Deploying and Managing Apama Applications

5.3.0

May 2015

This document applies to Apama 5.3.0 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: PAM-Deploying_and_Managing_Apama_Applications-5.3.0-20150518@254863

APAMA

Table of Contents

Preface

Deploying and Managing Apama Applications discusses the following topics:

- Issues involved in deploying the components that make up an Apama application
- Starting and managing Apama correlators and adapters with the Enterprise Management and Monitoring console (EMM)
- Tuning Apama applications for optimum performance
- Managing Apama applications over REST
- Using correlator utilities and the extended component configuration file

Documentation roadmap

On Windows platforms, the specific set of documentation provided with Apama depends on whether you choose the Developer, Server, or User installation option. On UNIX platforms, only the Server option is available.

Apama provides documentation in three formats:

- HTML viewable in a Web browser
- PDF
- Eclipse Help (if you select the Apama Developer installation option)

On Windows, to access the documentation, select Start > All Programs > Software AG > Apama 5.3 > Apama Documentation . On UNIX, display the `index.html` file, which is in the `doc` directory of your Apama installation directory.

The following table describes the PDF documents that are available when you install the Apama Developer option. A subset of these documents is provided with the Server and User options.

Title	Description
Release Notes	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.

Title	Description
Using the Apama Studio Development Environment	Instructions for using Apama Studio to create and test Apama projects, develop EPL programs, define Apama queries, develop JMon programs, and store, retrieve and play back data.
Developing Apama Applications	Describes the different technologies for developing applications: EPL monitors, Apama queries, Event Modeler, and Java. You can use one or several of these technologies to implement a single Apama application. In addition, there are C++, C, and Java APIs for developing components that plug-in to a correlator. You can use these components from EPL.
Connecting Apama Applications to External Components	Describes how to connect Apama applications to any event data source, database, messaging infrastructure, or application. The general alternatives for doing this are as follows: • Implement standard Apama Integration Adapter Framework (IAF) adapters. • Create applications that use correlator-integrated messaging for JMS or Software AG's Universal Messaging • Develop adapters with Apama APIs for Java and C++. • Develop client applications with Apama APIs for Java, .NET, and C++.
Building and Using Dashboards	Describes how to build and use an Apama dashboard, which provides the ability to view and interact with scenarios and DataViews. An Apama project typically uses one or more dashboards, which are created in the Dashboard Builder. The Dashboard Viewer provides the ability to use dashboards created in Dashboard Builder. Dashboards can also be deployed as simple Web pages, applets, or WebStart applications. Deployed dashboards connect to one or more correlators by means of a Dashboard Data Server or Display Server.
Deploying and Managing Apama Applications	Describes how to use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. Also provides information for: • Improving Apama application performance by using multiple correlators and saving and reusing a snapshot of a correlator's state. • Managing and monitoring over REST (REpresentational State Transfer). • Using correlator utilities.
Using the Dashboard Viewer	In a User installation of Apama, this document describes how to view and interact with dashboards that are receiving run-time

Title	Description
	data from the correlator. In the Developer and Server installations, this information is included in *Building and Using Dashboards*.

Preface

Contacting customer support

You may open Apama Support Incidents online via the eService section of Empower at http://empower.softwareag.com. If you are new to Empower, send an email to `empower@softwareag.com` with your name, company, and company email address to request an account.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Directory at https://empower.softwareag.com/public_directory.asp and give us a call.

Preface

Chapter 1: Overview of Deploying Apama Applications

Deploying and managing Apama applications involves the following activities:

- Starting and managing Apama components with the Enterprise Management and Monitoring console (EMM)
- Starting and managing correlators with Apama command line utilities
- Tuning Apama applications for optimum performance

About deploying components with EMM

Apama's Enterprise Management and Monitoring (EMM) console provides a graphical interface for configuring, deploying, and monitoring various Apama components across multiple hosts.

- For a general overview of EMM, see "Using the Management and Monitoring Console" on page 12.
- For information on using EMM to start and manage correlators, see "Deploying and Configuring Correlators" on page 41.
- For information on using EMM to start and manage adapters, see "Deploying and Configuring Adapters" on page 60.

Overview of Deploying Apama Applications

About Apama command line utilities

Apama provides a variety of command line tools for managing and monitoring Apama correlators. For information and instructions on using these tools to monitor and manage event correlators, see "Correlator Utilities Reference" on page 103.

Overview of Deploying Apama Applications

About deploying dashboards

Dashboard deployment and administration involves the following activities:

- Deployment package installation and configuration — see "Deploying Dashboards" in *Building and Using Dashboards*.

- Data Server and Display Server management — see "Managing the Dashboard Data Server and Display Server" in *Building and Using Dashboards*.
- Security administration — see "Administering Dashboard Security" in *Building and Using Dashboards*.

Before you perform these tasks, you should familiarize yourself with the deployment and administration concepts described in "Dashboard Deployment Concepts" in *Building and Using Dashboards*.

Deployment options

Dashboards can be deployed as simple, thin-client Web pages, as Java applets, as Java Web Start applications, or as files that can be loaded into a locally-installed, desktop application, the Dashboard Viewer. These deployment options are described and compared in "Deployment options" in *Building and Using Dashboards*.

Data server and display server

Scalability and security of dashboard deployment are supported by the use of the *Data Server* and *Display Server*, which mediate dashboard access to running scenarios and DataViews. The Data Server and Display Server are introduced in "Data Server and Display Server" in *Building and Using Dashboards*.

Process architecture

Applet and Web Start dashboards communicate with the Data Server via servlets running on an application server. Simple, thin-client, Web-page dashboards communicate with the Display Server via servlets running on your application server. Locally-deployed dashboards communicate directly with the Data Server. The structure of deployed configurations is detailed in "Process architecture" in *Building and Using Dashboards*.

Builders and administrators

Dashboard deployment involves the use of a dashboard deployment package that was generated by Apama Studio's Dashboard Deployment Configuration Editor. In some cases the user that generated the deployment package is different from the user that installs and configures the deployment and administers the Data Server. The information that must be transferred between these two types of users is discussed in "Builders and administrators" in *Building and Using Dashboards*.

Overview of Deploying Apama Applications

About tuning applications for performance

The performance of Apama applications can be enhanced by employing multiple correlators. For information about strategies for deploying multiple correlators and the Apama tools available for accomplishing this, see "Tuning Correlator Performance" on page 79. The section also contains information about preserving a correlator's runtime state.

Overview of Deploying Apama Applications

Chapter 2: Using the Management and Monitoring Console

This section describes how to use Apama's Enterprise Management and Monitoring (EMM) console to configure, deploy, and monitor Apama components.

EMM allows centralized, graphical management and monitoring of an Apama installation. From it, you can configure, start, stop and monitor Apama software components (correlators and IAF integration adapters) across multiple hosts. You can also centrally install and upgrade licenses.

Installation

EMM can be run from any computer that has network access to the hosts where Apama components are installed. That is, it does not have to be installed on the machine on which the other Apama components are to be run, although doing so is not detrimental as long as the host is powerful enough to cope with the performance requirements of the application being run on Apama.

Note: In this release, EMM is supported on only Windows platforms. However, it can manage components running on all supported hosts. For a complete list of supported platforms go to Software AG's Knowledge Center in Empower at https://empower.softwareag.com.

Once installed on the host of choice, you can run EMM by selecting the Program Files > SoftwareAG > Apama 5.3 > Administration > Management and Monitoring menu item from the Windows Start menu.

Alternatively, you can run it from the command line as follows:

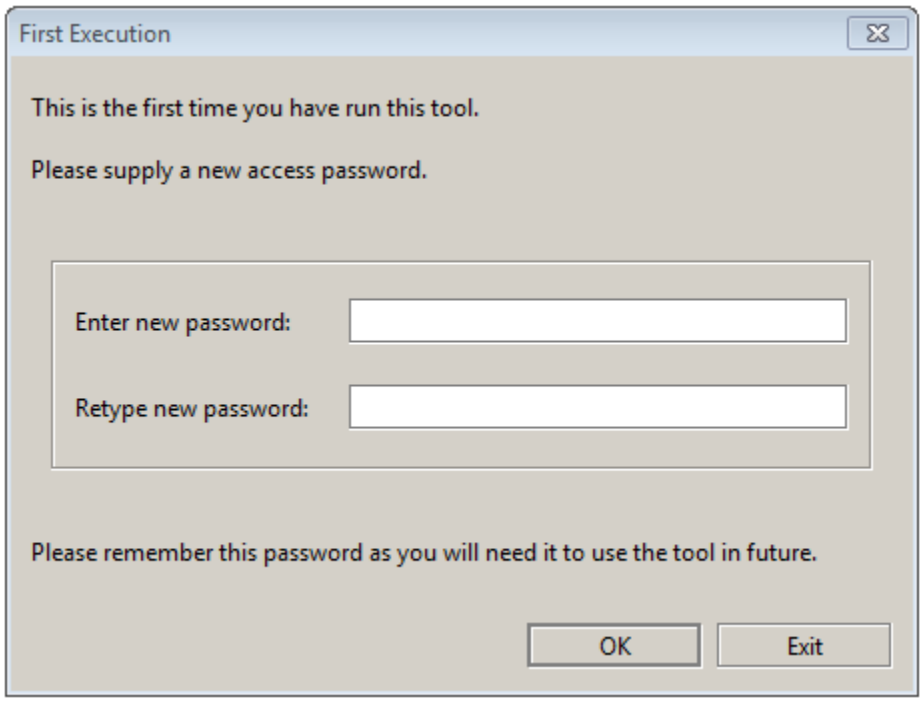
```
emm
```

from an Apama Command Prompt window

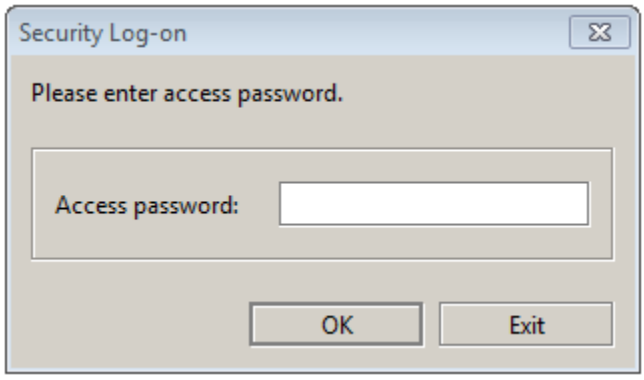
When EMM is run for the first time you will be asked to configure an access password, as shown in the illustration, below.

