processing engine and subscribers. In actual deployments, the two buses do not have to reside on separate appliances. Regardless of whether publishers and subscribers use the same or different appliances, there are two messaging appliances for each virtual messaging appliance — a primary and secondary for messaging appliance failover.
In this diagram, ESP1 is the active engine on start-up. ESP2 and ESP3 are standby engines that receive published event blocks. The standby engines do not send processed event blocks to the subscriber message bus, as depicted with dotted arrows. The event stream processing message bus connector for subscribe services is connected to the fabric. A standby engine does not send event blocks to the message bus unless it becomes active on failover.

All event stream processing engines in a 1+N failover cluster must implement the same model. It is especially important that all engines in the cluster use the same engine name to coordinate the topic names on which messages are exchanged through the message bus.

Publishers and subscribers can continue to use the publish/subscribe API even when they are subscribing or publishing through the message bus for failover.

The following transport options are supported by the publish/subscribe API and in adapter configuration. These options are supported so that failover can be introduced to an existing implementation without reengineering the subscribers and publishers:

- native
- Rabbit MQ
- Solace
- Tervela
- Kafka
When you use the message bus for publish/subscribe, the publish/subscribe API uses the message bus API to communicate with the messaging appliance. It does not establish a direct TCP connection to the event stream processing publish/subscribe server.

Engines implement Rabbit MQ, Solace, Tervela, or Kafka connectors in order to communicate with the message bus. Like client publishers and subscribers, they are effectively subscribers and publishers. They subscribe to the message bus for messages from the publishers. They publish to the message bus so that it can publish messages to the subscribers.

These message buses support using direct (that is, non-persistent) or persistent messaging modes.

- Rabbit MQ connectors implement non-persistence by declaring non-durable auto-delete queues. They implement persistence by declaring durable non-auto-delete queues. The durable queues require explicit message acknowledgment, which the connector does not do. Messages are read but not consumed from the queue when they are not acknowledged.
- Solace connectors can use either direct or persistent messaging.
- Tervela connectors require that Tervela fabrics use persistent messaging for all publish/subscribe communication between publishers, subscribers, and .
- Kafka is persistent by default (subject to the cluster log.retention parameters). Connectors, adapters, and publish/subscribe clients that consume from a Kafka partition can specify the offset from which to begin consuming.

Enabling persistent messaging on the message bus implies the following:

- The message bus guarantees delivery of messages to and from its clients using its proprietary acknowledgment mechanisms. Duplicate message detection, lost message detection, retransmissions, and lost ACK handling are handled by the message bus.
- Upon re-connection of any client and its re-subscription to an existing topic, the message bus replays all the messages that it has persisted for that topic. The number of messages or time span covered depends on the configuration of the message bus.
- At the start of the day, the message bus should be purged of all messages on related topics. Message IDs must be synchronized across all connectors.

The event stream processing engines are deployed in a 1+N redundant manner. This means the following:

- All the event stream processing engines in the 1+N cluster receive messages from the publishers.
- Only the active event stream processing engine in the 1+N cluster publishes messages to the subscribers.
- One or more backup event stream processing engines in a 1+N cluster might be located in a remote data center, and connected over the WAN.

For simplicity, the reference architecture diagram illustrates one cluster of 1+N redundant event stream processing engines. However, there can be multiple clusters of event stream processing engines, each subscribing and publishing on a different set of topics. A single publisher can send messages to multiple clusters of event stream processing engines. A single subscriber can receive messages from multiple event stream processing engines.

The message bus provides a mechanism to signal to an event stream processing engine that it is the active engine in the cluster. The message bus provides a way for an engine, when notified that it is active, to determine the last message published by the previously active engine. The newly active engine can resume publishing at the appropriate point in the message stream.

Sequence numbering of messages is managed by the event stream processor’s connectors for the following purposes:

- detecting duplicates
- detecting gaps
- determining where to resume sending from after a fail-over
An event stream processing engine that is brought online resynchronizes with the day’s published data and the active engine. The process occurs after a failure or when a new engine is added to a 1+N cluster.

Event stream processing engines are deployed in 1+N redundancy clusters. All engines in the cluster subscribe to the same topics on the message bus, and hence receive exactly the same data. However, only one of the engines in the cluster is deemed the active engine at any time. Only the active engine publishes data to the downstream subscribers.

Requirements

Required Software Components

Note the following requirements when you implement 1+N-way failover:

- The SAS Event Stream Processing model must implement the required Solace, Tervela, RabbitMQ, or Kafka publish and subscribe connectors. The subscribe connectors must have “hotfailover” configured to enable 1+N-way failover.

- Client publisher and subscriber applications must use the Solace, Tervela, RabbitMQ, or Kafka publish/subscribe API provided with SAS Event Stream Processing. For C or C++ applications, the Solace, Tervela, RabbitMQ, or Kafka transport option is requested by calling `C_dfESPpubsubSetPubsubLib()` before calling `C_dfESPpubsubInit()`. For Python applications, the Solace, Tervela, RabbitMQ, or Kafka transport option is requested by calling `SetPubsubLib()` before calling `Init()`. For Java applications, the Solace, Tervela, RabbitMQ, or Kafka transport option is invoked by inserting `dfx-esp-solace-api.jar`, `dfx-esp-tervela-api.jar`, `dfx-esp-rabbitmq-api.jar`, or `dfx-esp-kafka-api.jar` into the classpath in front of `dfx-esp-api.jar`.

- You must install the Solace, Tervela, RabbitMQ, or Kafka run-time libraries on platforms that host running instances of the connectors and clients. SAS Event Stream Processing does not ship any appliance standard API libraries. The run-time environment must define the path to those libraries (using LD_LIBRARY_PATH on Linux platforms, for example).

Required Hardware Components

SAS Event Stream Processing failover requires the installation of one of the following supported message buses. The message bus must be installed in redundant fashion to avoid a single point of failure.

Table 1  Message Bus Requirements

<table>
<thead>
<tr>
<th>Message Bus</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kafka</td>
<td>Requires three or more servers for redundant installation of Kafka and associated Zookeeper instances. Provides higher throughput than other software-based buses when reading input messages from multiple Kafka topics into multiple Source windows. See redundancy details here: <a href="https://kafka.apache.org/documentation/#replication">https://kafka.apache.org/documentation/#replication</a></td>
</tr>
<tr>
<td>RabbitMQ</td>
<td>Requires three or more servers for redundant installation. Recommended only when messages on an input RabbitMQ topic are produced by a single producer, else order is not guaranteed for the consumer (ESP redundant servers in this case). See redundancy details here: <a href="https://www.rabbitmq.com/ha.html">https://www.rabbitmq.com/ha.html</a></td>
</tr>
<tr>
<td>Solace VMR</td>
<td>Requires three or more servers for redundant installation. Provides dedicated commercial level support. See redundancy details here: <a href="http://docs.solace.com/Features/VMR-Redundancy.htm">http://docs.solace.com/Features/VMR-Redundancy.htm</a> Note: SAS Event Stream Processing primary and Solace primary are not supported on the same machine.</td>
</tr>
</tbody>
</table>
### Message Bus Requirements

<table>
<thead>
<tr>
<th>Message Bus</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solace</strong></td>
<td>Runs on dedicated hardware and supports higher throughput than software-based buses. Provides dedicated commercial level support. See redundancy details here: <a href="http://docs.solace.com/Features/Redundancy-and-Fault-Tolerance.htm">http://docs.solace.com/Features/Redundancy-and-Fault-Tolerance.htm</a></td>
</tr>
<tr>
<td><strong>Tervela</strong></td>
<td>Runs on dedicated hardware and supports higher throughput than software-based buses. Provides dedicated commercial level support. <a href="http://www.tervela.com">http://www.tervela.com</a></td>
</tr>
</tbody>
</table>

Software-based message buses usually require three or more servers for redundancy. This is well known, and is required to deal with the classic high availability problem outlined here: [https://en.wikipedia.org/wiki/Split-brain_(computing)](https://en.wikipedia.org/wiki/Split-brain_(computing))

A software-based message bus might or might not be co-located with the set of two or more redundant ESP servers. It should not matter whether the software bus or SAS Event Stream Processing is containerized, provided these conditions are met:

- IP connectivity between the ESP server and the message bus is maintained at all times.
- The servers or VMs are appropriately sized (CPU cores and memory) to handle maximum SAS Event Stream Processing and message bus load. Typically, this means that all servers are identically sized. Any one of them must be able to handle a full SAS Event Stream Processing and message bus load, even if only temporarily after a server failure.

### Failover Quick Start

#### Overview

Ensure that you meet the software and hardware requirements before proceeding. There are two ways to update models during failover: dynamic model replacement and complete model replacement.

#### Dynamic Model Replacement

There are a few restrictions on updating a model dynamically. In general, model updates that do not change schemas, update windows with connectors, or require Pattern windows to retain open pattern state can be performed dynamically. For complete information about this topic, see “Dynamic Model Changes” in SAS Event Stream Processing: Using the ESP Server.

To update a model dynamically, use the RESTful API to update the models on each ESP server. Use the RESTful API projects request, with a state value of modified, to update the model. Include the XML model definition on the request as either a URI or data.

```
http://server_name:http_port/SASESP/projects/project_name/state?value=modified
```

For example, suppose that you have a project named sample_project on the ESP server on localhost at port 8181. You can use the ESP client to update the project with an XML model that is defined in the file model-updated.xml with one of the following commands:

```
```

```
```
Complete Model Replacement

If you cannot update the model dynamically, then you must replace the entire model. This requires stopping the previous model and starting the updated model.

**Note:** Because the model is changing, you cannot use the ** persist-and-restore process** with the updated model.

If the schema of any of the windows with active subscribers or publishers changes, then all instances of the model must stop and all publishers and subscribers must stop.

You can restart the failover cluster after each server is updated with the new model. If the failover cluster can recover the state of the model from the message bus, then each server can be stopped and started with the new model. See “Restoring Failed Active Engine State after Restart” for details. If the failover cluster cannot recover the engine state, then you must stop all instances of the model, publishers, and subscribers. You can restart the failover cluster after each server is updated with the new model.

**Topic Naming**

Topic names are mapped directly to windows that send or receive event blocks through the fabric. Because all event stream processing engines in a 1+N cluster implement the same model, they also use an identical set of topics on the fabric. However, to isolate publish flows from subscribe flows to the same window, all topic names are appended with an “in” or “out” designator. This enables clients and SAS Event Stream Processing appliance connectors to use appliance subscriptions and publications, where event blocks can flow only in one direction.

Current client applications continue to use the standard URL format, which includes a ** host:port** section. No publish/subscribe server exists, so ** host:port** is not interpreted literally. It is overloaded to indicate the target 1+N cluster of event stream processing engines. All of these event stream processing engines have the same engine name, so a direct mapping between ** host:port** and engine name is established to associate a set of clients with a specific 1+N engine cluster.

You create this mapping by configuring each connector with a “**urlhostport**” parameter that contains the ** host:port** section of the URL passed by the client to the publish/subscribe API. This parameter must be identical for all appliance connectors in the same 1+N failover cluster.

