Connector Accelerator

Enabling Faster Third Party Integrations to SOI

Prepared by: Jeff Morris, CA Technologies Engineering Services
December 2014

Table of Contents

Overview

Connector Accelerator is a full-fledged Catalyst 2.5-based connector designed to reduce complexity and time to value for SOI customers attempting to integrate CIs and events from third party data sources for which no engineering supplied connector already exists.

It leverages an easy-to-use declarative XML configuration file to drive a scheduling mechanism for CI and alert discovery. It supports data sources that are either file-based, script-based or JDBC-based. File format support includes both CSV and XML.

In addition to adding a scheduling mechanism which reduces the need for customers to be writing code for this, we provide the "SOIPost" client which is a Java-based client tied to a command script whereby customers can "post" alerts and CIs to SOI via ConnectorAccelerator. This tool is analogous to those supplied by other event management and necessary for customers migrating from these tools. ConnectorAccelerator also supplies a REST interface to allow posting of CIs/Alerts.

In summary, the key features of Connector Accelerator are:

1) Schedule-based CI discovery and Alert polling
2) HTTP Listener supporting REST API and SOIPost client for ad hoc CI/Alert posting
3) EIF Listener to support Tivoli conversions
4) XML-configurable
5) CSV, XML, JDBC data sources supported

1: Installation

Install Connector Accelerator

Connector Accelerator can be installed on any machine where you have previously installed the SOI Integration Services. You can check for this by making sure that you have a "CA SAM Integration services" service.

Connector Accelerator is distributed as a .zip file, **ConnectorAccelerator.zip**. As there is not an installer wrapper, deployment is essentially a manual process accomplished by executing the following steps:

1) On the target connector machine, stop both the "CA SAM Integration Services" and "CA SAM Event Management" services
2) Extract the .zip file in to the <SOI> directory (The .zip file contains the files in the proper directory hierarchy already as long as you extract to the <SOI> directory)
3) The main connector initialization file will be deployed to <SOI>/resources/Configurations. By default it is named "ConnAccel_hostname.xml"
    a. Rename this file to change "hostname" to the name of the host where the connector is deployed (i.e. ConnAccel_sag-soi.xml)
    b. Edit this file and change the "MdrProdInstance" and "name" attributes of the <Silo> tag to also reflect your hostname. For example, if your hostname is "MORJE10", then something like the following.

    ```
    <Silo MdrProdInstance="ConnAccel-MORJE10"
    MdrProduct="CA:77001" State="Enabled"
    name="CA:77001_ConnAccel-MORJE10">
    ```

    c. Modify the "configFileName" attribute of the "ConnectionInfo" tag to point to the full path to your SOI installation's resources directory. The default connaccel-config.xml is deployed in this directory. (You can optionally move this file and rename it as long as you modify the setting in this master config file). For example:

    ```
    <ConnectionInfo
    configFileName="f:/opt/ca/soi/resources/connaccel-
    config.xml"/>
    ```

4) Open the file <SOI>/jsw/conf/SAM-IntegrationServices.conf with a text editor and locate the section with many entries starting with "wrapper.java.classpath.<nn>=…..". Go to the bottom of this section and do the following:
    a. Add this entry:

    ```
    wrapper.java.classpath.<nn>=../../lib/ext/jdom-2.0.5.jar
    ```

    Don't add '<nn>' literally… use the next number in sequence in the list

    b. If you are planning to use the EIF listener to integrate data from IBM Tivoli, add this entry:

    ```
    wrapper.java.classpath.<nn>=../../lib/ext/portmap.jar
    ```

    c. If you are going to use ConnectorAccelerator to connect to Oracle, add this entry:

    ```
    wrapper.java.classpath.<nn>=../../lib/ext/ojdbc6.jar
    ```

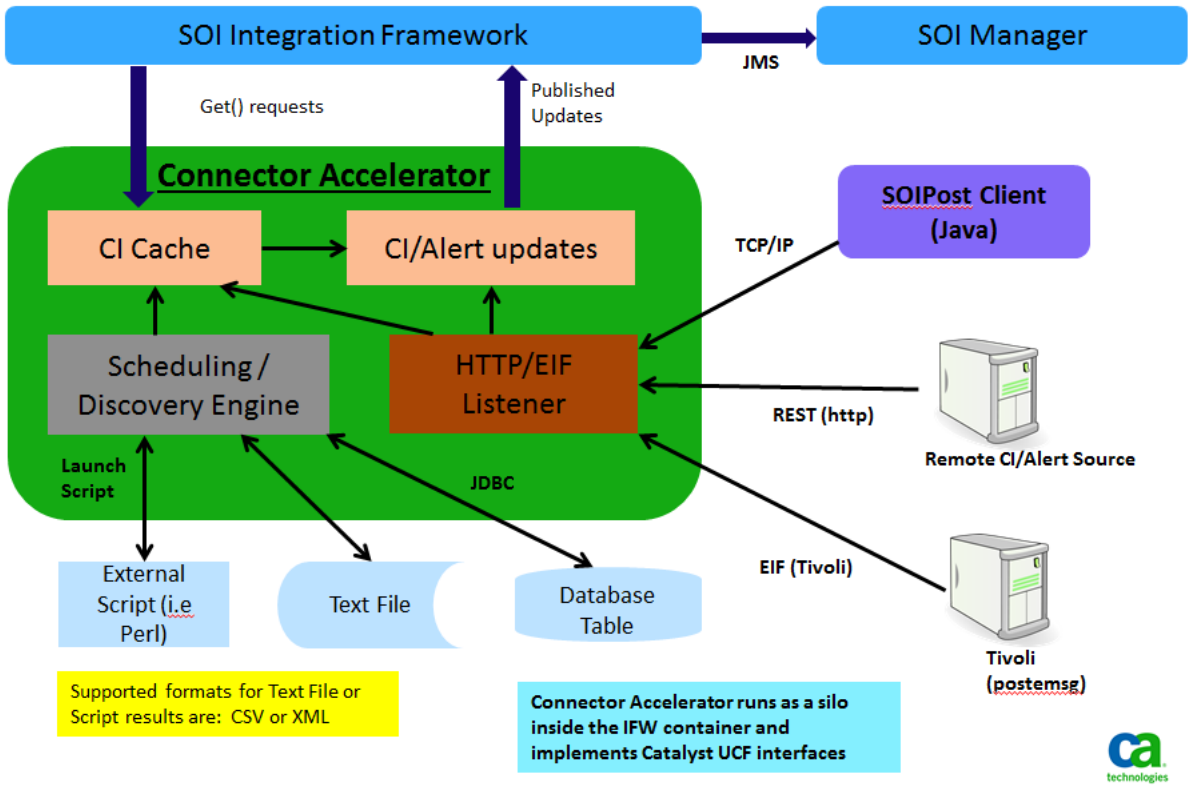
5) The connaccel-config.xml file, located in <SOI>/resources, is the main configuration file driving the functionality of the Connector Accelerator. By default it only is set up with an http listener to receive requests from the SOIPost client. The default listening port, specified in this file, is 7777.