- If you enter a password here, the password will be required whenever EMM is launched. It can be changed subsequently through the Options menu.
- If you do not enter a password here (i.e., leave the field blank), EMM will not prompt for a password when it starts up. This can be useful if you want to start EMM from a script.

Note: It is important that you keep a record of the access password, as you will be asked to enter it on all subsequent uses of the tool. If you forget the password you will have to reinstall the tool from the original installation distribution.



If you are not running the tool for the first time, you will be asked to enter the access password:



Once you have entered the correct password, you then have access to all the capabilities of the tool.

Using the Management and Monitoring Console

Concepts

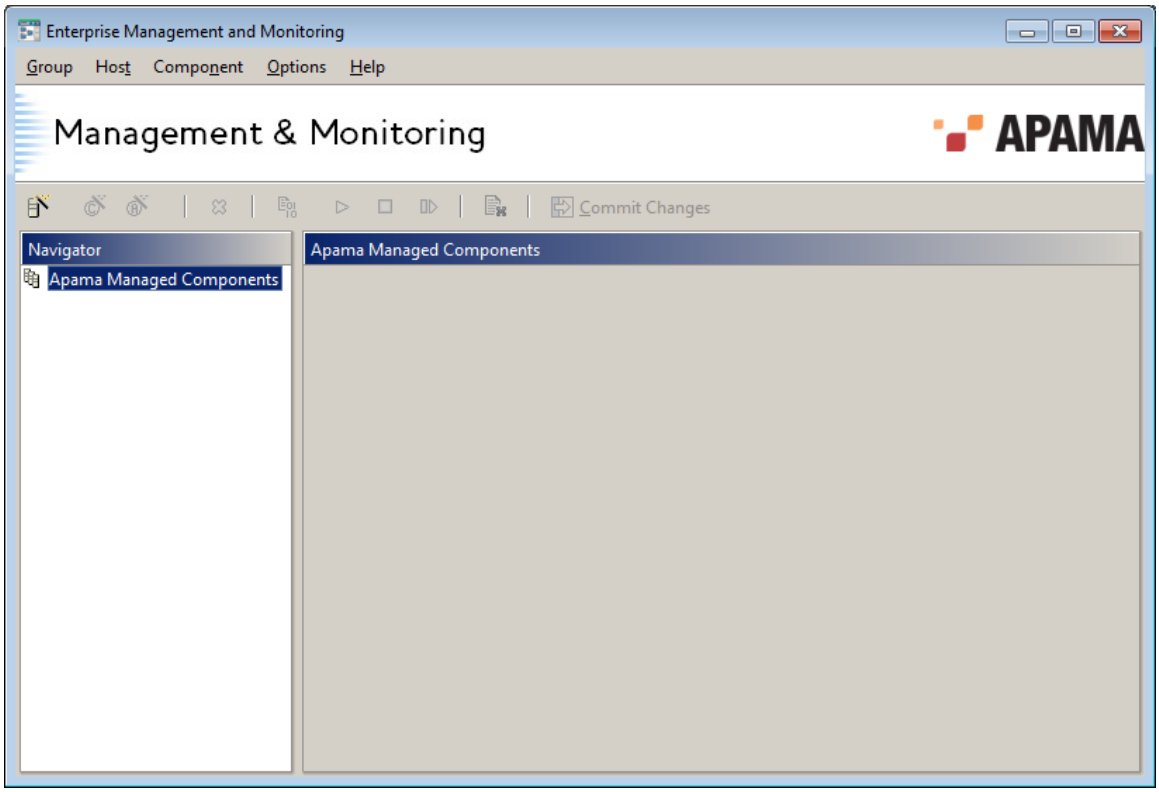
Two basic entities can be configured and manipulated in EMM: *hosts* and *components*. *Hosts* refer to personal computers, workstations or servers, running Solaris, Linux or Windows, which have Apama installations. In addition these machines must have had the *Sentinel Agent* (see "Sentinel Agents" on page 19) installed on them and it must be running. "Managing hosts" on page 18 describes the facilities provided by EMM for dealing with hosts.

Components refer to Apama event correlators and IAF adapters. Apama components can be run and manipulated by EMM on any host that is running the Sentinel Agent. "Managing components" on page 28 describes the facilities provided by EMM for dealing with components.

Using the Management and Monitoring Console

The EMM console window

The following illustration shows the key features of EMM's main window.



Menubar

The Menubar is the main way of interacting with the tool and carrying out operations on hosts and components.

Toolbar

The Toolbar contains a set of icons that provide a convenient graphical shortcut to the most common operations that can be carried out on hosts and components. These operations can also all be carried out from the Menubar.

Navigation Pane

The Navigation Pane lists the set of hosts that EMM is aware of and the Apama components that are running on them. It also provides visual cues as to the current operating status of the hosts and components. Right clicking on an entity displays a context-sensitive menu that lists the operations that can be carried out on that entity. These operations can also all be carried out from the Menubar and from the Toolbar.

Details Pane

The Details Pane displays information that is relevant to the currently selected host or component. Its contents change accordingly, and can be used to both configure as well as to monitor and inspect the status of the selected item.

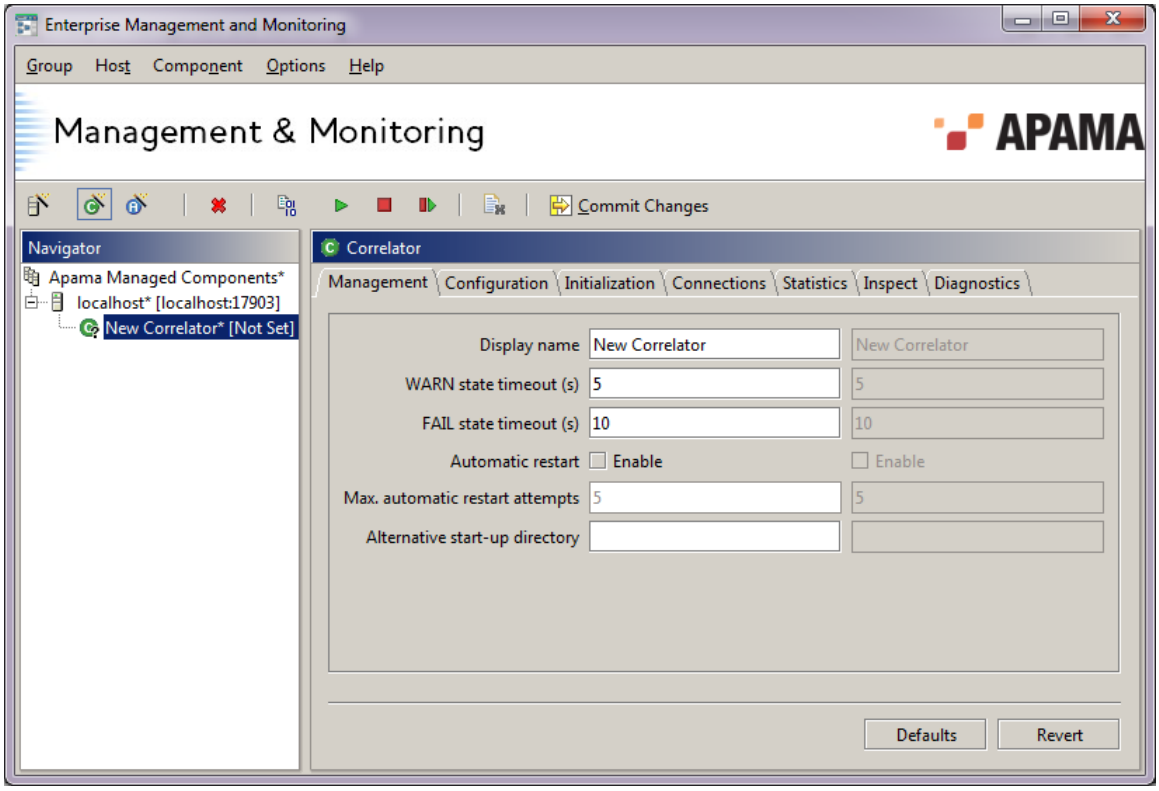
Concepts

Committed values and outstanding changes

EMM can configure and manipulate hosts and Apama components, and to do this it needs to maintain extensive configuration information.