**Failover Mechanisms**

If the active engine in a failover cluster fails, the standby SAS Event Stream Processing appliance connectors are notified. Then one of them becomes the new active engine. The fabric tells the new active connector the ID of the last message that it received on the window-specific “out” topic. The new active connector begins sending data on that “out” topic with ID + 1.

When appliance connectors are inactive, they buffer outbound messages (up to a configurable maximum) so that they can find messages starting with ID+1 in the buffer if necessary.

**Note:** Failover support is available only when the message format configured on the failover-enabled connectors is set to “binary”.

**Restoring Failed Active Engine State after Restart**

When you manually bring a failed active event stream processing engine back online, it is made available as a standby when another engine in the cluster is currently active. If the message bus is operating in “direct” mode, persisted messages on the topic do not replay. The standby engine remains out-of-sync with other engines with injected event blocks. When the message bus is in “persistence” or “guaranteed” mode, it replays as much data as it has persisted on the “in” topic when a client reconnects. The amount of data that is persisted depends on message bus configuration and disk resources. In many cases, the data persisted might not be enough to cover one day of messages.
**Using Persist/Restore**

Use the engine persist/restore feature with an appliance in “guaranteed” mode. This guarantees that a rebooted engine can be fully synchronized with other running event stream processing engines in a failover cluster. Using the persist/restore feature in this mode requires that engine state be periodically persisted by any single engine in the failover cluster. A persist can be triggered by the model itself; in a failover cluster that would generate redundant persist data.

Alternatively, a client can use the publish/subscribe API to trigger a persist by an engine. The URL provided by the client specifies `host:port`, which maps to a specific failover cluster. The messaging mechanism guarantees that only one engine in the cluster receives the message and executes the persist. On a Rabbit MQ server, this is achieved by having the connector use a well-known queue name. Only a single queue exists, and the first engine to consume the persist request performs the persist action. On Solace appliances, this is achieved by setting Deliver-To-One on the persist message to the metadata topic. On the Tervela Data Fabric this is achieved by sending the persist message to an inbox owned by only one engine in the failover cluster.

On a Kafka cluster, the Kafka connector consumes messages on the topic reserved for metadata requests using a global group ID. In this way, only one consumer in the group sees the message.

The persist data is always written to disk. The target path for the persist data is specified in the client persist API method. Any client that requests persists of an engine in a specific failover cluster should specify the same path. This path can point to shared disk, so successive persists do not have to be executed by the same engine in the failover cluster.

The other requirement is that the model must execute a restore on boot so that a rebooted standby engine can synchronize its state using the last persisted snapshot. On start-up, appliance connectors always get the message ID of the last event block that was restored. If the restore failed or was not requested, the connector gets 0. This message ID is compared to those of all messages received through replay by a persistence-enabled appliance. Any duplicate messages are ignored.

**Message Sequence Numbers**

The message IDs used to synchronize engine failovers are generated by the engine. They are inserted into an event block when that event block is injected into the model. This ID is a 64-bit integer that is unique within the scope of its project or query or window. Therefore, this ID is unique for the connector. When redundant engines receive identical input, this ID is guaranteed to be identical for an event block that is generated by different engines in a failover cluster.

The message IDs used to synchronize a rebooted engine with published event blocks are generated by the inject method of the Rabbit MQ, Solace, Tervela, or Kafka publisher client API. They are inserted into the event block when the event block is published into the appliance by the client. This ID is a 64-bit integer that is incremented for each event block published by the client. Configure the `useclientmsgid` parameter on the connector in order to use these message sequence numbers.

**Failover with RabbitMQ**

**Installing and Configuring RabbitMQ**

For information about installing RabbitMQ, see “Using the Rabbit MQ Connector” in SAS Event Stream Processing: Connectors and Adapters.

**Required Message Bus Configuration with Rabbit MQ**

You must install the presence-exchange plug-in in order to use the Rabbit MQ server in a 1+N-way failover topology. You can download the plug-in from https://github.com/tonyg/presence-exchange.
Required Client Configuration with RabbitMQ
A RabbitMQ client application requires a client configuration file named `rabbitmq.cfg` in the current directory to provide RabbitMQ connectivity parameters.

See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.

Topic Naming with RabbitMQ
The topic name format used on RabbitMQ appliances is as follows:

```
host:port/project/contquery/window/direction
```

direction takes the value "I" or "O". Because this information appears in a client URL, it is easy for clients to determine the correct appliance topic. SAS Event Stream Processing appliance connectors use their configured "urlhostport" parameter to derive the "host:port" section of the topic name, and the rest of the information is known by the connector.

Determining Engine Active/Standby State with RabbitMQ
All SAS Event Stream Processing subscribers declare a RabbitMQ exchange of type `x-presence`. The exchange is named after the configured exchange name with `_failoverpresence` appended. Then subscribers bind to a queue to both send and receive notifications of bind and unbind actions by all peers.

All engines receive send and receive notifications in the same order. Therefore, they maintain the same ordered list of present event stream processing engines (that is, those that are bound). The first engine in the list is always the active engine. When a notification is received, an engine compares its current active/standby state to its position in the list and updates its active/standby state when necessary.

To enable log messages that indicate which ESP server is active and which is standby, use the `-loglevel esp=info` option when instantiating the ESP server.

New Engine Active Actions on Failover with RabbitMQ
When a subscriber connector starts in standby state, it creates a queue that is bound to the out topic that is used by the currently active connector. The subscriber consumes and discards all messages received on this queue, except for the last one received. When its state changes from standby to active, the subscriber does the following:

- extracts the message ID from the last received message
- deletes its receive queue
- begins publishing starting with the following message:

  ```
  ID = last message ID + 1
  ```

The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. It discards older messages from the queue.

Metadata Exchanges with RabbitMQ
The RabbitMQ publish/subscribe API handles the `C_dfESPpubsubQueryMeta()` and `C_dfESPpubsubPersistModel()` methods as follows:

- The connectors listen for metadata requests on a special topic named "urlhostport/M".
- The client sends formatted messages on this topic in request or reply fashion.
- The request messages are always sent using a well-known queue name with multiple consumers. This is to ensure that no more than one engine in the failover cluster handles the message.
The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

**Failover with Solace**

**Required Appliance Configuration with Solace**

For information about the minimum configuration required by a Solace appliance used in a 1+N-way failover topology, see “Using the Solace Systems Connector” in SAS Event Stream Processing: Connectors and Adapters.

**Required Client Configuration with Solace**

A Solace client application requires a client configuration file named `solace.cfg` in the current directory to provide appliance connectivity parameters.

See the documentation of the C_dfESPpubsubSetPubsubLib() publish/subscribe API function for details about the contents of these configuration files.

**Topic Naming with Solace**

The topic name format used on Solace appliances is as follows: `host:port/project/contquery/window/direction`, where `direction` takes the value “I” or “O”. Because all this information is present in a client URL, it is easy for clients to determine the correct appliance topic. SAS Event Stream Processing appliance connectors use their configured “urlhostport” parameter to derive the “host:port” section of the topic name, and the rest of the information is known by the connector.

**Determining Engine Active/Standby State with Solace**

For Solace appliances, an exclusive messaging queue is shared amongst all the engines in the 1+N cluster. The queue is used to signal active state. No data is published to this queue. It is used as a semaphore to determine which engine is the active at any point in time.
Engine active/standby status is coordinated among the engines using the following mechanism:

1. When a SAS Event Stream Processing subscriber appliance connector starts, it tries, as a queue consumer, to bind to the exclusive queue that has been created for the engine cluster.

2. If the connector is the first to bind to the queue, it receives a “Flow Active” indication from the messaging appliance API. This signals to the connector that it is now the active engine.

3. As other connectors bind to the queue, they receive a “Flow Inactive” indication. This indicates that they are standby engines, and should not be publishing data onto the message bus.

4. If the active engine fails or disconnects from the appliance, one of the standby connectors receives a “Flow Active” indication from the messaging appliance API. Originally, this is the second standby connector to connect to the appliance. This indicates that it is now the active engine in the cluster.

**New Engine Active Actions on Failover with Solace**

The newly active engine determines, from the message bus, the last message published by the previously active engine for the relevant window. To assist in this process, guaranteed messaging Last Value Queues (LVQs) are used.

LVQs are subscribed to the same “out” topics that are used by the appliance connectors. An LVQ has the unique characteristic that it maintains a queue depth of one message, which contains the last message published on the topic to which it subscribed. When the engine can publish messages as “direct” or “guaranteed”, those messages can always be received by a guaranteed messaging queue that has subscribed to the message topic. Thus, the LVQ always contains the last message that an engine in the cluster published onto the message bus.

When an engine receives a “Flow Active” indication, it binds to the LVQ as a browser. It then retrieves the last message published from the queue, saves its message ID, disconnects from the LVQ, and starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent
messages from the queue that it maintained while it was inactive. It can ignore newly received messages until the one with ID = saved message ID + 1 is received.

**Figure 6  Last Value Queues**

---

**Metadata Exchanges with Solace**

The Solace publish/subscribe API handles the `C_dfESPpubsubQueryMeta()` and `C_dfESPpubsubPersistModel()` methods as follows:

- The connectors listen for metadata requests on a special topic named "urlhostport/M".
- The client sends formatted messages on this topic in request or reply fashion.
- The request messages are always sent using Deliver-To-One. This is to ensure that no more than one engine in the failover cluster handles the message.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

**Failover with Tervela**

**Required Client Configuration with Tervela**

A Tervela client application requires a client configuration file named `client.config` in the current directory to provide appliance connectivity parameters.

See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.
Required Appliance Configuration with Tervela

A Tervela appliance used in a 1+N-way failover topology requires the following configuration at a minimum:

- A client user name and password to match the connector's tvuserid and tvapassword configuration parameters.
- The inbound and outbound topic strings and associated schema. (See topic string formats described previously.)
- Publish or subscribe entitlement rights associated with a client user name described previously.

Topic Naming with Tervela

The topic name format used on Tervela appliances is as follows:

```
“SAS.ENGINES.engine.project.contquery.window.direction”, where direction takes the value “IN” or “OUT”. SAS Event Stream Processing appliance connectors know this information, so it is easy for them to determine the correct appliance topic.
```

Clients must be able to map the “host:port” section of the received URL to the engine section of the topic name. This mapping is obtained by the client by subscribing to a special topic named SAS.META.host:port. The SAS Event Stream Processing appliance connectors use their configured “urlhostport” parameter to build this topic name. They publish a metadata message to the topic that includes the “host:port” to engine mapping. Only after receiving this message can clients send or receive event block data. SAS Event Stream Processing appliance connectors automatically send this message when the SAS Event Stream Processing model is started.

Determining Engine Active/Standby State with Tervela

When using the Tervela Data Fabric, engine active/standby status is signaled to the engines using the following mechanism:

1. When an SAS Event Stream Processing subscriber appliance connector starts, it attempts to create a “well-known” Tervela inbox. It uses the engine name for the inbox name, which makes it specific to the failover cluster. If successful, that connector takes ownership of a system-wide Tervela GD context, and becomes active. If the inbox already exists, another connector is already active. The connector becomes standby and does not publish data onto the message bus.