You may change this to any port that you desire. You will also need to modify your SOIPost command script accordingly. More details on this in the **Configuration** chapter

After you have taken all of these installation steps and modified your **connaccel-config.xml** file accordingly, Connector Accelerator will start when you restart SAM Integration Services

2: Functionality and Concepts

Basic Architecture and Components



The Connector Accelerator contains the following primary components as outlined in the above diagram:

1) **Connector Data Source** – This is a data source for CIs or Alerts. These data sources can be either text files on disk (CSV or XML format), a database table accessed via JDBC, or an external script that executes and returns results in either a CSV or XML format. Each Connector Data Source is configured via XML using the **<Connector>** tag within the connaccel-config.xml file. (See the **Configuration** chapter for details on this configuration

2) **Listeners** – A Connector Data source can be configured to have a listener for external events representing CIs and Alerts. Currently only two formats are supported

    a. **HTTP –** This is the core REST API that can be used directly by users and is also used by the SOIPost Java client.

    b. **EIF –** This listener supports the Tivoli EIF format and allows Connector Accelerator to listen for incoming EIF traffic. Events posted by the "postemsg" command are supported with this listener

3) **Scheduling / Discovery Engine** – This is the core engine of the Connector Accelerator. At startup it loads the core Connector configuration file and configures one "connector" for each Connector Data Source. It then manages and gathers the CIs and alerts according to the schedule specified and returns / posts them to the SOI Integration Framework in response to the get() requests or by publishing CIs and Alerts via the subscription interface to the framework.

Connector CI and Alert Data

The key concept behind any of these "on-the-fly" connectors supported by Connector Accelerator as Connector Data Sources is that we are trying to "discover" CIs and Alerts from a remote data source. The script or file or DB query used as the data source is simply designed to extract a "map" of name/value pairs representing the attributes of the CI or Alert in question. Ultimately the "name" portion of each of these map elements must end up in USM (Unified Service Model) format in order to be processed by SOI.

When building / configuring your data source, you have the option of using connector policy (i.e. modifying the connaccel-policy.xml file) to transform your raw data into USM, or you can configure your XML, CSV, scripts, etc. to "speak USM" natively.

There is also a transformation concept (<ColumnMap> tag) built into the Connector Accelerator configuration file to make basic transformations more simplified and avoiding having to modify connector policy which uses a more complex XML syntax

XML Format for file-based CI/Alert data

When choosing the "XML" format for your CI/Alert files or script results, the XML must conform to the following structure:

```
<ConnnectorData>
<CI>
      <property tag="aaaaa" value="bbbbb" />
      <property tag="ccccc" value="ddddd" />
</CI>
<Alert>
      <property tag="aaaaa" value="bbbbb" />
      <property tag="ccccc" value="ddddd" />
</Alert>
```

All elements must be of type <CI> or <Alert>

The "tag" refers to the attribute name and "value" its value. (This is the same syntax used with the SOI Universal Connector and is maintained to make possible conversions easier)

CSV File Format

When using the CSV file format, the parser will assume that the first row in the CSV file contains the attribute names separated by commas. Then rows 2 through n should contain the data, separated by commas. If you need to embed a comma in a string, make sure to embed the entire string in quotation marks to avoid parsing problems. As is normal with CSV files, the values will be mapped to their attribute names in a columnar fashion.

Built-In and Default Values

One of the core design concepts behind Connector Accelerator is to follow the 80/20 rule and simplify the 80% case. One of the observations that has been made is that the types of 3rd party integrations that usually need to be "spun up" in a case like this are often following a basic principle of trying to discover and alert on "host" or "ComputerSystem" CIs. Following this observation much of the "default" behavior is set up to make this easier.

The following are "built-in" or "default" values that are automatically added to the input map by Connector Accelerator and are available for use in transformations in Connector Policy

casource – This value is used by the "postAlert" function of the REST API. Whenever an alert comes in via the REST API (including being sent from the SOIPost client), an attribute called "casource" is added to the map with the value "postAlert". This is used by "Classify" directives in the base connector policy to enable the default transformations in conjunction with these CIs and Alerts

caccid – The attribute "caccid" (which stands for "Connector Accelerator ID") is set to the value of the "id" attribute of the <Connector> tag for the Connector Data Source from which the Alert or CI was published via Connector Accelerator. This is so that if you have multiple data sources you can use your own Classify directives to separate the transformation logic for the different data sources.

user_class – This attribute should be used if you are trying to assign a ClassName for a CI via SOIPost or REST. The OOB policy will transform it to the USM "ClassName"

user_severity – The severity attribute from SOIPost is passed in the input map as "user_severity", and the default transformation rules in Connector policy use this attribute to map to the USM "Severity" attribute

hostname – This attribute is required for SOIPost and REST, and is used by default as the unique key and host name of the CI. Via this mechanism the default policy will map this to "PrimaryDnsName" and use it as the MdrElementID

CI / Alert Unique Keys

USM specifies that each CI and Alert must have a unique value across the enterprise. In USM this unique combination is the concatenation of the attributes "MdrProduct", "MdrProdInstance", and "MdrElementID". For Connector Accelerator, "MdrProduct" is always "CA:77001", and "MdrProdInstance" is taken from the main connector configuration file (in <SOI>/resources/Configurations) for the ConnectorAccelerator. This is usually the name of the host the Connector Accelerator is running on.