Throughout EMM there are concepts of the *current, committed* configuration data and of *outstanding changes*, or *working copy* data.

On each Details Pane for a host or a specific component where you can supply configuration data, you will see two columns of values, as shown in the following illustration. The rightmost, grayed out values indicate the actual *committed* data – the values currently in use. The leftmost column is where you can provide the new values you would like to use. As long as they are just shown in the leftmost columns, these *outstanding changes* are just that – temporary working data. A component or host with unsaved changes will have a `"*"` suffix in its entry in the Navigation Pane. For changes to take effect you need to commit them, using the Commit Changes button () on the Menubar, or the relevant menu items on the Group, Host and Component menus. Note that some component options cannot be changed once the component has started, and these will not take effect until the component is restarted, even after being committed.



The Commit Changes action scope varies according to what is selected when it is pressed:

- if a component is selected in the Navigation Pane, then all outstanding changes in that component, and only in that component, are committed (the Commit changes item on the Component menu is enabled in this case),
- if a host is selected in the Navigation Pane, then all outstanding changes in that host, and in all its components, are committed (the Commit changes item on the Host menu is enabled in this case),
- if Apama Managed Components is selected in the Navigation Pane, than all outstanding changes throughout all of EMM's hosts and components are committed (the Commit changes item on the Group menu is enabled in this case).

See "Managing hosts" on page 18 and "Working with hosts" on page 21 for information on the Details Pane for hosts and for the specific components supported by the EMM console.

Concepts

State persistence

All of EMM's management state, including the list of hosts and components it is managing and the committed settings for each (including the diagnostic messages), are persisted by EMM to disk.

When EMM is stopped and restarted no settings are lost, although note that in the current version EMM's status monitoring and auto-restart capabilities are only active while the console is running.

The state is stored in a file in the Apama Work directory called `etc\emm_state.dat` and a backup is created in `etc\emm_state.dat.bak`. EMM must have the necessary permissions to read/write these files. If there is a problem locating or reading the state file EMM will offer to load from the backup file instead. If neither file can be located or read, EMM will revert back to its default installation stage

configuration. Should EMM fail to save its state file correctly for whatever reason then temporary files of the form `serialXXXXX.tmp` will be created in the `etc` directory; these files can be safely deleted.

Using the Management and Monitoring Console

Managing licenses

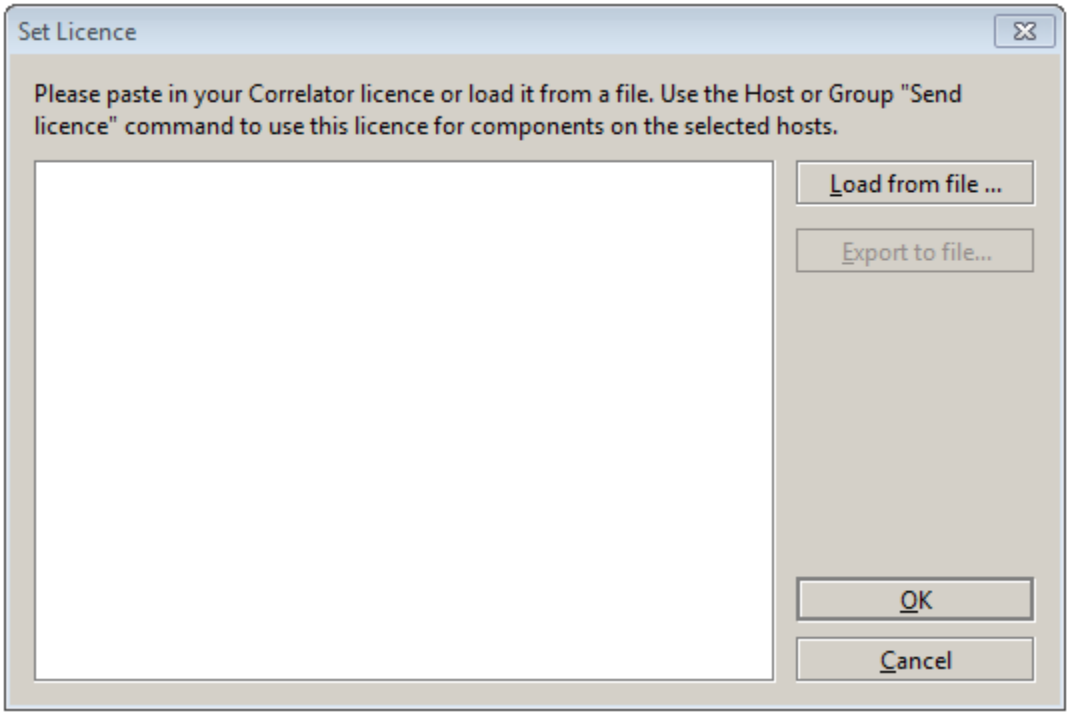
An Apama event correlator requires a valid Apama license. As well as allowing the correlator to start, a license specifies restrictions in its functionality and performance. A file containing a valid license must be available to each correlator all the time.

EMM aids license deployment and maintenance by its facility to push out licenses to any host where Apama correlators are to be started (or are already running). This facility also allows updating of existing installed licenses.

To load a license for use with EMM:

1. Select Set License… from the Options menu option on the Menubar.

   This displays the Set License dialog box shown in the following illustration.



   In theory you can type in the license but as it is quite a long string and must be entered exactly as it was sent to you by Apama Technical Support, it is greatly preferable to read it in from a text file.

2. Select Load from file …, locate the license file from where you saved it, and Open it.

3. Click OK to set this as the current active license to be used by EMM.

However, note that this license is only actually sent out to hosts when the Send license option is applied to the host entity. Alternatively, the licenses on all known hosts can be updated at the same time by selecting Apama Managed Components in the Navigation Pane and choosing the Send License option from the Group menu.

Note: EMM will not attempt to validate your license; therefore it is important that you enter it correctly. If you attempt to start a correlator and the host has no license file then the correlator will fail to start unless it is running on the local host. If the correlator is on the host named “localhost” or “127.0.0.1” then it will start, but in a restricted mode that prevents connections to other hosts and the correlator will automatically terminate after 30 minutes of operation; it will be flagged with a warning icon to indicate that it has started in this state (look at the component’s Diagnostics tab for more information). If the license file exists but is expired or invalid then the correlator will fail to start.

Using the Management and Monitoring Console

Managing security

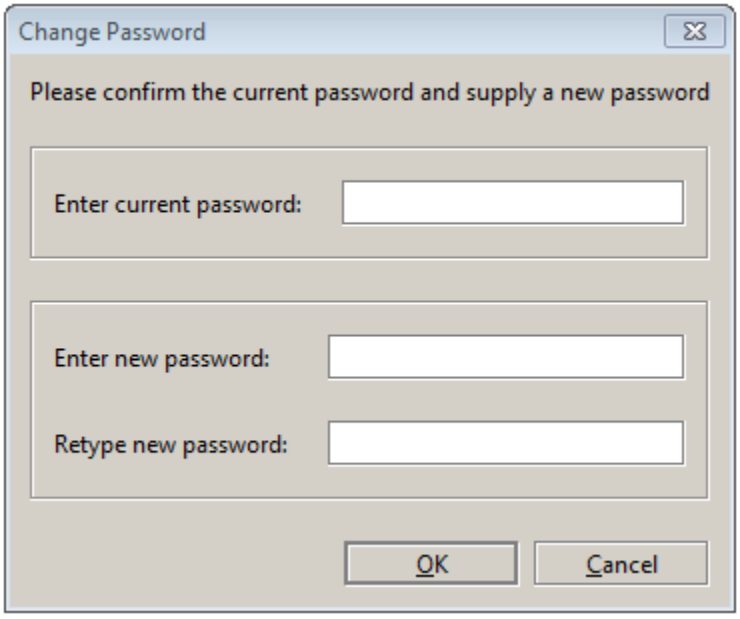
When you run EMM, if you have configured a password as described in "Installation" on page 12, the start up process will always request an access password. If you configured a password when you first ran EMM, this step cannot be bypassed. Should you forget the access password, you will need to reinstall the tool from the Apama installation media.

If you left the password field empty as described in "Installation" on page 12, you are not prompted to enter a password.

To change the password:

1. Select Change Password… from the Options menu option on the Menubar.

   This displays the **Change Password** dialog, shown in the following illustration.



Using the Management and Monitoring Console

Managing hosts

EMM can start and stop components on any Windows XP/2003, Solaris 10, RedHat Enterprise Linux 4.0, and SUSE Linux Enterprise Server 10.0 host within your network, as long as that host has had the relevant Apama components installed on it, and as long as it is running a Sentinel Agent.