2. When a connector becomes standby, it also connects to the inbox, and sends an empty message to it.

3. The active connector receives an empty message from all standby connectors. It assigns the first responder the role of the active standby connector by responding to the empty message. The active connector maintains a map of all standby connectors and their status.

4. If the active connector receives notification of an inbox disconnect by a standby connector, it notifies another standby connector to become the active standby, using the same mechanism.

5. If the active engine fails, the inbox also fails. At this point the fabric sends a TVA_ERR_INBOX_COMM_LOST message sent to the connected standby connectors.

6. When the active standby connector receives a TVA_ERR_INBOX_COMM_LOST message, it becomes the active engine in the failover cluster. It then creates a new inbox as described in step 1.

7. When another standby connector receives a TVA_ERR_COMM_LOST message, it retains standby status. It also finds the new inbox, connects to it, and send an empty message to it.
New Engine Active Actions on Failover with Tervela

Active Tervela appliance connectors own a cluster-wide Tervela GD context with a name that matches the configured "tvaclientname" parameter. This parameter must be identical for all subscribe appliance connectors in the same failover cluster. When a connector goes active because of a failover, it takes over the existing GD context. This allows it to query the context for the ID of the last successfully published message, and this message ID is saved.

The connector then starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. Alternatively, it can ignore newly received messages until the one with ID = saved message ID + 1 is received.

Metadata Exchanges with Tervela

The Tervela publish/subscribe API handles the C_dfESPpubsubQueryMeta() method as follows:

- On start-up, appliance connectors publish complete metadata information about special topic "SAS.META.host:port". This information includes the "urlhostport" to engine mapping needed by the clients.
- On start-up, clients subscribe to this topic and save the received metadata and engine mapping. To process a subsequent C_dfESPpubsubQueryMeta() request, the client copies the requested information from the saved response(s).

The Tervela publish/subscribe API handles the C_dfESPpubsubPersistModel() method as follows.

- Using the same global inbox scheme described previously, the appliance connectors create a single cluster-wide inbox named "engine_meta".
- The client derives the inbox name using the received "urlhostport" - engine mapping, and sends formatted messages to this inbox in request or reply fashion.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

Failover with Kafka

Required Message Bus Configuration with Kafka

When you use a Kafka cluster in a 1+N-way failover topology, the SAS Event Stream Processing Kafka connectors must have access to an Apache Zookeeper cluster. The connectors require this access in order to monitor the presence of other event stream processing servers in the failover group. When Zookeeper is installed as part of the Kafka cluster installation, you can use it for monitoring by configuring host:port on the Kafka connectors. You can download Zookeeper from https://zookeeper.apache.org.

Note: The Zookeeper configuration must specify a value for tickTime no greater than 500 milliseconds. If Zookeeper is already in use, this setting might conflict with client requirements. If there are conflicts, a separate Zookeeper installation is required for failover.

Note: Do not configure Kafka connectors or adapters or client transports with partition = –1. 1+N-way failover is supported only when all components read or write a single fixed partition.

Required Client Configuration with Kafka

An event stream processing publish/subscribe client application that uses Kafka requires a client configuration file named kafka.cfg. This configuration file must exist in the current directory to provide Kafka connectivity parameters. See the documentation of the C_dfESPpubsubSetPubsubLib() publish/subscribe API function for details about the contents of these configuration files.
**Topic Naming with Kafka**

The topic name format used on Kafka clusters is as follows:

```plaintext
host_port.project.contquery.window.direction
```

The `direction` takes the value "I" or "O". Because this information appears in a client URL, clients can easily determine the correct appliance topic. Kafka connectors use their configured `urlhostport` parameter to derive the "host:port" section of the topic name. The rest of the information is known by the connector. To meet Kafka topic naming requirements, the configured `urlhostport` string must replace ':' with '_'.

**Determining Engine Active/Standby State with Kafka**

All subscriber Kafka connectors that are enabled for failover require connectivity to a Zookeeper cluster. The first subscriber connector to start up within an event stream processing server implements a Zookeeper watcher. When necessary, the connector also creates a root "/ESP" zNode. Then it creates a leaf node that is both sequential and ephemeral. It creates the node using the path "/ESP/server_n_seq", where `seq` specifies an incrementing integer that is appended by Zookeeper. All other ESP servers in the failover group follow the same process. Thus, each server is represented in Zookeeper by a unique zNode. Each server implements a Zookeeper watcher. The first server to connect to Zookeeper has the smallest path (as identified by the `seq` field).

Status changes related to these Zookeeper nodes are logged to the event stream processing server console as info-level messages. When a watcher receives a zNode status change, it does the following:

- gathers all the current "/ESP" child nodes
- finds the path with the greatest path that is less than its own
- begins watching the zNode that owns that path

The watched zNode is the active engine. If a path was not found, the watcher itself has the smallest path and becomes the active engine.

The result is that the server with the smallest path is always the active engine. The server with the next smallest path (if there is one) is the watcher of the active engine. That server becomes the active engine when the active engine fails. In this way, no more than one zNode is watched. The zNode is watched only by one other zNode, which keeps Zookeeper chatter to a minimum. The Zookeeper `tickTime` configuration parameter must be no greater than 500 milliseconds. This enables the watcher to detect a failed active engine within one second.

**New Engine Active Actions on Failover with Kafka**

A standby engine server runs a Zookeeper watcher that watches the active server. When its state changes from standby to active, each subscriber Kafka connector does the following:

- queries the outbound Kafka topic for the current offset into the partition that the subscriber is configured to write to
- creates a consumer and consumes the message at `current_offset - 1` from that partition
- stops the consumer
- extracts the message ID from the event block in the received message
- begins publishing starting with the following message:

  ```plaintext
  ID = message_ID + 1
  ```

Suppose that the next event block produced by the subscribed window contains an ID greater than that. In that case, the connector publishes all messages in the gap from the queue that it maintained while it was in standby state. It then discards older messages from the queue. Then it resumes publishing messages produced by the subscribed window.
Metadata Exchanges with Kafka

The Kafka publish/subscribe API handles the `C_dfESPpubsubQueryMeta()`, `C_dfESPpubsubPersistModel()`, and `C_dfESPpubsubQuiesceProject()` methods as follows:

- The connectors running in a failover group listen for metadata requests on a special topic named "urlhostport.M". They consume the topic using a global group ID, such that only one consumer in the group processes any message sent on that topic.
- The client sends formatted messages on this topic in request-reply fashion.
- The request contains an additional field that specifies a GUID-based topic on which to send the response. Since only the requester is listening on this topic, the response is guaranteed to be received by the requester and only the requester.
- The response contains the same information that is provided by the native publish/subscribe API.

Publisher Adapter Failover with Kafka

Failover for C and Java publisher adapters uses the Kafka failover mechanism described previously. Thus, the publisher adapter must publish event blocks to the ESP server using the Kafka transport. It also must be able to access a Kafka broker and a Zookeeper cluster. Client configuration parameters must be configured in a file named kafka.cfg.

For a C adapter, the librdkafka and zookeeper client libraries must be installed. For more information, see "Using the Kafka Connector and Adapter" in SAS Event Stream Processing: Connectors and Adapters.

For a Java adapter, the native Kafka Java client JAR files and Apache Zookeeper Java client JAR files must be installed. For more information, see "Using Alternative Transport Libraries for Java Clients" in SAS Event Stream Processing: Publish/Subscribe API.

You must define the following parameters in kafka.cfg to support a failover enabled publisher adapter:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hotfailover</td>
<td>enables or disables hot failover. Valid values are <code>true</code> and <code>false</code>.</td>
</tr>
<tr>
<td>numbufferedmsgs</td>
<td>specifies the maximum number of messages to buffer on a standby adapter.</td>
</tr>
<tr>
<td>zookeeperhostport</td>
<td>specifies the Zookeeper cluster <code>host:port</code>.</td>
</tr>
<tr>
<td>failovergroup</td>
<td>a string defined identically for all publisher adapters that belong to the same failover group</td>
</tr>
</tbody>
</table>

To guarantee a successful switchover from standby adapter to active adapter, you must ensure the following:

- All publisher adapters in the failover group must be configured identically in order to ensure that they receive identical input from the publishing source. When the source is a message bus, the associated topic or queue must deliver identical messages to all clients.
- All publisher adapters in the failover group must begin publishing at approximately the same time in order to reduce buffering requirements on standby adapters. The maximum number of buffered event blocks is set by the `numbufferedmsgs` parameter.
Following these steps guarantees that all adapters in the failover group build the same sequence of SAS Event Stream Processing event blocks to publish to the ESP server. It also guarantees that the unique message ID associated with each event block is assigned identically on all adapters in the group.

When you restart a failed publisher adapter, it must have access to the complete source data. It needs this access in order to get back in sync with already running adapters in the same group. If it reads incomplete data, then it assigns message IDs to event blocks incorrectly. Thus, when it becomes active, the input stream to the ESP server contains missing or duplicate event blocks.

Example: Testing Failover with RabbitMQ

The following example sets up failover in SAS Event Stream Processing with RabbitMQ and tests a failover scenario. This example uses SAS Event Stream Processing Studio to create models.

SAS Event Stream Processing Model Setup

After you have installed and configured RabbitMQ, you can create a model with the appropriate RabbitMQ connectors and adapters in a failover environment. Setting up failover requires at least two systems.

SAS Event Stream Processing supports homogenous and heterogeneous failover environments. The following example uses two Red Hat Linux systems in a homogeneous failover environment: RH1 and RH2

The following model shows a Source window feeding events to three Copy windows. Each copy window has a different retention policy. Each of those Copy windows feeds events to an Aggregate window.

A connector to the source window (TradesSource) enables the model to receive data from a dynamic queue bound to a specific routing key. When data is published by the file and socket adapter using the transport option, it publishes to a dynamic RabbitMQ queue. This queue is subsequently consumed by a connector defined in the Source window. In order to publish to a RabbitMQ exchange, an exchange must be created. If you do not create an exchange manually, the connector creates the exchange automatically. An exchange serves the role of a message router. In this test, the same exchange is also used for subscribing.

The RabbitMQ connector is defined in one of the Aggregate windows (Aggregate24HR) to push events to a second RabbitMQ queue through the exchange. A second file and socket adapter subscribes to events by creating a dynamic queue for an exchange and routing key and writes events to a file.
You can use `$DFESP_HOME/examples/xml/vwap_xml/trades1M.csv` on RH2 to ensure that events successfully flow through the model and are written to a CSV file.

Set up the failover environment as follows:

1. Copy the original project to the second machine (in this case RH2).
3. Start ESP servers on both machines. Use the `-loglevel esp=info` option to display active and standby messages in the ESP server log.
4. Define a RabbitMQ exchange (Trades) with type `topic` and durability `transient`.
5. Define RabbitMQ connectors for a Source window (TradesSource) and an Aggregate window (Aggregate24HR).
6. With the transport option, use the file and socket adapter to publish events to a Source window (TradesSource).
7. Use a file and socket adapter to subscribe from an Aggregate window (Aggregate24HR) using the transport option.