This leaves "MdrElementID". This is where your connector must set the unique key of the CI or Alert.

If you don't set MdrElementID specifically from your data source, the default behavior is according to the following:

For CIs posted via SOIPost or REST API:

1) If the "MdrElementID" attribute is present in the map, it is used
2) If "ci_unique_id" is present in the map, it is copied to MdrElementID and used as the key
3) If the user has not changed the ClassName from the default of ComputerSystem, then the value of the "hostname" attribute is copied to MdrElementID and used as the unique key
4) If the ClassName is "Application", then a concatenation of the "hostname" and "appname" attributes are used to compute the unique key

NOTE: The goal is to make this unique key derivation based on other attributes both configurable and to add more out-of-the-box combinations. This functionality is on the short term roadmap and will be coming soon, but in the meantime for advanced, non-ComputerSystem or Application use cases, it is the user's responsibility to set MdrElementID appropriately.

For Alerts posted via SOIPost or REST API:

For an Alert the unique key (MdrElementID) is always a combination of the CIs unique key combined with an "alertKey" value

1) If the attribute "alertKey" is present in the input map, use it for alertKey
2) If not, generate "alertKey" as a hash of the Summary attribute. (This is based on the theory that for many MDRs what distinguishes on alert from another on the same CI is the summary text).

3) Create the MdrElementID with the following pattern **<ci_unique_key>-<alertKey>** . In other words, the unique key (MdrElementID of the CI which the alert is bound to), plus a hyphen, plus the alert key derived in steps 1 or 2 above

Just as a point of reference, for all alerts posted through SOIPost or the REST API, the **AlertedMdrElementID** of the alert is set to the **MdrElementID** of the associated CI

For Alerts and CIs incoming from a Connector Data Source

1) If MdrElementID is present, use it
2) Look in the <Params> tag for the Connector Data Source for the "**keyColName"** attribute. If present, grab that attribute from the input map and use it
3) If neither of the above produce a result, there will be no MdrElementID, and the CI or Alert will be rejected

Alert De-Duplication via Summary Transformation

This functionality stems primarily from a legacy event management use case that we saw happening with several customers participating in the beta program.

As discussed above, in a generic event management sense, the easiest way to distinguish unique alerts from one another on the same device while avoiding noise at the same time is to use the "Summary" value in conjunction with the unique key of the CI. Connector Accelerator uses this technique by default.

This works fine as long as the Summary values contain basically static text, such as "HIGH CPU UTILIZATION". Unfortunately, it falls apart when the event source uses variable data in the Summary message, such as "HIGH CPU UTILIZATION – 88%". In this case, if another alert comes in on the same device with the Summary as "HIGH CPU UTILIZATION – 92%", it will be treated as a separate alert, which will cause an explosion of alerts within SOI for the same condition.

The solution to this use case is to allow you to configure transformation rules against the Summary field via regular expressions that will "remove" the variable data from the Summary text and "de-duplicate" the alerts.

This is accomplished in the Connector Accelerator using the <Transform> tags under the <AlertSummary> element. (See the Configuration chapter for more details on this syntax)

Each <Transform> tag contains both a "pattern" attribute and a "replace" attribute. When these <Transform> tags are present, the incoming "**Summary**" attribute value is compared against each "pattern" value using regular expressions. If a match is found the new "Summary" value is set using the "replace" attribute, according to standard regex syntax.

The **"Message"** attribute will still contain the complete message, presuming no other overrides or changes were made to change the default behavior

Additionally, when such a de-duplication transformation occurs and the resulting Summary matches an active alert in the Connector Accelerator Alert cache, Connector Accelerator considers this to be a duplicate, and instead of publishing a new alert it publishes the alert with a status of "Update". Since some customers are interested in the number of occurrences of the same alert, we also keep an internal counter in the cache and on the updates publish this in a map attribute called "**NumOccur"**. You can map this to a USM property, such as "userAttribute5" using connector policy or the <ColumnMap> directive on the connector so it can be visible in the SOI console and usable in alert escalation policy.

When the alert is cleared, i.e. when the alert with Severity="Normal" is received, the counter is cleared and the alert is removed from the internal cache.

3: Configuration

Basic Configuration Details

The main configuration file that controls Connector Accelerator behavior is **connaccel-config.xml**. The default location is in the <SOI>/resources directory. All of this can be modified by changing the **configFileName** attribute of the **ConnectionInfo** tag in the connector's initialization file (see **Installation** chapter)

The root tag for the file is <**ConnAccel>** and this should not be changed.