Using the Management and Monitoring Console

Sentinel Agents

A Sentinel Agent is a small process that can start and stop Apama correlators and IAF adapters on the host where it is running. EMM starts and stops components on a particular host in a platform independent way by interacting with the Sentinel Agent running on that host. Therefore a Sentinel Agent must be running on any machine where you intend to manipulate Apama components using EMM.

On Windows, the Sentinel Agent can be started in one of these ways:

- If you checked the sentinel agent check box during the Apama installation process, sentinel agent is automatically installed with `startup type =automatic`.
- By starting the Windows Service with the `apama_services` command `apama_services -start -s sentinel`. The service can be stopped with `apama_services -stop -s sentinel`. If the service was not added during installation it can be added with `apama_services -add -s sentinel`.
- By starting the Sentinel Agent executable manually but not as a Windows Service. This can be done either from an Apama command prompt or command shell where you have run `apama_env.bat` to set the necessary Apama environment variables. Run `sentinel_agent.exe` and pass command line arguments to it.

On UNIX, Sentinel Agent can be started in one of these ways:

- By adding it as a service (daemon) through the installer. In this case the Sentinel Agent will automatically start up when the machine boots.
- By starting the service with the `apama_services` command `apama_services -start -s sentinel`. The service can be stopped with `apama_services -stop -s sentinel`. If the service was not added during installation it can be added with `apama_services -add -s sentinel`.
- By starting the Sentinel Agent either from a script or command shell where you have sourced `apama_env`. Run `sentinel_agent` and pass command line arguments to it.

Note: Any information written to `stdout` or `stderr` by components started with a Sentinel Agent will be logged in the Agent's log directory in files called `<component-type>-<PID>-std{out,err}.log` on UNIX and `<component-type>-<num>-std{out,err}.log` on Windows, where `<PID> = Process Id` and `<num>` is an integer number. Valid component types are `correlator` and `adapter`. The `stdout` / `stderr` logs will be deleted by Sentinel when the relevant component is stopped if they are empty (which should normally be the case).

Managing hosts

Options

The Sentinel Agent's startup and usage information is as follows:

```
Usage: sentinel_agent [ options ]
Where options may include any of:
   -V | --version               Print program version info
   -h | --help                  This message
   -p | --port <port>           Listen on alternate port (default is 17903)
   -f | --logfile <file>        Log to named file (default is stderr)
   -v | --loglevel <level>      Set logging verbosity.  Available levels
                                are TRACE, DEBUG, INFO, WARN, ERROR CRIT FATAL
                                (default logging level is WARN)
   -t | --truncate              Truncate log file at startup
   -N | --name <name>           Set the component name
   -l | --license <file>        Apama license filename
   -H | --apamahome <dir>       Root of Apama installation
   -W | --apamawork <dir>       Root of Apama work directory
 -Xconfig | --configFile <file> Use service configuration file <file>
```

The license is assumed to be in `APAMA_WORK\license\license.txt` unless overridden on the command line. The Apama work directory is set from the `-W` argument or from the `APAMA_WORK` environment variable if `-W` is not specified.

These options can be specified to the Sentinel Agent executable whether it is run as a normal Windows or UNIX executable.

Options

Table 1. Sentinel Agent options

Option	Description
`-h`	Display the above usage information. Does not start the Sentinel Agent.
`-p`	Name of the host on which the Sentinel Agent will listen for connections from EMM (default is `17903`).
`-l`	A reference to an Apama license file. If provided, the Sentinel Agent will be able to pass this license file's location on to the Apama components that it starts. Alternatively a license file can be sent to the Sentinel Agent dynamically via EMM.
`-H`	The root directory where the Apama executable files are installed, normally `C:\Program Files\Software AG\Apama` *`rel_num`*`\bin` on Windows or `/opt/SoftwareAG/apama_`*`rel_num`*`/bin` on UNIX.
`-W`	The root directory where the Apama work files are located, normally `C:\Users\`*`username`*`\Software AG\ApamaWork_`*`version`* on Windows or `$HOME/apama-work` on UNIX.
`-f`	The file where to output logging information from the Sentinel Agent. If a log file is not specified and the Sentinel Agent was run as a normal executable from a Command Prompt or a Terminal window, it will log to the system configured `stderr`. If the Sentinel Agent was run as a Windows service, a log file must be specified; otherwise the Agent will not start.
`-N` *`name`*	Sets the name of the component to *name*.

Option	Description
`-t`	If set, the log file will be emptied at start up, otherwise it will be appended to.
`-v`	Set the logging level. The log levels possible, in increasing order, are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `CRIT`, and `FATAL`, where each level includes all subsequent ones. For example, `WARN` includes all `WARN` level, all `ERROR` level, all `CRIT` level,and all `FATAL` level log messages. If not specified the default log level is `WARN`.
`-V`	Print out the Sentinel Agent's version information. Does not start the Sentinel Agent.
`-Xconfig` *`file`*	Use the specified service configuration file. For more information about this configuration file, see "Using the Apama component extended configuration file" on page 169.

Managing hosts

Working directory

The Sentinel Agent's current working directory/folder becomes the current working directory of all components started by the Agent, unless an `Alternative start-up directory` was configured for the component. The working directory is important because most of the files and paths specified when configuring components in EMM are assumed to be relative to the Sentinel Agent's working directory if a full absolute path is not provided.

Determining the working directory

If the Sentinel Agent was started manually then its current working directory is simply the current directory when it was run.

If it was started automatically – either as a service on Windows or as a daemon on UNIX – then the current working directory is determined as follows.

On Windows:

1. Normally, the Sentinel Agent tries to use the `logs` subdirectory of the Apama work directory.
2. Otherwise the Sentinel Agent tries to use the `%WinDir%\System32` directory.
3. If the Sentinel Agent cannot write to any of these directories, it will fail to start.

On UNIX:

1. The Sentinel Agent tries to use the `/logs` subdirectory of the Apama work directory.
2. If the Sentinel Agent cannot write to this directory, it will fail to start.

Managing hosts

Working with hosts

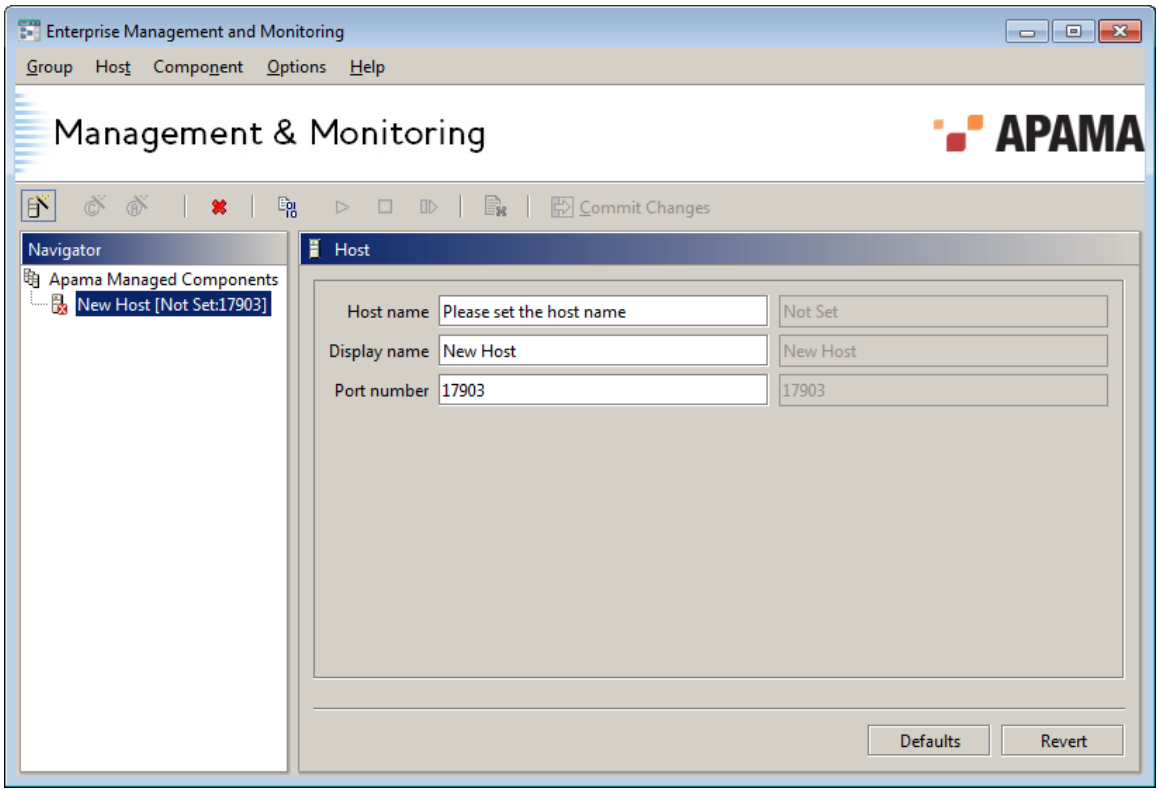
This documentation refers to the items on the EMM menu as the primary means of carrying out operations on hosts. However, these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

The Host menu provides the following options for interacting with hosts:

Add host

This is equivalent to using the  button on the toolbar.