The following diagram provides an overview of the failover environment that is created. TradesRMQ is the engine instance and trades is the project name (not to be confused with the exchange name). The RabbitMQ message bus, adapters, and one instance of the engine run on RH1. The CSV "event" files, published and subscribed, are also located on RH1. The only thing that runs on RH2 is the second instance of the TradesRMQ engine.
Testing Parallel Models

After you install and configure RabbitMQ and edit connectors in the models to use RabbitMQ queues, test the models for failover. Set up with two models. Two tests are necessary: one test without a failed server and one test with a failed server.

ESP servers had been started earlier. The next step loaded the modified model in Test mode in SAS Event Stream Processing Studio on each host. It then started the modified model. At this point, the adapters have not been started, so there are no events flowing.

Use the web-based RabbitMQ management tool to monitor and watch queues. After starting the models in Test mode, you can view the list of exchanges using the management tool, as seen in the following figure.

After the model is started, the subscriber connectors automatically create the Trades_failoverpresence exchange. The subscriber connector is configured with the hotfailover argument, which triggers the creation of the Trades_failoverpresence exchange. Notice that the exchange Type is x-presence. This is the plug-in that was added after you installed RabbitMQ. This mechanism detects what engine is active and when engine binds to a Standby engine if the Active engine fails.

Even though no events have been published to the model, queues have been created. There are queues for connectors, subscribers, failover, and metadata. The metadata includes information about messages. You can view additional information about each queue by selecting the queue name.

If you specified -loglevel esp=info at ESP server start-up, then messages appear in the log regarding active/standby status. For example, if the first ESP server was started and the model was loaded on RH1, then a message similar to the following is displayed:

2016-04-29T12:13:15,875; WARN; 00000053; DF.ESP; (dfESPrmqConnector.cpp:570); dfESPrmqConnector::goActive(): Hot failover state switched to active
Similarly, when the ESP server is started and the model is loaded on the second machine, then a message similar to the following is displayed:

```
2016-04-29T12:16:40,909; INFO; 00000050; DF.ESP; (dfESPrmqConnector.cpp:1746);
dfESPrmqConnector::start(): Hot failover state switched to standby
```

After the model is running in test mode, start the file and socket adapter to subscribe to the `Aggregate24HR` window within the model using the transport option. Use the following command:

```
$DFESP_HOME/bin/dfesp_fs_adapter -k sub -h "dfESP://host:port/tradesRMQ/
  trades/Aggregate24HR?snapshot=false" -f trades1M_aggr24hr.csv -t csv -l rabbitmq
```

Notice the `-l rabbitmq` argument on the command line, which specifies the transport type. The adapter looks for RabbitMQ connection information in the `rabbitmq.cfg` in the current directory. Start the subscriber before you start the publisher to ensure that all events are captured from the model.

The subscriber adapter creates a new associated queue that is specified as the “Output” queue. This is indicated by the last character in the queue name. Events processed by the `Aggregate24HR` window are placed in this queue where the adapter retrieves them and writes them to the `trades1M_aggr24hr.csv` file.

Now that the model is running in test mode and a subscriber adapter has been started, you can start a file and socket adapter to publish events into the model. Again specify the `-l rabbitmq` argument to specify the RabbitMQ message bus. Use the following command:

```
$DFESP_HOME/bin/dfesp_fs_adapter -k pub -h "dfESP://host:port/tradesRMQ/
  trades/TradesSource" -f trades1M.csv -t csv -l rabbitmq
```

Typically, the block parameter is specified to reduce overhead, but in this scenario, blocking was not specified. This results in a blocking factor of one. This results in slow throughput.

As events flow through the model, it is possible to review the number of event blocks incoming to the queue and delivered to the connector.

After all events are published, use an HTTP request to the HTTP port on each ESP server to confirm that both engine instances processed the same number of records. You can also acquire this information using the UNIX `curl` command.
Each model should have processed the same number of records and all window counts should match.

Finally, review the file that contains the events from the **Aggregate24HR** window using the subscribe adapter.

```
[sasinst@sasserver01 ~]$ ll trades1M*
-rw-r--r-. 1 sasinst sas  98345849 Apr 15 10:41 trades1M_aggr24hr.csv
```

**Failover Testing**

To ensure that processing continues when one of the ESP servers fail, repeat the previous test and terminate the active ESP server in the middle of processing events. Suppose that the active ESP server running on RH1 was terminated by using the Ctrl-C interrupt. SAS Event Stream Processing Studio issues an error message that it can no longer communicate with the ESP server.

Several queues associated with the terminated ESP server are removed, but the subscriber output queue is still delivering messages to the adapter.

If you specified `-loglevel esp=info` at ESP server start-up, the following messages confirm that failover occurred and that the standby engine is now the active engine.

```
2016-04-29T12:27:37,946; INFO; 00000053; DF.ESP; (dfESPrmqConnector.cpp:342); dfESPrmqConnector::sendSerializedBlock(): Standby sending buffered messages, last message id = 184211, current message id = 259306
2016-04-29T12:27:51,387; INFO; 00000053; DF.ESP; (dfESPrmqConnector.cpp:383); dfESPrmqConnector::sendSerializedBlock(): Standby synced, current message id = 259306
2016-04-29T12:27:51,446; WARN; 00000053; DF.ESP; (dfESPrmqConnector.cpp:570); dfESPrmqConnector::goActive(): Hot failover state switched to active
```

A second way to confirm that the RH2 ESP server has switched active servers is to review the network traffic associated with RH2.

The following figure shows the host traffic received when events start flowing through the RH2 engine. The received traffic, ~20MB/sec, represents the events retrieved from the publish queue (that is, sent to the model). After the failover occurs, you can see that sent traffic spikes to ~18MB/sec, which represents the traffic sent from the connector on RH2 to the queue on RH1.
Checking the subscriber file created by the file and socket adapter shows that it has been replaced. The size is the same as the file created when no failover scenario was created.

```
[sasinst@sasserver01 ~]$ ll trades1M*
-rw-r–r–. 1 sasinst sas  98345849 Apr 15 10:49 trades1M_aggr24hr.csv
```

The output file from the first test is renamed. Use the Linux `diff` command to compare to the one created in the failover scenario. The results show that the files are identical, indicating a successful failover.

You can repeat the same test with the RH2 ESP server as the active server. Terminating the RH2 server should produce the same result. The model on the RH1 engine should continue to run and the adapter on the RH1 machine should continue to write events to the CSV file.

### Running an Event Stream Processing Engine in a Hadoop YARN Container

#### Overview to YARN

Hadoop YARN serves as a resource management framework to schedule and handle computing resources for distributed applications. A Resource Manager manages the global assignment of compute resources for an application. A per-application Application Master manages an application’s scheduling and coordination. YARN provides processing capacity to applications by allocating containers to them. Containers encapsulate resource elements such as memory, CPU, and so on.

For more information about YARN, see the YARN documentation on the Apache Hadoop website.

#### Starting the Server in the Hadoop YARN Container

To run an event stream processing server in a Hadoop YARN container, run the following script to implement a YARN client.

```
$DFESP_HOME/bin/dfesp_yarn_joblauncher -e localdfesphome -a httpport
```
<u pubsubport> <q yarnqueue > <p yamnpriority >
-m yarnmemory > <c yamncores > <1> <o consulhostport>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e localdfesphome</td>
<td>Specify the $DFESP_HOME environment variable setting for SAS Event Stream Processing installations on Hadoop nodes.</td>
</tr>
<tr>
<td>-a httpport</td>
<td>Specify the value for the -http argument on the dfesp_xml_server command line. This port is opened when the ESP server is started.</td>
</tr>
<tr>
<td>-u pubsubport</td>
<td>Specify the value for the -pubsub argument on the dfesp_xml_server command line. This port is opened when the ESP server is started.</td>
</tr>
<tr>
<td>-q yarnqueue</td>
<td>Specify the YARN Application Master queue. The default value is default.</td>
</tr>
<tr>
<td>-p yarnpriority</td>
<td>Specify the YARN Application Master priority. The default value is 0.</td>
</tr>
<tr>
<td>-m yamncore</td>
<td>Specify the YARN Memory resource requirement for the Application Master and event stream processing server containers.</td>
</tr>
<tr>
<td>-c yamncore</td>
<td>Specify the YARN Virtual Cores resource requirement for the Application Master and event stream processing server containers.</td>
</tr>
<tr>
<td>-l</td>
<td>Specify that the Application Master should use the /tmp/ESP/log4j.properties file found in HDFS.</td>
</tr>
<tr>
<td>-o consulhostport</td>
<td>Specify the host:port of a consult service.</td>
</tr>
</tbody>
</table>

The YARN client submits an Application Master and a script to run an ESP server. The client also passes associated resource requirements to the YARN Resource Manager.

The value of the environment variable DFESP_HADOOP_PATH must be the location of Hadoop configuration files on the system where you execute dfesp_yarn_joblauncher. This path must also point to the location of the following Hadoop JAR files, which are required by dfesp_yarn_joblauncher:

- hadoop-yarn-client-*.jar
- hadoop-yarn-api-*.jar
- hadoop-yarn-common-*.jar
- hadoop-common-*.jar
- hadoop-hdfs-*.jar
- hadoop-auth-*.jar

supporting JAR files that are usually found in the hadoop/share/common/lib directory

The ApplicationMaster submitted by dfesp_yarn_joblauncher is copied to and executed from /tmp/ESP in HDFS, and is named "sas.esp.clients.yarn.*.*.jar".

Note: The Application Master is built using Java 1.8, so the Java run-time environment on the Hadoop grid nodes must be compatible with that version. A SAS Event Stream Processing installation must already exist on every node in the grid.

No model XML file is provided. The HTTP client must subsequently manage the model run by the server.

The following command line is executed by the Application Master submitted by dfesp_yarn_joblauncher:

dfesp_xml_server -http httpport <-pubsub pubsubport >
By default, `dfesp_yarn_joblauncher` sets the following values for the Application Master:

- YARN Queue = default
- YARN Priority = 0
- YARN Resources: Memory = 32768 MB, Virtual Cores = 4

You can override these values through optional arguments to `dfesp_yarn_joblauncher`.

When launched, the Application Master requests one container in which to run the server shell script. That container request specifies the same YARN resources as were requested for the Application Master. Its defaults are 32768 MB and 4 virtual cores, unless you have passed different values to `dfesp_yarn_joblauncher`. YARN might kill any running process at any time when it exhausts its resources. Thus, you should tune these memory and core requirements to match the requirements of the running model.

The event stream processing container runs on a Hadoop node that might or might not be the node running the Application Master container.

You can invoke the `dfesp_yarn_joblauncher` again to launch additional event stream processing servers, which all run independently and have no knowledge of any other servers running on the grid.

## Managing the Event Stream Processing Server

On successful start-up, the YARN Resource Manager web application should show an entry where the Name column shows “ESP”. Make a note of the associated application ID. Click the Tracking UI link to show the Application Master parameters. This URL connects to a web server running in the event stream processing Application Master itself.

Note the Master Host and Execution Host values. These are the nodes where the Application Master and event stream processing server containers are running, respectively. You can drill down to those specific nodes in the Node Manager. There, follow the links for your application ID to find process logs for the Application Master or event stream processing server.