The file can then contain one or more **<Connector>** tags which correspond to each unique data source for the Connector Accelerator

Additionally, there can be an optional **<AlertSummary>** tag containing one or more **<Transform>** children representing alert de-duplication summary transformations

NOTE: There must be a minimum of one <Connector> element. If you are not using any external data sources and only want to configure the http listener, you should use the "base" connector type (see below and in the **Examples** chapter)

<Connector> Tag

There can be one or more connector tags in the configuration file, each representing one data source. The type of data source is determined by the attributes as described below:

id – This is unique identifier for the connector data source. It is used internally by the Connector Accelerator to separate one data source from another. Its value also stored in the "**caccid**" attribute of the value map that is published to the integration framework, making it accessible in connector policy so that different transformation rules can be written based upon your different data sources

type – This indicates the type of data source. Valid values are "base", "file", "script", "db", or "Java" and work as follows:

> **file** – CIs and alerts are stored in a file on disk to be polled by Connector Accelerator
>
> **script** – The data source for this connector is an external script, to be launched on a schedule by the discovery engine
>
> **db** – CIs and alerts will be retrieved from a database query
>
> **base** – This is used when there is no external data source and we are configuring the Listener only
>
> **Java** – This is a mechanism to attach custom connector discovery code written in Java. Details of this advanced functionality are outside the scope of this document

format – This attribute applies to the "file" and "script" types and specifies the format of the results or script. Valid entries are "xml" or "csv"

ciPollInterval – Specifies the interval between CI "polls". For "file" type, this is how frequently Connector Accelerator will check the CI file, for "script" type, how often the CI script will be executed, and for "db" type, how frequently the CI query will be run. The value is specified in minutes.

alertPollInterval – Specifies the interval between Alert "polls". For "file" type, this is how frequently Connector Accelerator will check the Alert file, for "script" type, how often the Alert script will be executed, and for "db" type, how frequently the Alert query will be run. The value is specified in minutes.

File type attributes

These attributes of the <Connector> tag are specific to connectors of type “file”:

cipath – The file location of the CSV or XML file containing the CI data. (required)

alertpath – The file location of the CSV or XML file containing the Alert data (optional). Since often times alerts will come in via SOIPost or the REST API, an alert file is not required

Script path attribute

For “script” data sources, the external scripts have to be located on the file system. By default Connector will look in the directory <SOI>/resources/scripts for the specified scripts. To override this default you can set the attribute “**path”** in the <Connector> tag to specify an alternate path location.

Absolute paths can also be specified in the <Script> tags

<Script> tag

For connectors of type “script” there can be between one and four <Script> tags underneath the <Connector> element. The <Script> tag has three attributes, specified as follows:

Type – The type of script. It must be one of the following four values: “ci_initial”, “alert_initial”, “ci_update”, “alert_update”. These values have the following meanings:

> **ci_initial** – This script is executed once at the beginning of the lifecycle when Connector Accelerator first initializes the data source. One <Script> tag of this type is required.
>
> **ci_update –** This provides for an optional alternate script to be run at each “ciPollInterval” after the initial run. Presence of the ci_update type script is optional, and if omitted the **ci_initial** script will be run on each interval
>
> **alert_initial** – This script is executed once at the beginning of the lifecycle in response to the get() call for type=”Alert” when the Connector Accelerator first initialized the data source.
>
> **alert_update** – This script is an optional alternate script to be run at each “alertPollInterval” after the initial run. In the absence of this entry, the **alert_initial** script will be run at each interval if present

command – This is the executable command to be run to launch the script, i.e. “myscript.cmd” or “perl.exe”

script – This is the “script” which will be passed as an input value to “command” (i.e. “myscript.pl”)

(The combination of command + script constitutes the full command passed to the operating system)

<Params> Tag

The <Params> tag is an optional child tag underneath the <Connector> element, and is used to specify one or more parameters, specified as name/value pairs via attributes, to be passed on to the Connector Data Source. The different connector types expect and/or support different parameters as follows:

keyColName – Specifies the name of a column from the data map to be used as the unique “key” for the CI or Alert. By default ConnectorAccelerator looks for the USM property “MdrElementID” in the input data map to use as the key. However by specifying a value for this parameter, you can use a different named column. Be aware that if you are using this in conjunction with the “<ColumnMap>” tags (see below) to perform “before policy” mappings, you must use the name in the “to” entry from the ColumnMap as the value for “keyColName” since this lookup is done after that mapping takes place

These values are used with the “db” connector data source type

dbType – Database type – currently accepts “oracle” or “mssql”

dbHost – Database server name

dbPort – The port on the DB server to connect to

dbName – The database name

oraSID – (For Oracle Only) – the Oracle SID used in the JDBC connection string

dbUser – User name for DB connection

dbPw – Password for DB connection

<ColumnMap> Tag

The <Connector> tag can contain one or more child <ColumnMap> elements. These are used for what we call “before policy” mappings. This allows you to take map columns from the input source and rename them easily and facilitates converting values to USM without having to modify Connector policy directly.

The <ColumnMap> tag supports 2 attributes:

from – The column name in the raw input map. For a CSV file, this would be the name in the column header. For XML input it will be the “tag” value. For DB input it will be the column name from the database.

to – The name of the property to “rename” the “from” value to.

For example, let’s assume we have a CSV file with a column called “Name” which represents the name of a computer system host for which we want to create a CI. We then want to map this value to the USM property “PrimaryDnsName”. To do this with ColumnMap, we simply add the following element:

```
<ColumnMap from=”Name” to=”PrimaryDnsName” />
```

<Listener> Tag

The <Listener> element is a child of the <Connector> element, and allows you to attach a listener to that connector’s processor. The <Listener> tag has 2 attributes as follows:

type – The listener type. Supported values are “http” or “eif”

port – The port to configure the listener.

The most common use of the <Listener> tag is to set up the http listener for use with the SOIPost utility and/or the REST interface to Connector Accelerator

Here is the example of that configuration:

```
<Listener type=”http” port=”7777” />
```

Any value can be chosen for the “port”. 7777 was just randomly chosen as a default. Just make sure that the Listener config here matches what is used in the SOIPost client script.