Add host allows you to add a host to the `Apama Managed Components` group, the root node illustrated in the Navigation Pane. This adds a placeholder node in the Navigation Pane and provides options for configuring EMM to work with the host in the Details Pane, as illustrated in the following.



You need to provide the following information:

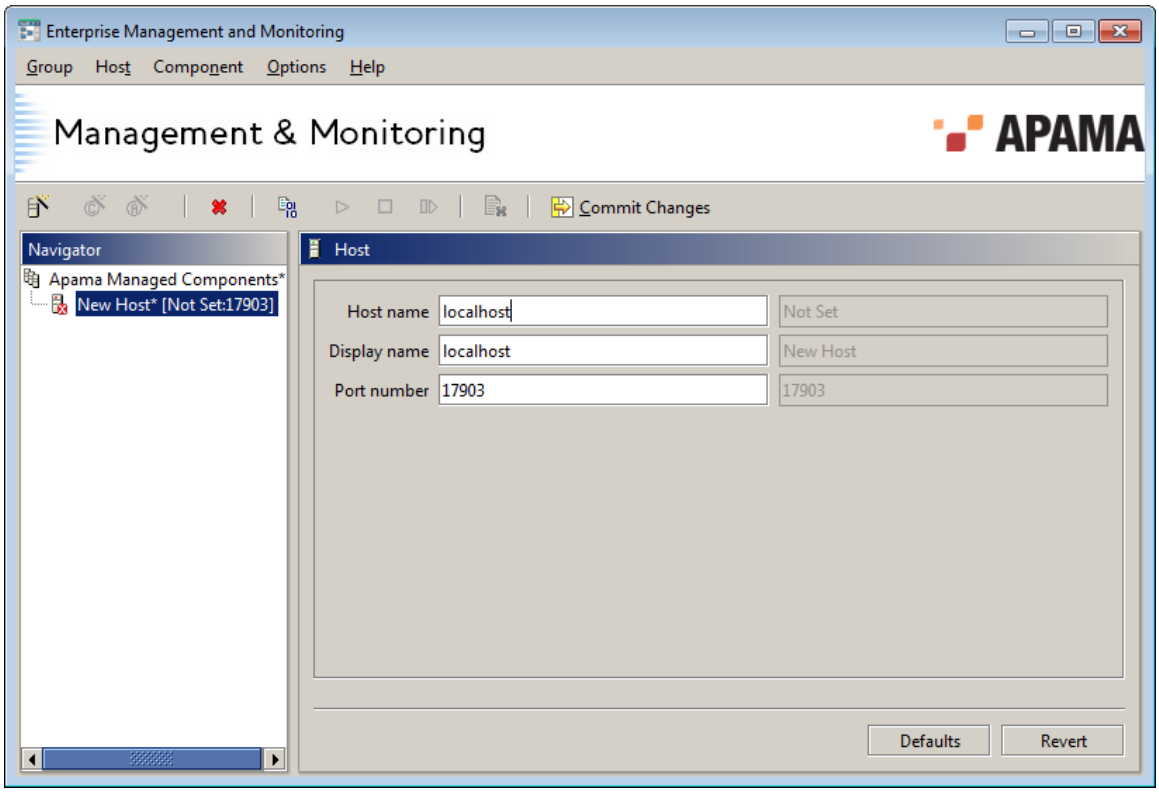
- Display name – A descriptive name for the host. This name is what is displayed in the Navigation Pane.
- Host name – The host's name or IP address, for example `lawrence.apama.com` or `127.0.0.1`. Whether you need to provide the full name as in this example, or whether an abbreviation (e.g. `lawrence`) will do depends on your network configuration and DNS/WINS settings, but in most circumstances it is advisable to use the host's full network name.

You can only use ASCII characters in a host name.

Note that it is not possible to change the Host name once components have been added to the host.

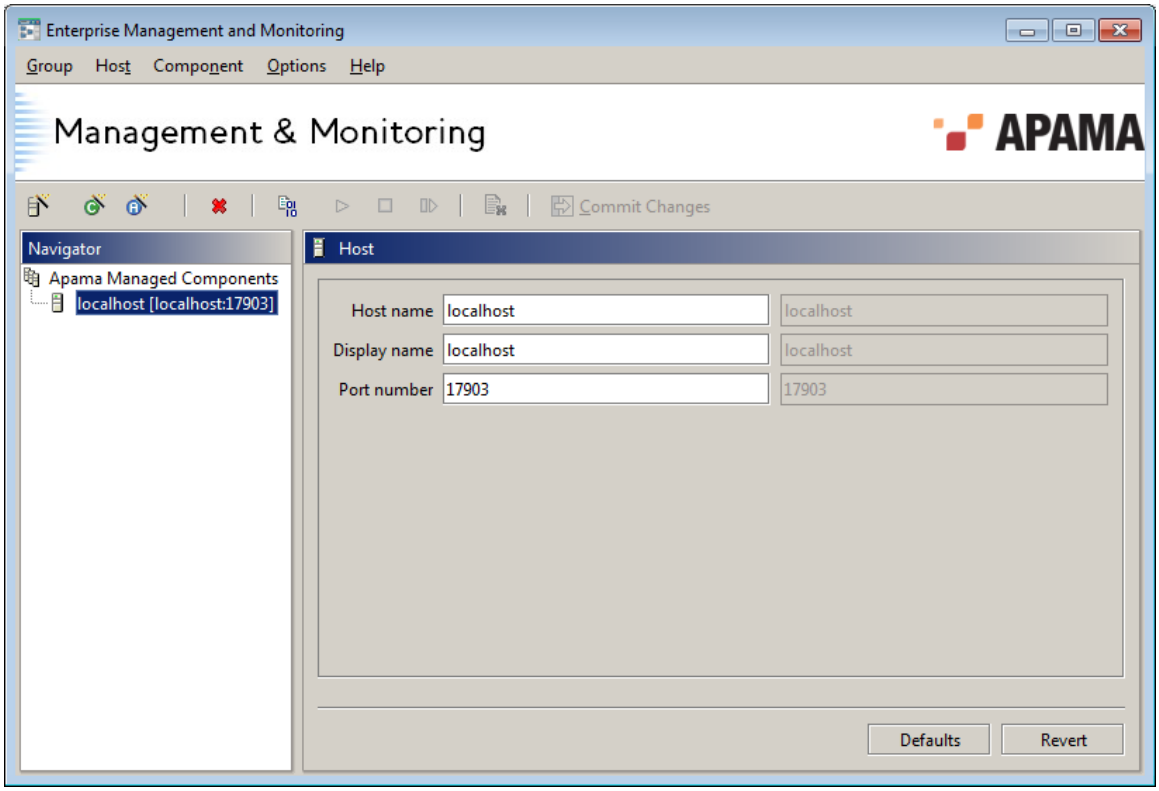
- Port number – The port that the Sentinel Agent (on that host) is listening on. By default this is `17903`, although it might have been changed by whoever installed and configured the Sentinel Agent on that host.

In following illustration, we have created a reference to the host `localhost` and specified port `17903` (the default port) as the port to use for communicating with the Sentinel Agent running on this machine.



Once the host information has been set, click on the Commit Changes button ( ) on the toolbar. Note that no components can be added to a host until it is committed.

The Commit Changes button will be disabled if the Host name has not yet been set or if the host has already been setup and committed, and there are no outstanding changes to commit on the host or any associated component.



Notice how the icons next to the new node in the Navigation Pane are different in the illustrations above. When you first define a new host the icon alongside it in the Navigation Pane will be . This will change to once EMM successfully manages to contact the Sentinel Agent running on that host.

EMM will continue verifying communication with the Sentinel Agent on that host at regular intervals, for as long as that host is included in the Apama Managed Components group.

If EMM cannot communicate with a host's Sentinel Agent, or connects and subsequently loses connection, the icon against that host will change to and stay that way until the problem is resolved. This might be the case if the host is no longer available on the network, the network has failed, the remote Sentinel Agent has been shut down or if a firewall (such as the one installed by default with Microsoft Windows XP Service Pack 2) is blocking the port that the EMM console uses to communicate with the Sentinel Agent.