If a failure occurs, the option to view logs depends on whether logging aggregation is enabled on your YARN installation. If it is enabled, log on to any Hadoop node, navigate to the local Hadoop installation, and run 
```
/bin/yarn logs –applicationId=ESP_application_id
```
If it is disabled, log on to the individual Hadoop node and find the yarn logs in the local Hadoop installation directories.

To kill containers started by `dfesp_yarn_joblauncher`, log on to any Hadoop node, navigate to the local Hadoop installation, and run
```
/bin/yarn application –kill ESP_application_id
```

## Connecting to an ESP Server

The http-admin and publish/subscribe ports opened by the ESP server should be reachable by devices outside of the Hadoop grid, if network connectivity is available. The server host name is the name of the Hadoop node where the ESP server is running (shown in the Execution Host value displayed by the YARN Resource Manager). The port is the http-admin or publish/subscribe port passed to the `dfesp_yarn_joblauncher` script.

Clients connecting to this ESP server do not know that the server is running in a YARN container on a Hadoop node. Functional behavior is identical to a stand-alone server running outside a Hadoop grid.
Using the Cluster Manager

Overview

The Cluster Manager maps sources from edge devices to event stream processing engines that are provisioned within the cloud. By managing the mapping between connectors and engine instances, the Cluster Manager facilitates the elastic deployment of SAS Event Stream Processing in the cloud and eases large-scale deployment.

For example, you could use Cluster Manager when you deploy SAS Event Stream Processing as a service in Cloud Foundry. Typically, you create and manage a pool of VMs (or containers) that have their hardware resources in the cloud. Through a Cloud Foundry service broker API, you could request engine instances to be provisioned on the containers. After you provision those engine instances, you could instruct Cluster Manager to do the following:

- Deploy projects to provisioned engine instances through the administrative REST API.
- Start more than one data sources (through connectors) in an orchestrated fashion.
- Stream events for processing and analyzing through their publish/subscribe API.
- Dynamically add or remove an engine instance to or from the Cluster Manager’s control.

The following figure shows that Cluster Manager runs on the ESP server.

**Figure 7  Cluster Manager**

Cluster Manager supports the following functions:

- **Connector orchestration.** In the figure, the dotted lines between connectors represent the order of connector execution.
- **Data event routing.** Cluster Manager automatically creates a router in the ESP Server to map events to corresponding engine instances. Mappings are based on the policy that is defined in the Cluster Manager’s configuration file.
Loading projects to engine instances and dynamically managing engines. A SAS Event Stream Processing deployment in Cloud is intended to be flexible. You can dynamically add more engine instances when you need more processing power. You can remove engine instances when you do not need them.

When you run Cluster Manager, it reads the project XML file and deploys the project to engine instances that are described in the Cluster Manager configuration XML file. Cluster Manager provides a REST API to dynamically add or remove event stream processing engine instances. The routing policy reacts accordingly.

Routing Policies

Three event routing policies are supported:

- Multicast policy sends every event to all the engine instances.
- Round Robin policy sends events to engine instances in a round-robin fashion.
- Hash policy hashes the value of some pre-defined fields and uses that value to decide where to send the event.
- Durable Hash policy treats the hash value space as a ring. It uses two hash functions: one to hash the ID of a server and the other to hash an event based on fields specified in the XML model.

In Cloud deployment, you sometimes need to stream the same data to different engines. Multicast policy facilities this need. You do not have to deal with each engine instance individually and keep track of the engine instances that are subject to dynamic changes. Multicast policy relies on a multicast destination that serves as a single point of contact.

Round Robin policy distributes event blocks that are received from a connector to engine instances in a round robin fashion.

Hash policy examines each event that it receives from a connector and performs a hash function on the specified fields. It then sends the event to the engine instance according to the hash result. Fields are interpreted as strings, and the hash function might not cause a perfect uniform distribution because of its complexity constraint. Keep in mind that hashing can slow performance and significantly increase memory usage.

Durable Hash policy is suitable for a dynamic environment where servers are on and off unpredictably. When you use Durable Hash, the addition or removal of a server affects only the server; it does not lead to a change in the hash function itself. Events hashed to another server continue to be hashed to that server.

Running the Cluster Manager

Before you run Cluster Manager, provision engine instances in the cloud. Have these engine instances run the basic ESP server with their administrative and publish/subscribe ports open.

Use the following command to run Cluster Manager:

```
dfesp_cluster_manager -http-admin port -pubsub port -cluster-manager model <-auth> <-output-projects file> <-output-routers file>
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-http-admin port</td>
<td>Specify the port for HTTP administrative commands.</td>
</tr>
<tr>
<td>-pubsub port</td>
<td>Specify the port for publish/subscribe commands.</td>
</tr>
<tr>
<td>-cluster-manager model</td>
<td>Specify the XML model for Cluster Manager to use.</td>
</tr>
<tr>
<td>-auth</td>
<td>(Optional) Enable authentication.</td>
</tr>
<tr>
<td>Argument</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-output-projects file</td>
<td>(Optional) Write the project's model to the specified file and deploy manually.</td>
</tr>
<tr>
<td>-output-routers file</td>
<td>(Optional) Write the router's model to the specified file and deploy manually.</td>
</tr>
</tbody>
</table>

You can enter this command without arguments in order to obtain basic usage information.

Create a model (for example, file://config.xml) to specify the configuration file for Cluster Manager. The model must contain a reference to a project XML file that deploys projects into engine instances in the cloud. Internally, Cluster Manager parses the model and generates two configuration files that start connectors and a router on the local event stream processing server. To produce a dump of these two files, use the -output-projects and -output-routers arguments to the command that starts the ESP server.

You can use the following REST API to add and remove ESP instances:

- **PUT /routerEngines/routerId/engineId.** The routerId is the name of the <esp-map> element specified in the configuration file. The engineId is the name of the <esp-engine> element specified in the configuration file. The request body must contain the engine definition.
  
  For example: curl -X PUT -d '<esp-engine name="esp3" host="10.37.24.3" port="31417" ha_port="31416"/>' http://localhost:12345/SASESP/routerEngines/esp-map1/esp2

- **DELETE /routerEngines/routerId/engineId.**

---

**Specifying the Cluster Manager Configuration File**

What follows is a sample configuration file for Cluster Manager. It contains four major sections:

- the projects to be loaded in engine instances that receive events from data sources, defined in the <projects> element
- data sources, defined in the <raw-sources> element
- the mapping from raw-sources to the destination windows (as specified in the projects) that are running on engine instances, defined in the <esp-maps> element
- a collection of engine instances, defined in the <esp-clusters> element

**Note:** The <esp-cluster-managers> container element currently supports only a single cluster manager.

```xml
<engine>
  <esp-cluster-managers>
    <esp-cluster-manager name='adapter-manager'>
      <projects>
        <project name='test' type='reference'> ¹
          <project-url>file://broker.xml</project-url>
          <project-name>project</project-name>
        </project>
      </projects>
    </esp-cluster-manager>
  </esp-cluster-managers>
  <raw-sources> ²
    <raw-source name='trades' class='fs'>
      <properties>
        <property name='type'>pub</property>
        <property name='fstype'>binary</property>
        <property name='fname'>trades50M256B.bin</property>
      </properties>
    </raw-source>
</raw-sources>
```
<raw-source name='venuesSource' class='fs'>
  <properties>
    <property name='type'>pub</property>
    <property name='fstype'>csv</property>
    <property name='fnname'>venues.csv</property>
  </properties>
</raw-source>

<raw-source name='restrictedSource' class='fs'>
  <properties>
    <property name='type'>pub</property>
    <property name='fstype'>csv</property>
    <property name='fnname'>restricted.csv</property>
  </properties>
</raw-source>

<esp-maps>
  <map name='venuesMap' from='venuesSource'/>
  <multicast-destination name='dest2' opcode='insert'>
    <publish-target>
      <project-func>project</project-func>
      <contquery-func>query</contquery-func>
      <window-func>venuesSource</window-func>
    </publish-target>
  </multicast-destination>
</map>

<map name='restrictedMap' from='restrictedSource'/>
  <multicast-destination name='dest3' opcode='insert'>
    <publish-target>
      <project-func>project</project-func>
      <contquery-func>query</contquery-func>
      <window-func>restrictedSource</window-func>
    </publish-target>
  </multicast-destination>
</map>

<map name='tradesMap' from='trades'/>
  <hash-destination name='dest4' opcode='insert'>
    <publish-target>
      <project-func>project</project-func>
      <contquery-func>query</contquery-func>
      <window-func>trades</window-func>
    </publish-target>
    <fields>
      <field name='broker'/>4
    </fields>
  </hash-destination>
</map>

<orchestration>
  <adapter-groups>
    <adapter-group name='G1'>
      <!-- adapter code here -->
    </adapter-group>
  </adapter-groups>
</orchestration>
<adapter-entry adapter='venuesSource' state='finished'/>
<adapter-entry adapter='restrictedSource' state='finished'/>
</adapter-group>

<adapter-group name='G2'>
    <adapter-entry adapter='trades' state='finished'/>
</adapter-group>
</adapter-groups>

<edges>
    <edge source='G1' target='G2'/>
</edges>
</orchestration>
</esp-map>
</esp-maps>

<esp-clusters>

  <esp-cluster name='openstack'>
    <esp-engines>
      <esp-engine name='esp1' host='10.37.24.3' port='31415' ha_port='31414'/>  
      <esp-engine name='esp2' host='10.37.24.3' port='31417' ha_port='31416'/>  
      <esp-engine name='esp3' host='10.37.24.3' port='31419' ha_port='31418'/>  
    </esp-engines>
  </esp-cluster>

</esp-clusters>
</esp-cluster-managers>
</engine>

1 Here, "test" references a project that is specified through the project-name element in a file that is specified by the project-url element

2 Specify a data source as you would use in any esp-map element. Connection parameters are identical to connector parameters in any model.

3 This element defines the connectivity between raw-source and destination window connectivity and the orchestration of raw-sources. The element references the cluster as well as the model defined in the projects tag. The destination window is defined in the model-ref tag. That window is running on servers that are defined in the cluster-ref tag.

   Currently, the esp-map tag supports three types of destinations, multicast-destination, roundrobin-destination, and hash-destination. Esp-map also supports connector orchestration.

4 This is the field to be hashed. The hash value is an integer between 0 and nth number of engine instances minus one. The router uses the hash value to determine what engine instance receives the event.

5 For more information, see "Orchestrating Connectors" in SAS Event Stream Processing: Connectors and Adapters.

6 An esp-cluster is a collection of running engine instances. The value of host is the IP address of the instance. The value of port is the publish/subscribe port. The value of ha_port is the http port.

**Cluster Redundancy**

Enable cluster redundancy by setting up spare engines. When an engine fails, the router automatically and transparently replaces it with the specified spare engine. The spare engine inherits the failed engine’s name, so that the events hashed to the failed engine are routed to the new engine.

Use the **spare-esp-engines** element to set them up.

Here is an example:
When you specify more than one spare, the system traverses the list in order. After you set `redundancy=n`, an event is copied to `n-1` redundant engines when hashed to an engine. That way, it can survive up to `n-1` simultaneous engine failures.

**Note:** Currently, only Hash and Durable Hash policies support redundancy.

**Note:** The addition or removal of a spare engine using the REST API is not supported.

**Note:** Adding or removing an engine to a cluster does not trigger the swap-in of a spare engine.

### Using the Cluster Manager with SASLogon

Oauth2 is a protocol that enables internet users to log on to third-party websites using public accounts without exposing their passwords. SASLogon is an Oauth2 server that is compatible with Cloud Foundry UAA.

To enable SASLogon on the ESP server, enter the following at the command prompt:
```
dfesp_xml_server -saslogon-url authorization_server_url
```

Authorization occurs after the following:

1. A project is loaded to an engine.
2. SASLogon is enabled through its HTTP admin port.
3. The publish/subscribe client connects to a window in that engine.

Because Cluster Manager deploys projects to remote engines, it must specify an authorized user in the configuration file. Specifically, in the `<esp-cluster>` element of the model, here is how you specify an engine that requires SASLogon:

```
<esp-engine name='esp1' host='localhost' port='31415' ha_port='31414'>
  <auth-user>test_client</auth-user>
</esp-engine>
```

You must create an .authinfo file that can be found by Cluster Manager. Cluster Manager retrieves the password from this file and uses it to access the engine.

For more information, see “Overview” in SAS Event Stream Processing: Security.

### Cluster Manager Failover

Enable Cluster Manager failover by setting up a cluster of Cluster Managers. Make one of them the leader and designate the others standby.

For example, suppose you start a normal Cluster Manager as follows:
```
dfesp_cluster_manager -http-admin 12347 -pubsub 12348 -cluster-manager file://config.xml
```

Start two other Cluster Managers and designate them standby:
dfesp_cluster_manager -http-admin 12341 -pubsub 12342 -cluster-manager file://config.xml -standby localhost:12347

dfesp_cluster_manager -http-admin 12343 -pubsub 12344 -cluster-manager file://config.xml -standby localhost:12347

All three Cluster Managers reference the same configuration file. Each uses a different admin and pubsub port. The first Cluster Manager that you started becomes the leader.

The –standby parameter sets the Cluster Manager to a standby mode. It notifies the leader of its existence. When the leader fails, one of the standby Cluster Managers automatically becomes the new leader and continues publishing to the remote servers. The new leader traces the last received message by the targets so that it can start sending from the next message. In this way, a message is sent once and only once as long as there is a standby Cluster Manager available.

Notice that the standby Cluster Manager specifies the leader’s host:port as the argument to the –standby parameter. Events published to these standby Cluster Managers are queued in the memory.

The failover mechanism is implemented using HTTP protocol and uses the existing admin port. Cluster Manager failover assumes the sources of data are reliable (has their own fault tolerance guarantee) and guarantees that all events are delivered with order once and exactly once. The leader Cluster Manager periodically broadcasts an update message to standby Cluster Managers.

Cluster Manager has four command line parameters that can be used to customize the configuration. To make the protocol consistent, the standby Cluster Managers should use the same configuration of these parameters as the leader.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticktime</td>
<td>Defines the heartbeat rate in milliseconds. The default value is 1000ms. When the Cluster Manager runs on an edge device, specify a long ticktime.</td>
</tr>
<tr>
<td>retries</td>
<td>Defines the number of updates to receive before concluding that the leader is down. The default value is 2. To omit transient failures, set retries to a large value. The failover protocol is less aggressive, but there is a longer failure detection time.</td>
</tr>
<tr>
<td>timeout</td>
<td>Defines the time-out in ms. When a connection is half closed (for example, a server accepts a connection request but does not respond within the defined time), the initiator of the connection deems the server to be unreachable. The default value is 3000.</td>
</tr>
<tr>
<td>buffersize</td>
<td>Specifies the size of the queue. The default value is 1m.</td>
</tr>
</tbody>
</table>

When a standby Cluster Manager fails to receive a number of retries in a row, it assumes that the leader is down. The standby initiates an election immediately until all the standby Cluster Managers reach a consensus choice for the next leader. Then the next leader assumes leadership. This procedure goes on until all Cluster Managers fail.

Inside Cluster Manager, the publish/subscribe layer maintains a queue for each ESP engine. If an engine is in standby mode, events are queued rather than published.
Using SAS Event Stream Processing with SAS Cloud Analytic Services Actions

Overview

The SAS Cloud Analytic Services (CAS) server provides a cloud-based run-time environment for data management and analytics. CAS can run on a single machine or on multiple machines as a distributed server. The distributed server consists of one controller and one or more workers.

All data is available to CAS through a caslib. A caslib is an in-memory space to hold tables, access control lists, and data source information. Clients can add or upload data. The server can load data from server-side data sources.

All operations in CAS that use data are performed after you set up a caslib. Authorized users can add or manage caslibs with the CASLIB statement.

CAS actions are tasks that are performed at the request of the user. These actions are grouped into action sets. Actions perform common tasks such as data management, server management, and administration. For more information, see SAS Viya Actions and Action Sets by Name and Product.

You can set a caslib as a connection point between a CAS server and an ESP server:

1. Start the ESP Server.
2. Open SAS Studio.
3. Declare a caslib. Specify the port and server name of the ESP Server. Specify the source type as esp.

   In the following example, the caslib reference name is espStatic. The caslib connects to an ESP Server named hdp04.orion.com through port 5555. The specified source type is esp.

   /*Create an "ESP caslib" to read from ESP windows*/
   caslib espStatic desc="vwap ESP window"
   datasource=(port=5555, server="hdp04.orion.com", srcType="esp")
   global;

   This ESP caslib has global scope. This means that is it accessible from all sessions, subject to access controls.

This ESP caslib can now be used as a data source of event data for any single-pass CAS analytical action. When running that analytical action, you must specify the following information:

- The ESP caslib name
- The project_name/query_name/window path as the espUri path

You do not need a SAS Event Stream Processing license to set up an ESP caslib.

For more information about CAS, see SAS Cloud Analytic Services: Fundamentals. For more information about the programming language elements to use to start and manage CAS, see the SAS Viya Programming: CAS Language Reference.
CAS Action Sets for Use with SAS Event Stream Processing

loadStreams Action Set
The loadStreams action set enables you to load event data into in-memory tables that are populated with data from a SAS Event Stream Processing event. Using these actions requires a license for SAS Event Stream Processing and a running ESP server.

Specifically, the following actions are available:

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appendSnapshot</td>
<td>Appends the current snapshot of a window to the table</td>
</tr>
<tr>
<td>loadSnapshot</td>
<td>Loads the current snapshot of a window to the table</td>
</tr>
<tr>
<td>loadStream</td>
<td>Streams data from a window into the table</td>
</tr>
<tr>
<td>nMetaData</td>
<td>Captures all model metadata for projects, queries, and windows</td>
</tr>
</tbody>
</table>

These actions connect to a specific window and consume event data for a specified number of seconds. They load event data into an in-memory table. After that, the action disconnects from the ESP server. You can run single-pass or multi-pass analytical actions against the data in the in-memory table.

You can run any single-pass analytical action against data streams published by a SAS Event Stream Processing model in real time with the loadStream action. The loadStream action stays connected to the ESP server and continues to gather data. The action periodically (based on a number of rows or a specified length of time) appends the data gathered to an output table. The output table in SAS Cloud Analytic Services must be global in scope. The global table continues to grow until this action is stopped.

```
proc cas;

    session.sessionId;                                                /* 1 */
    loadStreams.loadStream /                                          /* 2 */
      casLib="espStatic"                                              /* 3 */
      espUri="trades_proj/trades_cq/Trades"                           /* 3 */
      casOut={caslib="mycas",name="streamTrades",promote=true};    /* 4 */

    table.fetch /                                                     /* 5 */
      table={caslib="mycas",name="streamTrades",
              vars={"security","tradeID","quantity","price"}};

run;
```

1 Display the session ID. You can use the ID to stop the subsequent loadStream action.
2 Run the loadStream action specifying the caslib for the ESP server (espStatic in this example).
3 The espUri requires a format: **project_name/query_name/window_name**.
4 Specify a different caslib and table name for the output. The output table must be global in scope, so set `promote='true'`. 
Use the fetch action to capture the output of the table snapshot.

The fetch action does not start until you explicitly stop the loadStream action. In SAS Studio, the program displays the running status, and you can simply click Cancel to stop it. You can also stop the process by issuing stopAction, specifying the session ID. The Trades example is not running live streaming content, so the fetch action should look much like this:

```
session.stopAction uuid='46415f51-904d-6a4f-9a0e-9ef1fa9e1360';
```

**espCluster Action Set**

The espCluster action set, which interfaces directly with SAS Event Stream Processing, enables a CAS server to start and view a cluster of ESP servers. SAS Event Stream Processing must be installed and configured on the worker nodes of a CAS cluster. The ESP Server on each worker node must be licensed.

This action set is essentially a sophisticated SAS Event Stream Processing client that uses the publish/subscribe interface for the analytical action event stream processing events.

### Table 4  espCluster Actions

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listServers</td>
<td>Lists ESP server information</td>
</tr>
<tr>
<td>startServers</td>
<td>Starts a cluster of ESP servers</td>
</tr>
<tr>
<td></td>
<td>After you issue this action, ESP servers continue running until the CAS session ends.</td>
</tr>
</tbody>
</table>

1 Load the espCluster action set.

2 Start the servers identified by fully qualified domain names. You can list the servers individually or start all of the servers in the cluster by specifying nodes=c().

3 Get a list of the servers in the cluster.

<table>
<thead>
<tr>
<th><code>ESP Servers</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
</tr>
<tr>
<td>grid002.example.com</td>
</tr>
<tr>
<td>grid003.example.com</td>
</tr>
<tr>
<td>grid004.example.com</td>
</tr>
</tbody>
</table>

You can use the Cluster Manager to push a project to the ESP server running on the worker nodes. The Cluster Manager connects to raw sources to get input data and appropriately stream them to a model running on the worker nodes.

When you run single-pass statistical algorithms, use the following settings for the caslib for the ESP server cluster (started by the startServers action):

- Set the port to undefined
Set the authenticationType to none

Using SAS Event Stream Processing with a Cloud Infrastructure

SAS Event Stream Processing is cloud-ready. You can deploy it almost anywhere, from data centers to edge devices and from physical machines to clouds. Currently, SAS Event Stream Processing can be used with deployment tools such as BOSH and Chef. Contact your SAS representative to initiate a SAS Event Stream Processing deployment in the cloud.

Using the Apache Camel Framework

Overview
The Apache Camel framework enables you to integrate different applications into a single, cohesive architecture. Using the Apache Camel framework, you set up routes that contain endpoints:

```xml
<route id="injectTrades" startupOrder="10">
  <from uri="systemA://someThing"/>
  <to uri="systemB://someOtherThing"/>
</route>
```

The “from” and “to” elements in the previous code are Apache Camel endpoints that refer to components. Apache Camel supports many out-of-box components. It gives you the tools to develop custom components. An Apache Camel Consumer maps to a “from” endpoint, and an Apache Camel Producer maps to a “to” endpoint.

For more information about the Apache Camel framework, see the documentation.

Installing the Apache Camel Framework

In order to use the Apache Camel framework with SAS Event Stream Processing, you must download and install various files. These include Apache components and specific JAR files.

1. Access and install the following Apache components:
   - Apache Camel, which you can download from http://camel.apache.org/
   - Apache Maven, which you can download from https://maven.apache.org/

   Apache Maven is a build environment that you can use to create projects that leverage components from SAS Event Stream Processing. When you install Apache Maven, make sure that you install the `bin` directory subordinate to your Maven install directory in your path.

2. After you have installed the Apache components, install two JAR files into your local Maven repository:
   - ESP API Client JAR
   - ESP Camel JAR

   Note: You can find these JAR files in `$DFESP_HOME/lib`. They contain the client API and the Camel components.

3. Install the JAR files into the Maven repository:

   ```sh
   $ cd $DFESP_HOME/lib
   ```
4. After you have installed the JAR files, you can reference them from your Maven project object model (pom.xml) file. You must have an entry for both the event stream processing client API and the SAS Event Stream Processing Camel components.

Here is the event stream processing client API entry:

```xml
<dependency>
    <groupId>com.sas.esp</groupId>
    <artifactId>dfx-esp-api</artifactId>
    <version>[4.2]</version>
</dependency>
```

Here is the SAS Event Stream Processing Camel entry:

```xml
<dependency>
    <groupId>com.sas.esp</groupId>
    <artifactId>dfx-esp-camel</artifactId>
    <version>[4.2]</version>
</dependency>
```

Here is the CAS authentication entry:

```xml
<dependency>
    <groupId>com.sas.esp</groupId>
    <artifactId>dfx-cas-auth</artifactId>
    <version>3.0.6</version>
</dependency>
```

---

### Installing the RabbitMQ Library

Configure the Maven project so that SAS Event Stream Processing uses RabbitMQ as an alternative transport library:

1. Install the RabbitMQ API JAR:

   ```bash
   $ cd $DFESP_HOME/lib
   $ mvn install:install-file -Dfile=dfx-esp-rabbitmq-api.jar -DgroupId=com.sas.esp
   -DartifactId=dfx-esp-rabbitmq-api -Dversion=3.2 -Dpackaging=jar
   ```

2. Update the Maven project object model (pom.xml) to include RabbitMQ dependency information:

   ```xml
   <dependency>
       <groupId>com.rabbitmq</groupId>
       <artifactId>amqp-client</artifactId>
       <version>3.5.6</version>
   </dependency>

   <dependency>
       <groupId>commons-configuration</groupId>
       <artifactId>commons-configuration</artifactId>
       <version>1.10</version>
   </dependency>

   <dependency>
       <groupId>com.sas.esp</groupId>
       <artifactId>dfx-esp-rabbitmq-api</artifactId>
   </dependency>
   ```
Note: You must define the dependency for the RabbitMQ JAR (dfx-esp-rabbitmq-api) in the pom.xml file before you define dependency for the ESP API Client JAR (dfx-esp-api).

**SAS Event Stream Processing Implementation**

The SAS Event Stream Processing implementation consists of SAS Event Stream processing Camel Endpoints that are either Consumers (which implement publish/subscribe subscribers) or Producers (which implement publish/subscribe publishers). A Consumer maps to a **from** endpoint, and a Producer maps to a **to** endpoint. For example, to receive events from one publish/subscribe server and send them to another, execute the following code:

```xml
...<endpoint id="subscribe" uri="esp://espsrv01:46003">
    <property key="project" value="project" />
    <property key="contquery" value="query" />
    <property key="window" value="transform" />
</endpoint>

<endpoint id="publish" uri="esp://espsrv01:47003">
    <property key="project" value="project" />
    <property key="contquery" value="query" />
    <property key="window" value="trades" />
</endpoint>

<route>
    <from uri="ref:subscribe" />
    <to uri="ref:publish" />
</route>
...```

The SAS Event Stream Processing components can work with the following formats representing events:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map&lt;String, Object&gt;</td>
<td>A standard JAVA Map object in which the keys are field names and the values are field values.</td>
</tr>
<tr>
<td>List&lt;Map&lt;String, Object&gt;&gt;</td>
<td>A standard JAVA List of JAVA Map objects in which the keys are field names and the values are field values.</td>
</tr>
<tr>
<td>XML</td>
<td>Event data in XML.</td>
</tr>
<tr>
<td>JSON</td>
<td>Event data in JSON.</td>
</tr>
</tbody>
</table>

A SAS Event Stream Processing Consumer (subscriber) receives events and converts them into one of these formats to send them along the route. A SAS Event Stream Processing Producer (publisher) receives data in one of these formats, converts it into events, and publishes them.

Through these standard formats, SAS Event Stream Processing easily integrates with any of the other components available in the Camel framework and shares data with them. If some transformation is required in order to get the data into the required format, you can use a transformation bean between the endpoints.

Here is an example of transforming the standard comma-separated value event format into one of the supported types:
First create an endpoint called csvData to read the CSV data from a file. You also create an endpoint called inject to inject this data into an ESP Source window. The route that you use goes from the file into SAS Event Stream Processing, but you must get the CSV data into one of the supported formats.

To do this, create a bean called csvTransform, which requires a schema and an output format. After that, you can form events from the CSV data and create an XML document to pass along the route. This is consumed by the Producer, which injects the data into the specified Source window. The method attribute for any of the SAS Event Stream Processing transformation beans is always a transform.

Using Camel Components in a Maven Project

In order to reference the SAS Event Stream Processing Camel components by URI, you must create the following file:

```
META-INF/services/org/apache/camel/component/esp
```

The file must contain the following line:

```
class=com.sas.esp.clients.camel.EspComponent
```

Note: This is described in further detail at http://camel.apache.org/writing-components.html.

This enables you to reference the event stream processing components with a URI such as the following:

```
<endpoint id="inject" uri="esp://<pub/sub host>:<pub/sub port>">
```

Configuring Endpoints

You must determine the host and port of the SAS Event Stream Processing publish/subscribe server with which you are going to communicate. Use this information to specify the URI of the component. For example, if your publish/subscribe server is running on port 46003 on machine espsrv01, your URI is:

```
esp://espsrv01:46003
```
Because you are always putting events into or getting events out of windows, you also must specify the project, continuous query, and window in which you are interested. You can do this using either of the following methods:

add parameters to the URI

\[ \text{esp://espsrv01:46003?project=myproject&contquery=mycq&window=trades} \]

specify an endpoint element with properties:

\[
\text{<endpoint id="inject" uri="esp://espsrv01:46003">}
\text{\hspace{1cm}<property key="project" value="project" />}
\text{\hspace{1cm}<property key="contquery" value="query" />}
\text{\hspace{1cm}<property key="window" value="trades" />}
\text{\hspace{1cm}</endpoint>}
\]

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Consumer</th>
<th>Producer</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>project</td>
<td>The event stream processing project.</td>
<td>x</td>
<td>x</td>
<td>A valid project.</td>
</tr>
<tr>
<td>contquery</td>
<td>The event stream processing continuous query.</td>
<td>x</td>
<td>x</td>
<td>A valid continuous query.</td>
</tr>
<tr>
<td>window</td>
<td>The event stream processing window.</td>
<td>x</td>
<td>x</td>
<td>A valid window.</td>
</tr>
<tr>
<td>format</td>
<td>The data format for this component. This is usually used with a Consumer to specify the format of the event data to send down the route. It can be used with a Producer. If the component receives a message with a body that is a String, the producer uses the format (either XML or JSON) for conversion.</td>
<td>x</td>
<td>x</td>
<td>map, list, XML, JSON</td>
</tr>
<tr>
<td>blocksize</td>
<td>The size of the event blocks to inject into a Source window.</td>
<td>x</td>
<td></td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>blob</td>
<td>The name of an event field that is used to contain the entire message body. If this property is set, the Producer receives a message and creates a MapMap&lt;String, Object&gt; where the key is the field indicated by the blob property. The value is the entire message body represented as a String. This data is injected into the appropriate Source window. Note: When using the blob property, the Source window into which the event is being injected must set both ‘insert-only=true’ and ‘autogen-key=true’.</td>
<td>x</td>
<td></td>
<td>A valid event field</td>
</tr>
<tr>
<td>authUser</td>
<td>The name to use with SASLogon authorization.</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>oAuthToken</td>
<td>The OAuth token to use with OAuth authentication.</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Using Transformation Beans

When data is not in a format immediately usable by the event stream processing components, you must use a transformation bean to convert the data into a usable format. You can use a transformation bean to convert events in the SAS Event Stream Processing CSV format into a usable format. Place the transformation beans in a route between the from and to endpoints.

Here is an example:

```xml
<route id="injectTrades">
  <from uri="ref:csvData"/>
  <bean ref="csvTransform" method="transform" />
  <to uri="ref:inject"/>
</route>
```

Each bean takes certain parameters to help it convert the data to be usable by SAS Event Stream Processing. The CSV transformation bean requires the event schema and the output data format. Note that the bean lies between a “from” endpoint that contains a file reference and a “to” endpoint to publish events into a window.

Table 6  Transformation Beans in the SAS Event Stream Processing Camel Package

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.sas.esp.clients.camel.transforms.CsvTransform</td>
<td>Transforms CSV data into ESP event data schema. The event schema for the events represented by the CSV data.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>com.sas.esp.clients.camel.transforms.RssTransform</td>
<td>Transforms RSS data into ESP event data format. The data format to use to write events. Valid values are list, XML, and JSON.</td>
</tr>
</tbody>
</table>

Examples

Where to Find Examples

A set of examples is available at `$DFESP_HOME/examples/java/camel`. 
CSV Injection

The following example reads trade data from a CSV file and injects the trades into the broker surveillance model. It also subscribes to the brokerAlertsAggr window and writes these events to the console in JSON format.

1. **Edit** `src/main/resources/esp.properties` so that it contains your publish/subscribe server information:

   ```
   espServer=esp://espsrv01:46003
   tradesFile=data/trades1M.csv
   ```

2. **Start your ESP server**:

   ```
   $ dfesp_xml_server -model file://model.xml -http-admin <http admin port> -http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
   ```

3. **Start the project**:

   ```
   $ mvn camel:run
   ```

Distributed Modeling

This example distributes broker surveillance model between two servers. The first server takes the trade data and performs all the dimensional additions (broker info, venue data) to the event. This model is in primary.xml and ends with a Functional window called transform. The events generated by transform contain a full set of trade information. The project subscribes to this window in server1 and forwards the events to server2, which looks for the broker alerts. Another route is used to subscribe to brokerAlertsAggr window in server2 and dump the events to the screen in Map format.