<AlertSummary> and <Transform>

If used, the **<AlertSummary>** element must appear as a direct child of the **<ConnAccel>** element, meaning that is global in scope and applied to all Connector Data Sources. One and only one <AlertSummary> element is needed, and it can have one or more **<Transform>** child elements that

specify the alert de-duplication transformation rules, as described in the **Functionality and Concepts** chapter.

<AlertSummary> takes no attributes and only functions as a container for the <Transform> elements.

All <Transform> elements must have the following two attributes:

pattern – This is a pattern in regular expression format that is matched against the Summary attributes of incoming alerts looking to be transformed to remove variable data

replace – This is the replacement pattern to replace the Summary data with if a match is found on the pattern. It is also in regular expression format and can and generally will utilize matching groups for the transformation

As an example, let's say we are receiving noise from Summary messages similar to the following:

"**HIGH CPU UTLIZATION (83%) on device MyServer"**

And then later we receive

"**HIGH CPU UTLIZATION (92%) on device MyServer"**

What we want to do is remove the "(83%)" and "(92%) from these message because it is variable data and we don't want to create two duplicate alerts as this is really the same condition with a different value.

To fix this, we can use <AlertSummary> and <Transform> in the following manner:

```
<AlertSummary>
      <Transform pattern="(HIGH CPU UTILIZATION )\(.*?\) (on device .*)"
replace="$1$2" />
</AlertSummary>
```

The way this works is as follows: (sorry if this is review for you regex experts)

1) The first portion, **(HIGH CPU UTILIZATION)** is the first "matching group" and matches the static text at the beginning of the Summary. The ')' characters in this portion are not literal, but special characters telling regex where the beginning and end of the matching group is

2) The next portion of the pattern is **\(.*?\)**. The '\(' and '\)' represent the literal parentheses from the Summary. The backslash (\) character tells regex to treat the following character literally and not use its special meaning as a matching group. The '.*?' matches anything between the parentheses, which is really the variable data including the '%' sign.

3) Then the following portion **(on device .*)** matches the remainder of the Summary. The .* matches the variable data containing the device data, but this is variable data that we don't want to filter out.

4) We can see that this pattern does indeed match the actual Summary text in both of the examples above. Since it matches, when Connector Accelerator sees it, it will replace the Summary using the replace parameter, which as you can see is a simple expression, **"$1$2".** This simply means take matching group 1 and matching group 2 from the pattern and concatenate them together.

5) As we saw in steps 1) and 3) above, matching group 1 = "HIGH CPU UTILIZATION " and matching group 2 = "on device MyServer" so the resulting Summary text will be

   HIGH CPU UTILIZATION on device MyServer.

6) When the second alert is received since this alert is already present (based on the Summary being the same), we will publish it as an Update, and NumOccur will be set to 2 in the input map.

SOIPost Client and REST API

SOIPost Installation / Configuration

The SOIPost client is a Java client that is distributed as a .zip file. The zip file contains an "SOIPost" folder so you can extract it to any parent directory on your file system and all of the SOIPost files will be contained in the SOIPost folder underneath.

There is a simple wrapper script, SOIPost.cmd that is used to execute the client. This script assumes that the **java** executable is somewhere within the user's executable path. If it is not, you will need to either add java to the PATH or modify the SOIPost.cmd script and add an absolute path to the line which calls the "java" executable.

NOTE: SOIPost supports only Java 6 or Java 7 JREs

SOIPost Usage

SOIPost is a command-line parameter driven client. The core usage pattern for the client when running with SOIPost.cmd is as follows:

```
SOIPost [-debug] –S <host or IP> -p <port> –r <severity> -m <message>
hostname=<hostname> [<parm1>=<value1>,[<parm2>=<value2>,…]]
```

The parameters and their descriptions are as follows:

-S (or –server) – This is the connector server where Connector Accelerator is running and the HTTP Listener configured

-p (or –port) – This is the port the Connector Accelerator is listening on for HTTP. This must correspond to the "port" attribute of the <Listener> tag in the Connector Accelerator config file

-r – This represents the severity of the Alert. You can use either the exact USM values for severity. Additionally some common values for severity are supported via the out-of-the box connector policy. These are NOT case sensitive and map as below. Also the severity passed via the SOIPost is given the map attribute "**user_severity**" in the input map in Connector Accelerator. It is then mapped to the USM attribute Severity using the connector policy. If you wish to customize connector policy you need to use this "user_severity" attribute.

"**fatal", "down"** : maps to "Fatal" in USM (shows as "Down" in the SOI UI)

"critical" : maps to "Critical" in USM

"**major"** : maps to "Major" in USM

"minor", "warning", "unknown" : maps to "Minor" in USM

"**normal", "clear", "harmless" :** maps to "Normal" in USM (and functions to clear the alert)

-m – Because one key use case of SOIPost is to provide compatibility with legacy event management systems, by default the value of this parameter is passed to BOTH the Summary and Message attributes in USM. If you want to have a value for "Message" that is different than that of "Summary" you should add "Message=<message>" to the end of your SOIPost command line. In this case the value of the "-m" parameter will only be passed to the Summary attribute

hostname – This is another parameter meant to address the 80/20 rule and legacy event management systems, for which the assumption is that alerts being posted via the command line utility are destined for ComputerSystem CIs. The "hostname" parameter is required and is via OOB connector policy mapped to the "DeviceDnsName" parameter in USM. To extend the behavior of Connector Accelerator and SOIPost to other CI types you can extend the OOB connector policy, as well as pass the ClassName of the CI on the command line by adding "user_class=<classname>" at the

end of the command line. Hostname in these cases often will then get mapped to DeviceDnsName in the case of "child" CI types, such as Application.

-debug – This enables debugging with the SOIPost client. If you add this parameter, the complete HTTP response from ConnectorAccelerator is output. This can be useful if you are having errors posting the CIs / Alerts

The above parameters, with the exception of –debug, are the ones that are required and the minimum required to generate an alert on a device. Supplying these parameters is generally sufficient in the 80% case where we are dealing with alerts on ComputerSystem CIs.