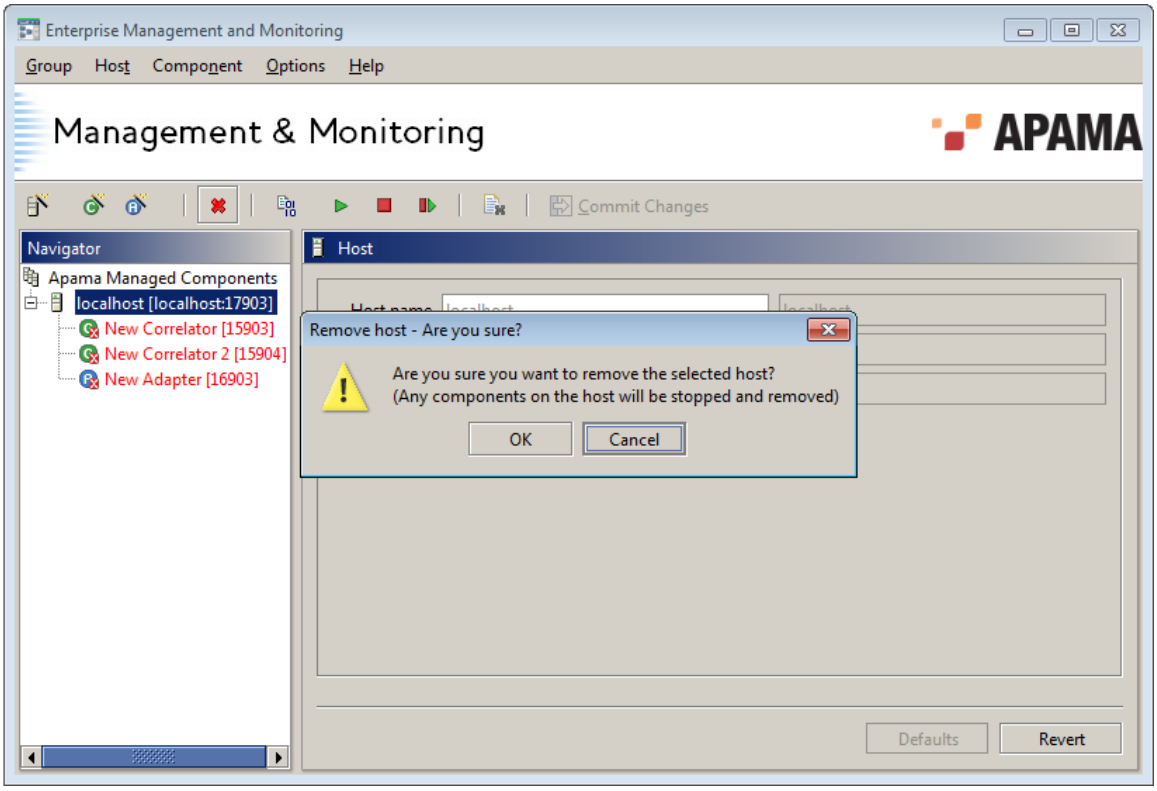
Remove host

This is equivalent to using the button on the toolbar, and is only available if a host is currently selected in the Navigation Pane.

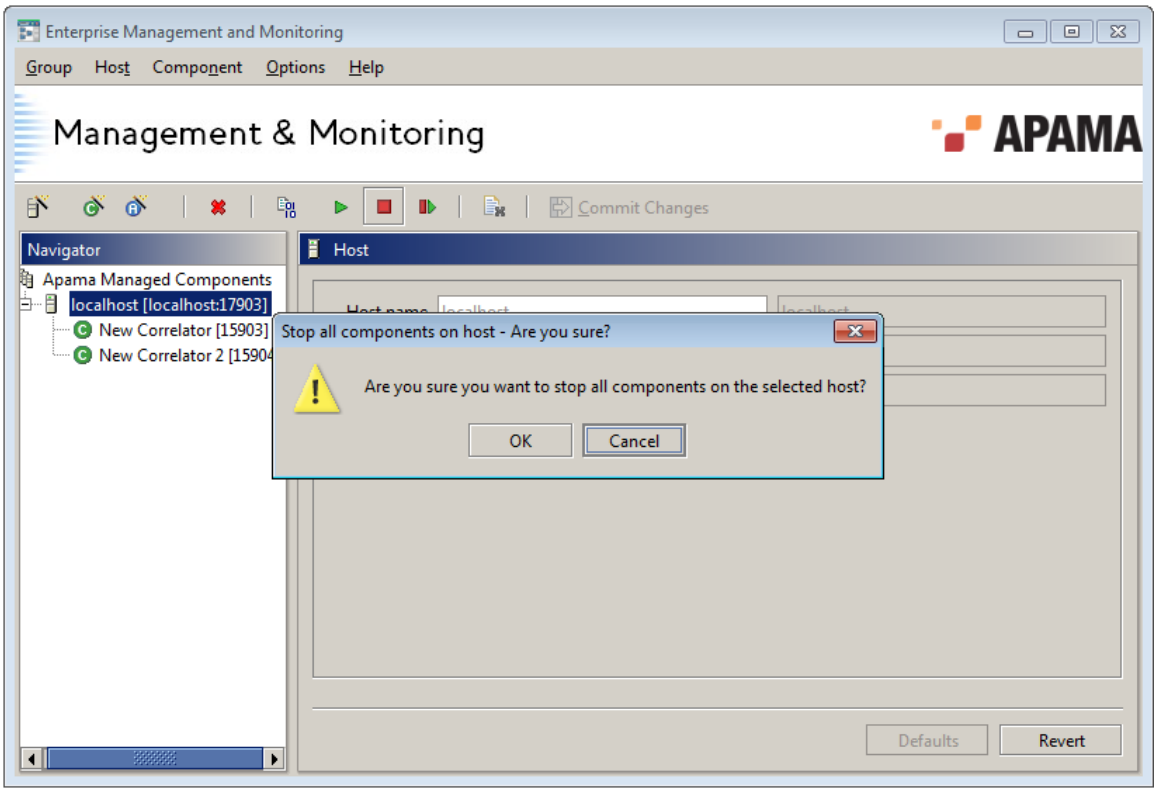
Choose it to remove all components on that host from the Apama Managed Components group.

Note that this operation does not shut down the remote Sentinel Agent on the host in question.

By default, EMM displays a confirmation dialog providing the option to cancel the removal of the host, as shown in the following illustration.



If the host contains one or more components that are known to be running, as well as providing the option to cancel the removal the dialog asks whether these component(s) should be stopped before being removed; see the following illustration.



Note: See "Preferences" on page 34 for information about how these confirmation dialogs can be turned on or off using the EMM Preferences dialog.

Start component(s)

This is equivalent to using the button on the toolbar. This option is available if a host is currently selected in the Navigation Pane.

Choose it to start all Apama components configured on that host. See Chapter 3 for details on how to add and configure Apama components.

See "Start component" on page 31 for information on why components might fail to start.

Stop component(s)

This is equivalent to using the button on the toolbar. This option is available if a host is currently selected in the Navigation Pane.

Choose it to stop all Apama components configured on that host.

Restart component(s)

This is equivalent to using the button on the toolbar. This option is available if a host is currently selected in the Navigation Pane.

Choose it to re-start (stop and start again) all Apama components configured on that host.

Send license

This is equivalent to using the button on the toolbar. This option is available if a host is currently selected in the Navigation Pane and a license has been configured within EMM (through the Options / Set license menu option).

Choose it to push out the currently configured license to the Sentinel Agent running on that host, which the Sentinel Agent will use to create a local license file on that host. The license file will be written to the location that was specified in the Sentinel Agent's command line startup parameters, replacing any existing license file.

A dialog will be displayed indicating if the license push was successful or whether the license was rejected as invalid or some other error occurred.

Components do not have to be restarted to become aware of the new license file.

Managing hosts

Managing all known hosts

All hosts defined within EMM are attached to the Apama Managed Components group item in the Navigation Pane.

The Group menu option allows one to run operations across all hosts in this group; that is all known hosts, without having to select them individually.

The operations available on the Group menu are also available by selecting the Apama Managed Components item in the Navigation Pane and then right-clicking to get its context-sensitive popup menu. While the item is selected you can also use the toolbar buttons to the same effect.

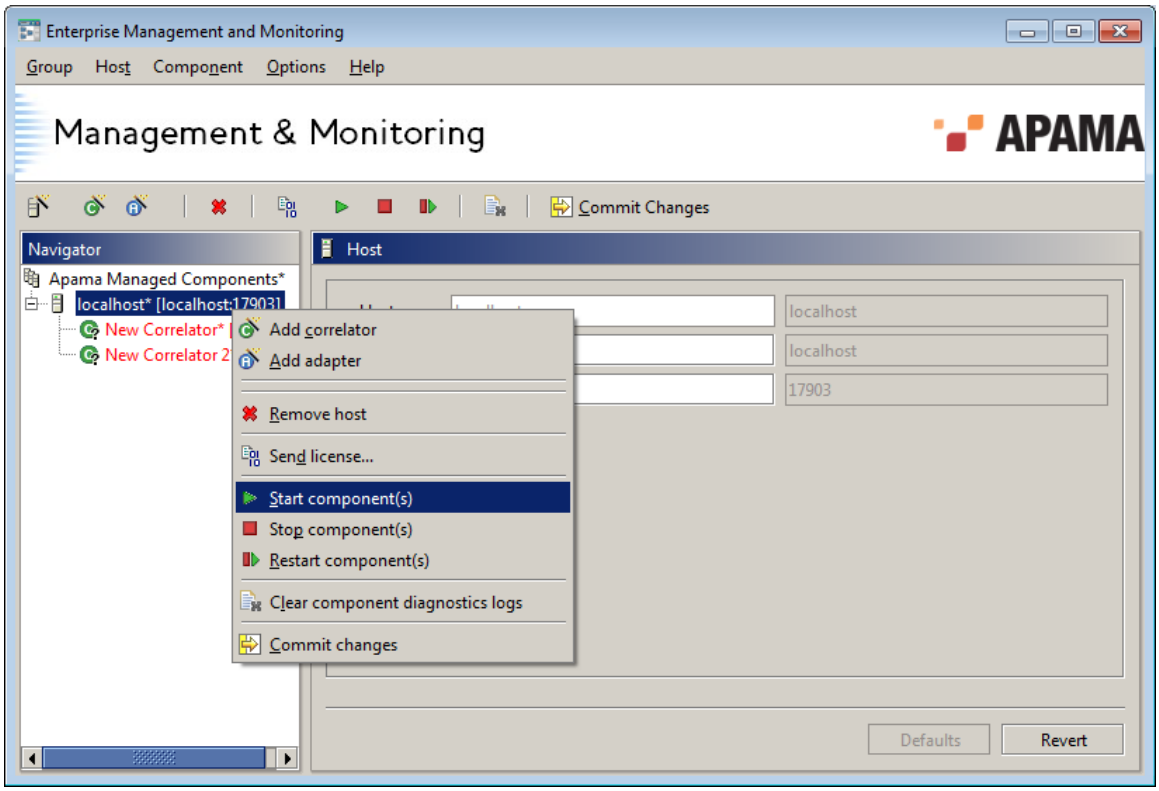
Restart components

Choose this option to restart all components on all hosts. This is equivalent to clicking on the toolbar.