1. **Edit** `src/main/resources/esp.properties` so that it contains your publish/subscribe server information:

   ```
   espServer1=esp://espsrv01:46003
   espServer2=esp://espsrv01:47003
   tradesFile=data/trades1M.csv
   ```

2. **Start your primary event stream processing server**:

   ```
   ```

3. **Start your secondary event stream processing server**:

   ```
   ```

4. **Start the project**:

   ```
   $ mvn camel:run
   ```

RSS

This example uses the Camel RSS Component to set up a route that reads data from any number of RSS feeds and injects them into SAS Event Stream Processing. You should be able to add any RSS feeds to the route.

```xml
<route>
  <from uri="rss:http://feeds.reuters.com/reuters/businessNews" />
  <from uri="rss:http://feeds.reuters.com/reuters/topNews" />
  <from uri="rss:http://feeds.reuters.com/reuters/technologyNews" />
  <bean ref="rssTransform" method="transform" />
</route>
```
<to uri="ref:publishNews" />
</route>

...

You can also use a new transformation bean to transform the RSS data into a supported format:

```xml
<bean id="rssTransform" class="com.sas.esp.clients.camel.transforms.RssTransform">
    <property name="opcode" value="upsert" />
</bean>
```

...

In the following project, the RSS data is keyed by title:

```xml
<project name='project' pubsub='auto' threads='4'>
    <contqueries>
        <contquery name='cq' trace='src'>
            <windows>
                <window-source name='src'>
                    <schema-string>title*:string,author:string,link:string,
                        description:string,categories:string,pubDate:date</schema-string>
                </window-source>
            </windows>
        </contquery>
    </contqueries>
</project>
```

1 Edit `src/main/resources/esp.properties` so that it contains your publish/subscribe server information:

   espServer=esp://espsrv01:46003

2 Start your event stream processing server:

   $ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
   -http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup

3 Start the project:

   $ mvn camel:run

**Weather**

This example uses the Camel Weather Component to set up a route that reads weather data for any number of locations and injects it into SAS Event Stream Processing. You should be able to add any locations to the route. The locations can be defined as endpoints as shown in the following example:

```xml
<endpoint id="cary" uri="weather:foo">
    <property key="location" value="cary,nc"/>
    <property key="mode" value="XML"/>
    <property key="units" value="IMPERIAL"/>
</endpoint>

<endpoint id="morehead" uri="weather:foo">
    <property key="location" value="moreheadcity,nc"/>
    <property key="mode" value="XML"/>
    <property key="units" value="IMPERIAL"/>
</endpoint>
```
<endpoint id="chapelHill" uri="weather:foo">
    <property key="location" value="chapelhill,nc"/>
    <property key="mode" value="XML"/>
    <property key="units" value="IMPERIAL"/>
</endpoint>

You can then add these endpoints to your route:

<route>
    <from uri="ref:cary"/>
    <from uri="ref:morehead"/>
    <from uri="ref:chapelHill"/>
    <to uri="ref:publishWeather"/>
</route>

... 

1 Edit `src/main/resources/esp.properties` so that it contains your publish/subscribe server information:
   
   espServer=esp://espsrv01:46003

2 Start your event stream processing server:
   
   $ dfesp_xml_server -model file://model.xml -http-admin <http admin port> -http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup

3 Start the project:
   
   $ mvn camel:run

Setting Up Apache NiFi to Run with SAS Event Stream Processing

Setting Up Apache NiFi

Apache NiFi is a system to automate the flow of data between systems. You can download Apache NiFi from [https://nifi.apache.org/](https://nifi.apache.org/). For more information about Apache NiFi, see the [FAQ](https://nifi.apache.org/faq). 

1 Download Apache NiFi version 1.x into a working directory, `$WORK`, on a UNIX system, `$HOST`.

2 Enter the following commands at the command prompt:
   
   a  cd $WORK
   b  gunzip nifi-1.x-bin.tar.gz
   c  tar xvf nifi-1.x-bin.tar
   d  export NIFI_HOME=$WORK/nifi-1.x
   e  export PATH=$NIFI_HOME/bin:$PATH
3 A NiFi Archive (NAR) file enables components and their dependencies to be packaged together. Copy the SAS Event Stream processing NAR file (nifi-esp-nar-version.nar) from $DFESP_HOME/lib to $NIFI_HOME/lib.

4 Edit $NIFI_HOME/conf/nifi.properties to change the following line to use an available port:
   nifi.web.http.port=8080
   For example, if port 31005 is available, the line reads as follows:
   nifi.web.http.port=31005

5 Start Apache NiFi as follows: $NIFI_HOME/bin/nifi.sh run.
   Note: The DFESP_HOME environment variable must be set before starting Apache NiFi with SAS Event Stream Processing. Update the $NIFI_HOME/bin/nifi-env.sh script to set DFESP_HOME if you are starting Apache NiFi at system start-up.

6 Start Apache NiFi in a browser. Specify the available port that you selected in step 4: http://$HOST:available_port/nifi
   You should now be able to design your Apache NiFi flows. The SAS Event Stream Processing processors ListenESP and PutESP should be available. When processor updates become available, you need to update only the NAR file in $NIFI_HOME/lib.

PutESP Processor

The PutESP processor enables you to publish events from Apache NiFi into an ESP engine. It requires that you specify an ESP project, continuous query, and Source window hierarchy for the processor. Given this information, it can accept FlowFiles in AVRO format. For each object in the AVRO input, PutESP takes any data that maps to a field in the ESP source window schema and creates an ESP event. When all the events have been created, they are sent into ESP.

Table 7 Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pub/Sub Host</td>
<td>The host name of the ESP engine publish/subscribe server.</td>
</tr>
<tr>
<td>Pub/Sub Port</td>
<td>The port of the ESP engine publish/subscribe server.</td>
</tr>
<tr>
<td>Project</td>
<td>The project into which events are published.</td>
</tr>
<tr>
<td>Continuous Query</td>
<td>The continuous query into which events are published.</td>
</tr>
<tr>
<td>Source Window</td>
<td>The Source window into which events are published.</td>
</tr>
<tr>
<td>Record Schema</td>
<td>The AVRO schema for input records. If the schema is not set, it generates an AVRO schema from the publish target Source window in the ESP model. You can get the AVRO schema from a running event stream processing model through the REST interface.</td>
</tr>
<tr>
<td>Block Size</td>
<td>If this is set, PutESP limits the events sent at one time into ESP engine to this number.</td>
</tr>
<tr>
<td>Input Date Format</td>
<td>If this is set, PutESP expects dates in the incoming data to be strings in this format.</td>
</tr>
</tbody>
</table>
### Convert Inserts to Upserts

This should be set if you want to convert any opcodes coming in as inserts to upserts before publishing.

Allowable Values:
- **Yes**
  - Convert inserts to upserts.
- **No**
  - Do not convert inserts to upserts.

**Note:** Defaults to **No**

### Authentication User

The SASLogon authentication user.

### OAuth Token

The token to use for OAuth authentication.

---

### Table 8  Relationships

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>success</strong></td>
<td>If the events are successfully sent to the ESP engine, PutESP sends the originating FlowFile to this relationship.</td>
</tr>
<tr>
<td><strong>failure</strong></td>
<td>When PutESP detects a failure in its interaction with the ESP engine, it sends a FlowFile to the failure relationship.</td>
</tr>
</tbody>
</table>

**Note:** No **Reads Attributes** or **Writes Attributes** specified.

---

### ListenESP Processor

The ListenESP processor enables you to subscribe to any number of windows in a SAS ESP engine and receive events from those windows. The events are collected and converted to AVRO format and sent along the success relationship. The project, continuous query, and window attributes are set on the FlowFile so that downstream processors can act on this information. For example, one can use the **RouteOnAttribute** processor to send events down different paths depending on the originating project, continuous query, and window. You can also specify field values from the ESP events to put on the FlowFile as attributes.

The ListenESP processor requires information for the subscribed ESP windows in order to collect events of interest. You must supply valid ESP engine connection information in the **Pub/Sub Host** and **Pub/Sub Port** properties. For example, if you are running an ESP server on a host called **mymachine**,

```
$ $DFESP_HOME/bin/dfesp_xml_server -model file://model.xml -pubsub 28003
```

your connection properties are as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pub/Sub Host</td>
<td>mymachine</td>
</tr>
<tr>
<td>Pub/Sub Port</td>
<td>28003</td>
</tr>
</tbody>
</table>

Once you have entered the connection information, you need to specify regular expressions to determine the subscribed windows from which to get the event data. To get all of an ESP element from the engine, whether it is projects, continuous queries, or windows, you simply use the `.\*` regular expression. Because many engines contain a single project and continuous query, this is the simplest way to specify that you want them all. Usually, you do not want to subscribe to all the windows in a model.
The following example subscribes to all windows whose name begins with `frontRunning` or whose name is `brokerAlertsAggr` in all continuous queries in all projects.

If you want to grab field values from each received event, you use the **Attribute Fields** property. This is a comma-separated list of ESP field names. For each event that contains a value for each specified field in this property, values are put onto the FlowFile as an attribute.

The following example tells the processor to put the values of the `brokerName` and `symbol` fields onto the FlowFile.

**Table 9  Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pub/Sub Host</td>
<td>The host name of the ESP engine publish/subscribe server.</td>
</tr>
<tr>
<td>Pub/Sub Port</td>
<td>The port of the ESP engine publish/subscribe server.</td>
</tr>
<tr>
<td>Project</td>
<td>This is a regular expression used to specify the projects in the engine to which to subscribe.</td>
</tr>
<tr>
<td>Continuous Query</td>
<td>A regular expression used to specify the continuous queries in the engine to which to subscribe.</td>
</tr>
<tr>
<td>Window</td>
<td>A regular expression used to specify the windows in the engine to which to subscribe.</td>
</tr>
<tr>
<td>Snapshot</td>
<td>Set to <strong>Yes</strong> to get a snapshot of the events in the window. Allowable Values:</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: <strong>Defaults to No</strong></td>
</tr>
<tr>
<td>Output Date Format</td>
<td>If this is set, ListenESP writes date and timestamp fields as strings using this format instead of as numeric values.</td>
</tr>
<tr>
<td>Attribute Fields</td>
<td>These are event field values to put onto the FlowFile as attributes.</td>
</tr>
<tr>
<td>Authentication User</td>
<td>The SASLogon authentication user.</td>
</tr>
<tr>
<td>OAuth Token</td>
<td>The token to use for OAuth authentication.</td>
</tr>
</tbody>
</table>

**Table 10  Relationships**

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>FlowFiles containing ESP events in AVRO format are passed to this relationship upon arrival.</td>
</tr>
<tr>
<td>failure</td>
<td>When ListenESP detects a failure in its interaction with the ESP engine, it sends a FlowFile to the failure relationship.</td>
</tr>
</tbody>
</table>
### Table 11  Writes Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>project</td>
<td>The project containing the current set of events.</td>
</tr>
<tr>
<td>contquery</td>
<td>The continuous query containing the current set of events.</td>
</tr>
<tr>
<td>window</td>
<td>The window containing the current set of events.</td>
</tr>
<tr>
<td>path</td>
<td>This is set to the model container of the event: project or continuous query.</td>
</tr>
<tr>
<td>filename</td>
<td>This is set to the window of the event with a timestamp appended.</td>
</tr>
</tbody>
</table>

**Note:** No Reads Attributes specified