For all other cases, SOIPost provides the ability to pass an extended input map by adding any number of parameters on the command line. These are done using the syntax

<parmN>=<valueN>

Where <parmN> represents the name of the attribute and <valueN> is its value. All of these name/value pairs then become part of the "input map" passed to ConnectorAccelerator. They can be specified either in a generic syntax which is more understandable by the users of SOIPost, or optionally can be specified in USM directly. If these attributes are not specified in USM yet are destined for USM properties, the appropriate entries must be added either to the Connector policy or using the <ColumnMap> tag for simple attribute name mapping

Using the REST API

The REST API which is included with Connector Accelerator allows users to send alerts and CIs to Connector Accelerator via HTTP using either a browser, a utility such as **curl** or via a programmatic method such as Perl or Java which can make HTTP connections

In fact the SOIPost client which is shipped with Connector Accelerator uses the same REST API to communicate with Connector Accelerator.

The basic URL syntax for posting to Connector Accelerator via the REST API is:

http://<server>:<port>/<function>?<param1>=<value1>&<param2>=<value2> ….

<server> represents the Connector Accelerator server

<port> represents the Connector Accelerator port as configured in the <Listener> tag

<function> is the API function and is one of "**postAlert"**, "**delete"**

When calling the REST API, any errors or missing parameters are reported directly in HTML in the http response.

postAlert

The **postAlert** function is the most common and is the same one used by the SOIPost client to send alerts.

Param1, 2, etc. must be sent so create the input map. The following parameters are required, similar to what is required by SOIPost.

NOTE: When using the REST API, the URLs and parameter names described below **ARE case-sensitive**

user_severity or **Severity** – You must specify the severity (corresponding to the –r parameter of SOIPost). If you assign it to user_severity then you can use all of the extended values and they will be transformed accordingly via connector policy. If you choose to use "Severity", your values must conform to USM

hostname – As with SOIPost, hostname is a required attribute

Summary – This is the message summary and is the required USM summary. If "Message" is not present, Summary will be copied to Message, but you have the option of including the Message parameter if the values are different

For example,

http://soi:7777/postAlert?hostname=Myserver&Severity=Minor&Summary=This%20is%20a%20Test

NOTE that because we are using HTML for rest we have to use %20 to represent a space in the summary / message.

delete

The "**delete"** function provides away to remove a CI from SOI that may have been posted erroneously from SOIPost, or is in fact, no longer part of the environment

The delete function contains one parameter, **id**. The "id" parameter corresponds to the MdrElementID of the CI to be deleted

For example,

http://soi:7777/delete?id=MyServer

This would delete the "MyServer" CI presuming you are using the host name as the unique ID

Alert/CI Workflow for SOIPost (and REST)

The following is the workflow that takes place when the SOIPost client is executed. (The workflow is identical for the "postAlert" function of the REST API, beginning with step 3. Step 2 is you making the REST call.)

1) SOIPost takes the input parameters from the command-line and converts them into an **input map** consisting solely of name/value pairs of map attributes and their values.
2) An HTTP connection is established with Connector Accelerator on its listening port and the input map is transmitted to Connector Accelerator
3) Connector Accelerator performs validation on the input map, ensuring that all required attributes are present. If not, an error response is sent back to the client
4) Presuming that the map passes validation, TWO input maps are passed to SOI, one representing the data as a CI (Item), and one representing the data as an Alert. This is the critical piece of the workflow and allows for the creation of CIs on the fly. As you know, for an alert to be posted to SOI, there must be a corresponding CI to which to "bind" the alert. Using this "two map" (aka "two pass") workflow ensures that the CI will also be posted. If you are familiar with the SNMP connector for SOI, it uses a similar workflow when processing incoming traps.

One other function of the workflow that is worthy of note: For each CI/alert posted via SOIPost or REST, a unique GUID is created to identify that client request. This attribute, called either "alertGUID" for Alerts or "ciGUID" for CIs is added to the map passed to SOI, and functions as an audit trail mechanism for troubleshooting purposes. If for some reason the alert or CI doesn't end up in SOI properly, log files will contain reference to this GUID. The GUID is also passed back to the client in the response for this tracking purpose.

Example Scenarios

This section contains some examples of Connector Accelerator configuration files to achieve some common use cases. For reference on the syntax see the **Configuration** chapter

Basic SOIPost / http Listener only

This use case is a basic event management use case using only the SOIPost client or REST API. In other words there is no preliminary CI discovery through the file, script, or DB mechanisms. It is the most simple use case, but probably also the most common.

The connacel-config.xml file for this scenario is shown below. You will notice there is an <AlertSummary> section demonstrating how the alarm de-duplication can work, but this is completely optional.

A few things you will notice about this configuration:

1) The “type” of the connector is “Base” since we are not implementing any discovery. The “id” is set to “Base” also, but this has no relevance. This value can be set to anything. It simply needs to be unique among all <Connector> elements in the configuration file

2) There is a <Listener> tag indicating that we will be listening for http requests on port 7777. This is what enables the SOIPost client and the REST API. The <Listener> tag must be a child of a <Connector> element. In our case we only have the one <Connector>, which is only present to support the Listener