Start components

Choose this option to start all components on all hosts. This is equivalent to clicking on the Menubar.

See "Start component" on page 31 for information on why components might fail to start.



Stop components

Choose this option to stop all components on all hosts. This is equivalent to clicking on the Menubar.

Send license

Update the license on all hosts. This is equivalent to clicking on the Menubar.

See "Send license" on page 26 for more information about license sending.

Managing hosts

Managing components

EMM can support various Apama components that can be used together to create a distributed Apama deployment.

The components are:

- Event Correlators – the event correlator is the core Apama event matching and analytic engine. See "Add correlator" on page 30 and "The correlator tabs" on page 42.
- IAF Adapters – the Integration Adapter Framework (IAF) allows rapid generation of integration adapters. These allow Apama to interface with an external source of events like a message bus, an event feed, or a database. See "Add adapter" on page 30 and "The Adapter tabs" on page 62.

For information on how to use Apama command line utilities to run and configure the event correlator, see "Correlator Utilities Reference" on page 103.

For information about using the adapters provided with Apama or creating adapters of your own, see:

- "Using Standard Adapters" in *Connecting Apama Applications to External Components*
- "Developing Custom Adapters" in *Connecting Apama Applications to External Components*

Using the Management and Monitoring Console

Component status indicators

EMM can configure, start and stop components. For all the components it manages, EMM can monitor execution and automatically restart in case of failure (re-initializing where applicable).

As components are managed, their icons in the Navigation Pane will change to indicate their status. Although the base icons for a correlator and an IAF adapter are different, being  and , they have the same set of symbols overlaid on them to indicate their state. Using the correlator's icons as an example, the indicated states are:

-  - UNKNOWN. The state is still unknown, i.e. EMM is trying to communicate with the component.
-  - STOPPED. The component is not running and it is not supposed to be, that is, it has not been started, or it was explicitly stopped, in EMM.
-  - RUNNING. The component is running normally and it is supposed to be, that is, it was started from EMM.
-  - STOPPING, STARTING. A stop or start operation is in progress.
- - WARN. This can mean one of the following:
  - The component is supposed to be running but is **not responding** – EMM has lost contact with it and is trying to reestablish communication. If this does not succeed the component will progress to the FAIL state.

- The component itself started correctly but **one or more initialization actions failed**. Use the component's Diagnostics tab to investigate where the problem occurred. The warning can be cleared by pressing the Clear warning button on the Diagnostics tab.
- The component has been started but it is **not supposed to be running** (e.g. it was started from outside EMM). The warning can be cleared by pressing the Clear warning button on the Diagnostics tab.
- A correlator started on the local machine but there was no license file, so the component will automatically terminate after 30 minutes. The warning can be cleared by pressing the Clear warning button on the Diagnostics tab, however Apama recommends supplying a valid license and restarting the component.
- The component itself started correctly but one or more **upstream connections to other components could not be established**. Use the component's Diagnostics tab to investigate where the problem occurred. The warning can be cleared by pressing the Clear warning button on the Diagnostics tab.

-  - FAIL. The component has failed. The component stopped responding to EMM for the entire WARN timeout configured for the it plus the FAIL timeout, so EMM now assumes that the component is no longer running.

In case of a FAIL check the Diagnostics tab to discover which of these situations occurred.

Note: The time to wait after a component stops responding before entering the WARN state, and the time between entering the WARN state and progressing to FAIL may both be configured on a per-component basis using the Management tab in the Details Pane.

Using the Management and Monitoring Console

Working with components

This section refers to the items on the EMM menu as the primary means of carrying out operations on Apama components. However, these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

Before operations can be carried out on these components, a Sentinel Agent must be installed and running on the target host, and the EMM console must have been configured to manage that host, as detailed in "Managing hosts" on page 18.

The following options are available from the Components menu when a host is selected:

- "Clear all component logs" on page 33

Using the Management and Monitoring Console

Add correlator

This option is only available if a host (or an existing component on a host) is selected in the Navigation Pane. It is equivalent to pressing the button on the toolbar.

Choose this option to define and configure a new event correlator component on the selected host.

Working with components

Add adapter

This option is only available if a host (or an existing component on a host) is selected in the Navigation Pane. It is equivalent to pressing the button on the toolbar.

Choose this option to define and configure a new IAF Adapter component on the selected host. This will add an Adapter to the host in the Navigation Pane, select it, and display the Management tab in the Details Pane.

Working with components

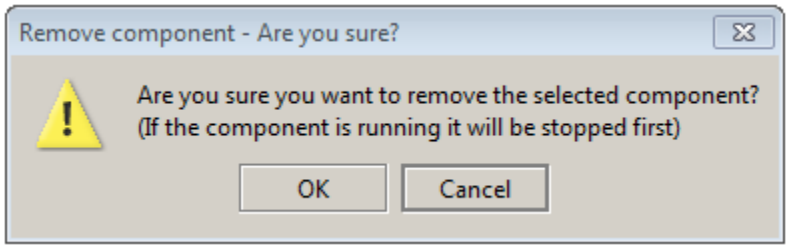
Remove component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the button on the Menubar.

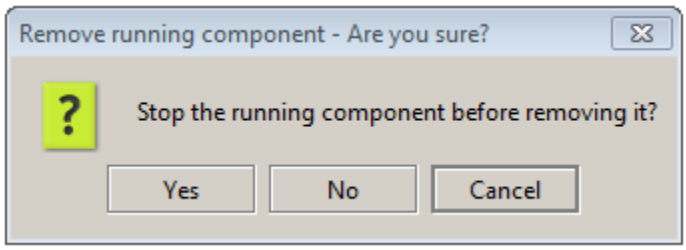
Choose this to remove the currently selected component from the Navigation Pane and stop it from being managed by EMM.

You can also remove a component by selecting it and pressing the Delete key.

By default, EMM displays a confirmation dialog providing the option to cancel the removal of the component, as shown below.



If the component is known to be running, as well as providing the option to cancel the removal the dialog asks whether the component should be stopped before being removed, as shown below.



Note: "Warnings" on page 34 describes how these confirmation dialogs can be turned on or off using the **EMM Preferences** dialog.

Working with components

Move component to host

This option is only available if a component is selected in the Navigation Pane. When a component is selected and you select Move component to host from the Component menu, the **Select New Host for Component** dialog is displayed. Select the new host and click OK.

Any component type may be moved with this command. The moved component will have exactly the same configuration as it did on its previous host, including the same logical ID.

Working with components

Start component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the ▶ button on the toolbar.

Choose this option to start the currently selected component. The component's icon will change to indicate that the component is starting, and then to show it is operational. Note that this can take a few seconds since the icon is only updated once EMM can satisfactorily communicate with the component.

A component can fail to start for several reasons. The most common are:

- A valid license was not available on that host.
- The Sentinel Agent is not running on the host (check that its icon in the Navigator Pane is and not)
- The component's executable was corrupt, or inaccessible due to user permissions issues.
- The component's listening port was not available as it is in use by something else.
- There was insufficient disk space on that host for the Sentinel Agent's or the component's logging.
- The component could not write to the specified log file due to user or file permissions issues.

- The component could not locate its essential runtime libraries. Either the component or the libraries it requires were manually moved, or the environment variables are set incorrectly on that host.
- You might have set important management and configuration parameters but forgotten to commit them.
- The component might appear not to have started even though in fact it has, due to a firewall (such as the one installed by default with Microsoft Windows XP Service Pack 2) blocking the port that the EMM console uses to communicate with the component. The firewall must be disabled or reconfigured to allow traffic on the port the component is configured to listen on.
- The Sentinel Agent on that host could not locate the component's binary. This means it was not configured correctly; normally this is due to it not having had valid component paths specified.

Note: All components started by the Sentinel Agent are started under the same user context (and therefore with the same privileges) as the user the Sentinel Agent itself was started as.

Therefore, if the Sentinel Agent was started as a service as the user `Local System`, all the components it starts will also start as the user `Local System`. Note that `Local System` is not a Windows Domain user, and therefore might not be allowed access to Domain resources. This might be particularly relevant to IAF adapters which link against custom user code and are likely to require access to various network services and resources.