3) There is one <ColumnMap> element for our <Connector>. This automatically maps the attribute “appname” from the input map to the USM property “ProcessDistinguishingID”. This is a nice shortcut for use with Application CIs. ProcessDistinguishingID is one of the correlatable properties that are required for Application CIs. By adding this mapping when you use the SOIPost client or REST API, you can pass the value using the easier-to-type identifier of appname and the ConnectorAccelerator will convert it to USM for you.

```
<?xml version="1.0" encoding="UTF-8"?>
<ConnAccel>
	<Connector id="Base" type="Base" >
		<ColumnMap from="appname" to="ProcessDistinguishingID" />

		<Listener type="http" port="7777" />
	</Connector>

	<AlertSummary>
		<Transform pattern="(HIGH CPU UTILIZATION )\(.*?\) (on device
.*)" replace="$1$2" />
		<Transform pattern="(.*experiencing high CPU utilization)\.High
CPU [\d\.]+ for.*minutes.*" replace="$1" />
		<Transform pattern="(DISK.*WARNING.*on.* )is.*" replace="$1" />
		<Transform pattern="(Server .*: The amount of swap space free
)is now .*" replace="$1 is low" />
	</AlertSummary>
</ConnAccel>
```

CSV file + HTTP Listener

In this scenario, we want to perform CI Discovery using a CSV file containing our CI data, but we also want to receive alerts using SOIPost. So we need to configure Connector Accelerator with a Connector of type “file” and format “csv” that also contains an http Listener.

In our use case we have a CSV file with the following column headers in the first row, and each row in the CSV file contains the data corresponding to these columns.

Name,IP Address,Temp Key,Lifecycle Status,Location,Class,Operating System

These are all host machine devices which we want to map to ComputerSystem in SOI. Some of these columns, such as Temp Key, Lifecycle Status, etc. are values that, even though they are not part of USM, are important for us and we need to use them to group alerts in SOI for different response teams as well as utilize this data when integrating with our ticketing system. To satisfy these requirements we can map these to user-defined attributes in SOI.

The configuration file below illustrates this example, and here are some highlights of how this configuration implements these requirements declaratively. (Below this config file we describe the minor changes needed in the Connector policy)

1) The connector id is set to "csvdata". This value will be passed to the input map attribute "caccid" and we can use this in our connector policy file to enable customizations for this connector
2) We set type="file" and format="csv" to enable the CSV file input use case
3) "cipath" points to where the .CSV will be located.
4) "ciPollInterval" = 60 tells Connector Accelerator to check for new CIs every 60 minutes. Presumably there is some external script which is updating this CSV periodically with new device data.
5) There is no "alertpath" specified because we are not publishing alerts via CSV. Instead we are going to publish them via SOIPost. (Notice alertPollInterval is still present. This is OK, but it will be ignored since there is no alert source for the CSV connector)
6) The <ColumnMap> entries facilitate easy mapping from the columns in our CSV file to the USM properties needed by SOI.
    a. The key thing to note is that the "Name" column is mapped to "PrimaryDnsName". This is the key correlatable property for the ComputerSystem class and is definitely required
    b. We also map "IP Address" to "PrimaryIPV4Address" which is also a correlatable property in USM of lower priority
    c. The only other USM property we map is "Operating System" to "PrimaryOSVersion"
    d. You can see that the other properties are mapped to CIuserAttribute1, 2, and 3. These are the user-defined CI properties
7) In the <Params> element we set the keyColName to PrimaryDnsName. This will cause ConnectorAccelerator to also copy this value in to the unique key, MdrElementID. Notice that we must use the property name "post-ColumnMap" for this directive
8) The <Listener> tag is the same we saw in our previous example and enables the Alerts to be sent via SOIPost or REST
9) Finally, since our input file doesn't contain the ClassName (in USM terms), as it assumes everything is a ComputerSystem, we decide to automatically set this using Connector Policy (see below).

```
<?xml version="1.0" encoding="UTF-8"?>
<ConnAccel>
       <Connector id="csvdata" type="file" format="csv"
cipath="f:/soi/resources/csvfiles/cidata.csv" ciPollInterval="60"
alertPollInterval="2" >
              <Params keyColName="PrimaryDnsName" />

              <ColumnMap from="Name" to="PrimaryDnsName" />
              <ColumnMap from="IP Address" to="PrimaryIPV4Address" />
              <ColumnMap from="Lifecycle Status" to="CIuserAttribute1" />
              <ColumnMap from="Operating System" to="PrimaryOSVersion" />
```

```
            <ColumnMap from="Class" to="CIuserAttribute2" />
            <ColumnMap from="Location" to="CIuserAttribute3" />

            <Listener type="http" port="7777" />
      </Connector>
</ConnAccel>
```

Our final challenge is that we need to automatically set all CIs to have their USM ClassName be ComputerSystem. Below we have the changes that need to be made to the connector policy file to implement this use case. This file is located at <SOI>/Core/CatalogPolicy/connaccel-policy.xml

NOTE: Since this use case it seems may become common, in the next release we are looking at an addition to the connaccel-config.xml to support simply adding static values to the map, but for the meantime, this must be accomplished with a policy change. The other way would be to populate a column called user_class in the CSV file.

```
<EventClass name='Item' >
      <Classify>
            <Field conditional="casource" input="casource"
pattern="postAlert" output="eventtype" outval="GenericItem" />
            <Field conditional="caccid" input="caccid" pattern="csvdata"
output="eventtype" outval="csvitem" />
      </Classify>
```

This is the first section. Note that only the line in yellow needs to be added to the file – the other lines were already present. This line uses the value “caccid” which is automatically populated by Connector Accelerator with the “id” value for our data source, which we set above to “csvdata”. As a result, per this directive above, all CIs coming into the connector from that data source will be mapped to the EventClass “csvitem”

```
<EventClass name="csvitem" extends="Item" >
   <Format>
      <Field input="" output="ClassName" format="ComputerSystem" />
   </Format>
</EventClass>
```

This is a simple event class that simply sets the ClassName attribute of the map to ComputerSystem.

Publishing Alerts on Application CIs with REST

Veering outside the norm a bit and getting away from the outdated “all CIs are ComputerSystems” paradigm, let’s explore the use case where we want to publish and alert on CIs of type “Application”. An “Application” CI falls in to the class of “child CI”. That is to say in a logical paradigm, an application resides ON a host machine, very much like a Disk Drive, thus we think of it as a “child”.

From a USM standpoint, we will set the property “DeviceDnsName” on the Application CI instead of “PrimaryDnsName” which we would set on a ComputerSystem. DeviceDnsName says “this is the name of the device that I (the Application) reside on. DeviceDnsName functions as a correlatable property, but because there can be a many-to-one relationship between Application CIs and ComputerSystem CIs, that is to say many Applications can reside on the same ComputerSystem, we need another mechanism to uniquely identify the Application. This is where the “ProcessDistinguishingID” USM property comes in. This is essentially a property to give the Application CI a unique name amongst all applications running on the same ComputerSystem host. This is also a required field when creating an Application CI.

As you saw in the first example above we can invent a shorthand name for this, which we called “appname” and then map it to “ProcessDistinguishingID” later using the <ColumnMap> tag. We will use this construct in this example.

Beyond this, to make Application CIs work, we also need a slight tweak to the Connector Policy as well. So we make the following changes to support this use case:

```
<EventClass name="GenericItem" extends="Item" >
	<Classify>
		<Field conditional="user_class" input="user_class"
pattern="^.+$" output="eventtype" outval="CAcc_HasClass" />
		<Field input="hostname" pattern=".*" output="eventtype"
outval="CAcc_ComputerSystem" />
	</Classify>
```

The first change is only the addition of the line in yellow. The rest is part of the base policy file. This simply redirects the CI to a different EventClass if the user specifies a ClassName using the user_class attribute. (We will be doing this)

```
<EventClass name="CAcc_HasClass" extends="GenericItem" >
	<Classify>
		<Field input="user_class" pattern="ComputerSystem"
output="eventtype" outval="CAcc_ComputerSystem" />
		<Field input="user_class" pattern="Application"
output="eventtype" outval="Application" />
		<Field input="user_class" pattern=".*" output="eventtype"
outval="CAcc_ComputerSystem" />
	</Classify>
```

Again we only add the line in yellow. This simply checks if we set user_class and re-classifies the CI to the Application EventClass in this case (Don't confuse EventClass with ClassName. EventClass is only used within the policy file processing, although there is often a one-to-one relationship)

```
<EventClass name="Application" extends="GenericItem" >
	<Format>
		<Field input="hostname" output="DeviceDnsName" format="{0}" />
	</Format>
	<Format2>
		<Field input="MdrElementID" output="InstanceName" format="{0}"
/>
		<Field conditional="ProcessDistinguishingID"
input="ProcessDistinguishingID" output="Label" format="{0}" />
	</Format2>
</EventClass>
```

In this case we add this entire Event Class. A few things are happening here

1) Our hostname (required) is now being mapped to DeviceDnsName for Application
2) The MdrElementID is being pushed to an internal USM property called InstanceName. (This is overriding some unfortunate OOB behavior in the core SOI policy engine)
3) If we set ProcessDistinguishingID (which we will via our appname mapping) it will be used as the Label of this CI

NOTE: These policy file changes to support Application CI class will likely be added to the base policy in an upcoming release, but since they are not there yet, we wanted to publish this example as this is a fairly important use case

Finally, now that we have everything configured properly, here's the UI to send an alert to (and create) our application CI

http://soi:7777/postAlert?hostname=MyServer?user_class=Application&appname=MyApp&Severity=Critical&Summary=This%20is%20an%20application%20alert

Troubleshooting

The following are some key things to know for troubleshooting ConnectorAccelerator

1) The main Connector Accelerator log file is located at <SOI>/log/ConnectorAccelerator.log

2) It uses Apache log4j as all SOI connectors do. This log file is controlled by the configuration file located at <SOI>/resources/Configurations/log4j/ConnAccel_log4j.xml

    a. The main thing you may want to do for troubleshooting is to change the log file from **INFO** to **DEBUG** mode. To do this, navigate near the bottom of this file and look for the entry **<logger name="com.ca.field"…** On the next line set **<level value="DEBUG" />**

3) If you're not seeing anything helpful in ConnectorAccelerator.log or you cannot even get the engine to start, look at **<SOI>/jsw/log/SAM-IntegrationServices.log**, and look for entries for the ConnectorAccelerator (CA:77001 is the product code for the connector)

4) If you're having problems with the transformation, you can enable SOI "debugData" by modifying **<SOI>/resources/log4j.xml**. In this file you will see a section like the following:

5) In the default configuration you will see a snippet like the section below. Uncomment the logger entries in the **RAW ENTITY DEBUG FILES** and **PUB ENTITY DEBUG FILES** sections

```
*********************************************************************
* IFW logger definition
*********************************************************************

<!--  UNCOMMENT AND SET LEVEL TO DEBUG TO SEE DEBUG DATA FILES AND TRACING
FOR IFW -->

<!--      FOR RAW ENTITY DEBUG FILES -->
<!--
    <logger name="com.ca.sam.ifw.framework" additivity="false">
       <level value="DEBUG" />
             <appender-ref ref="IFW" />
    </logger>
-->

<!--    FOR IFW EMBEDDED EI ENTITY DEBUG FILES -->
<!--
    <logger name="com.ca.sam.ifw.eventplus" additivity="false">
        <level value="DEBUG" />
             <appender-ref ref="IFW" />
    </logger>
-->

<!--      FOR PUB ENTITY DEBUG FILES -->
<!--
    <logger name="com.ca.sam.ifw.publisher" additivity="false">
        <level value="DEBUG" />
             <appender-ref ref="IFW" />
    </logger>
-->
```

6) If you're having problems with the SOIPost client, turn on the –debug flag when calling this client and you will get the full HTML response

7) If CIs are not showing up, look for them in the <SOI>/tomcat/logs/ci-invalid.log file on the SOI Manager Server. You can search for the "alertGUID" value that is passed back to SOIPost and appears in the HTML response from the REST API.