If this situation is encountered, you need to change the user that the Sentinel Agent is started as, which can be done through Control Panel, Administrative Tools, Services, right click on Apama Sentinel Agent, select Properties, then the Log On tab. Choose the This account: radio button and enter the username and password of the user to use instead. You can specify a domain user by preceding the username with the domain as follows: *<domain name>/<username>*. You will of course have to be an Administrator on the machine to be able to do any of this.

Working with components

Stop component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the  button on the toolbar.

Choose this option to stop the currently selected component. The component's icon will change to indicate that the component is no longer running or available. Note that as the icon is only updated once EMM can no longer communicate with the component this can take a few seconds.

Working with components

Restart component

This option is only available if a component is selected in the Navigation Pane. It is equivalent to pressing the  button on the toolbar.

Choose this option to trigger a restart on the currently selected component. The component's icon will change to indicate that the component has stopped and then change again to indicate it has started up and is operational again.

Working with components

Clear all component logs

This option is equivalent to clicking the  button. If Apama Managed Components is selected in the Navigation Pane, this option clears the diagnostic logs for all the managed components. If a host is selected in the Navigation Pane, it clears the diagnostic logs for all the components on that host.

Working with components

Configuring components with the details pane

The contents of the Details Pane change according to what is currently selected in the Navigation Pane. When a component is added to a host, it is automatically selected in the Navigation Pane.

Note: As already described, you need to press Commit Changes () before any changes you make to any panel on the Details Pane take effect. The values that are currently in effect are shown within the rightmost grayed out labels.

Using the Management and Monitoring Console

Specifying paths and filenames in the Details Pane

On several of the component Details Pane panels you are asked to provide a path or filename, (for example to specify where logging information should be stored, or where an adapter configuration file should be loaded from).

It is recommended that *absolute paths* and filenames be used where possible – an absolute path being one that specifies the whole path, for example:

`c:\Document and Settings\My User\apama-work\logs\` (on Windows)

or

`/home/apama-work` (on UNIX)

In contrast, *relative paths* are incomplete, or just consist of a filename on its own, such as:

```
New_Correlator_1.log
```

Important notes

- Unless otherwise stated, all paths are located on the file system of the remote host, on which the Sentinel Agent and managed components are running, rather than the local host where EMM is running.
- Filenames and paths should never be enclosed in quotes, even if they contain spaces.
- If a relative path is provided – as is the case by default for component log files – it is assumed to be relative to the component's `Alternative start-up directory` setting if one was configured, or to the **current working directory** of the Sentinel Agent on that host if not. See "Working directory" on page 21 for details of how the Sentinel Agent's working directory is determined.

Configuring components with the details pane

Preferences

Several of EMM's default operational characteristics such as the warnings it issues, some of its display properties, and how it updates component information can be configured by the user.

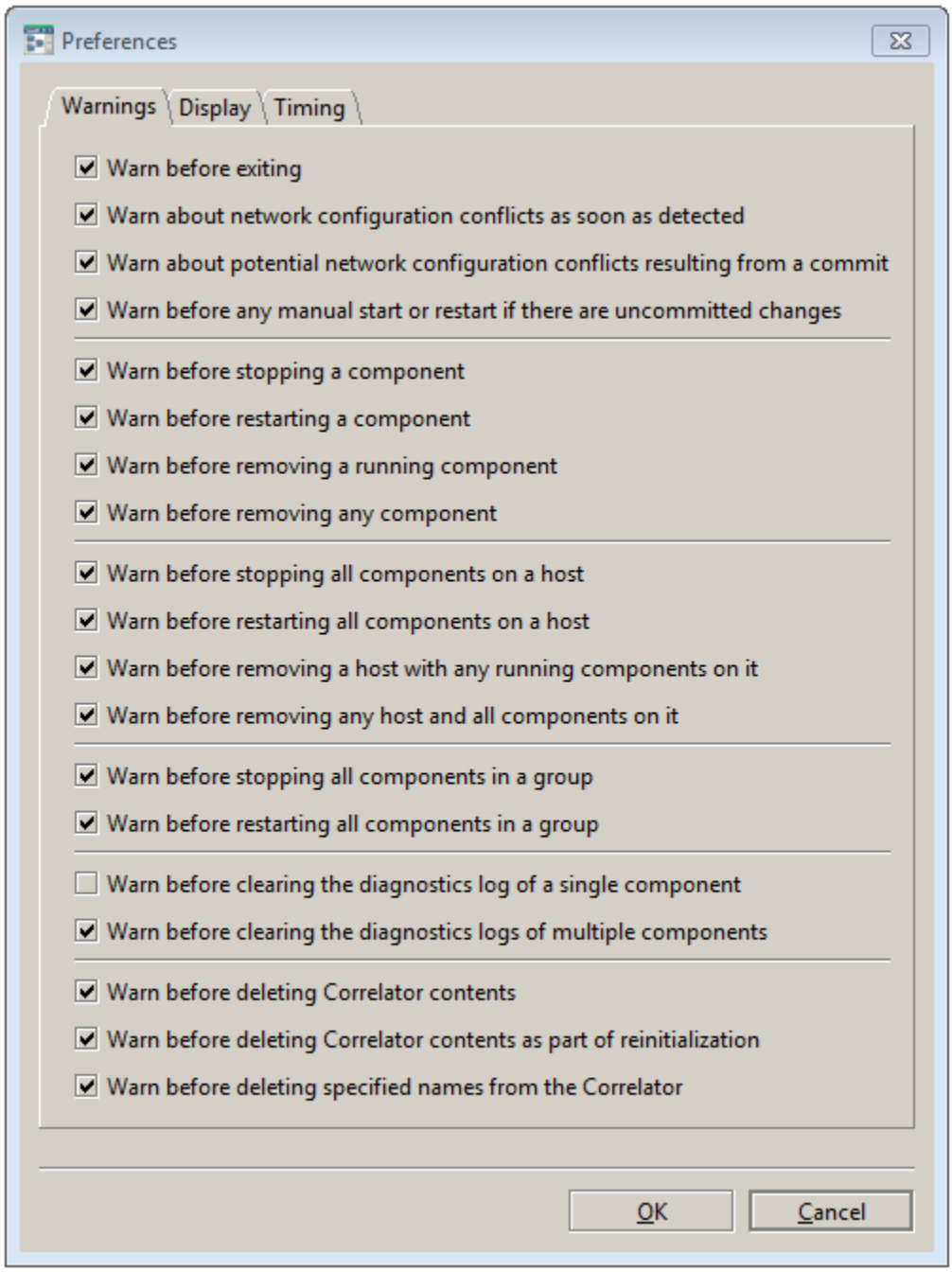
These options are set using the Preferences dialog. This can be accessed by selecting the Preferences… option from the Options menu on the EMM menu.

The dialog has three tabs: Warnings, Display, and Timing.

Using the Management and Monitoring Console

Warnings

The illustration below shows the Warning tab.



The Warning tab allows various confirmation dialogs to be turned off if desired.

Miscellaneous warnings:

- `Warn before exiting` – Ask for confirmation when the user asks to shut down EMM. Although all settings in EMM are preserved on shutdown and re-read the next time it is restarted, the background monitoring and automatic restart functionality only works while EMM is running.
- `Warn about network configuration conflicts as soon as detected` – Warn if a component conflict (same host-port combination) is detected.

- `Warn about potential network configuration conflicts resulting from a commit` – Ask for confirmation if a component conflict (same host-port combination) would result from a `Commit Changes` operation.
- `Warn before any manual start or restart if there are uncommitted changes` – Provide a warning when a component is started or restarted from EMM and changes have been made to the component's temporary working configuration that will be ignored when the component is started, because they have not yet been committed.

Component operation warnings:

- `Warn before stopping a component` – Ask for confirmation when the user tries to stop a component.
- `Warn before restarting a component` – Ask for confirmation when the user tries to restart a component.
- `Warn before removing a running component` – Ask for confirmation when the user tries to remove a component that is running, and provide the option of removing without stopping the component.
- `Warn before removing any component` – Always ask for confirmation when the user tries to remove a component, whether it is running or not.

Host operation warnings:

- `Warn before stopping all components on a host` – Ask for confirmation when the user tries to stop all components on a specific host.
- `Warn before restarting all components on a host` – Ask for confirmation when the user tries to restart all components on a specific host.
- `Warn before removing a host with any running components on it` – Ask for confirmation when the user tries asks to remove a host with any components on it that are known to be running; this provides the option of removing the host without stopping running components.
- `Warn before removing any host and all components on it` – Always ask for confirmation when the user tries asks to remove a host, whether it contains running components or not.

Group operation warnings:

- `Warn before stopping all components in a group` – Ask for confirmation when the user tries to stop all components in the `Apama Managed Components` group.
- `Warn before restarting all components in a group` – Ask for confirmation when the user tries to restart all components in the `Apama Managed Components` group.

Clearing log warnings:

- `Warn before clearing the diagnostics log of a single component` – Ask for confirmation before clearing the diagnostics log when a single component is selected.
- `Warn before clearing the diagnostics log of multiple components` – Ask for confirmation before clearing the diagnostics logs for all the components associated with a selected host or group.

Deleting correlator contents warnings:

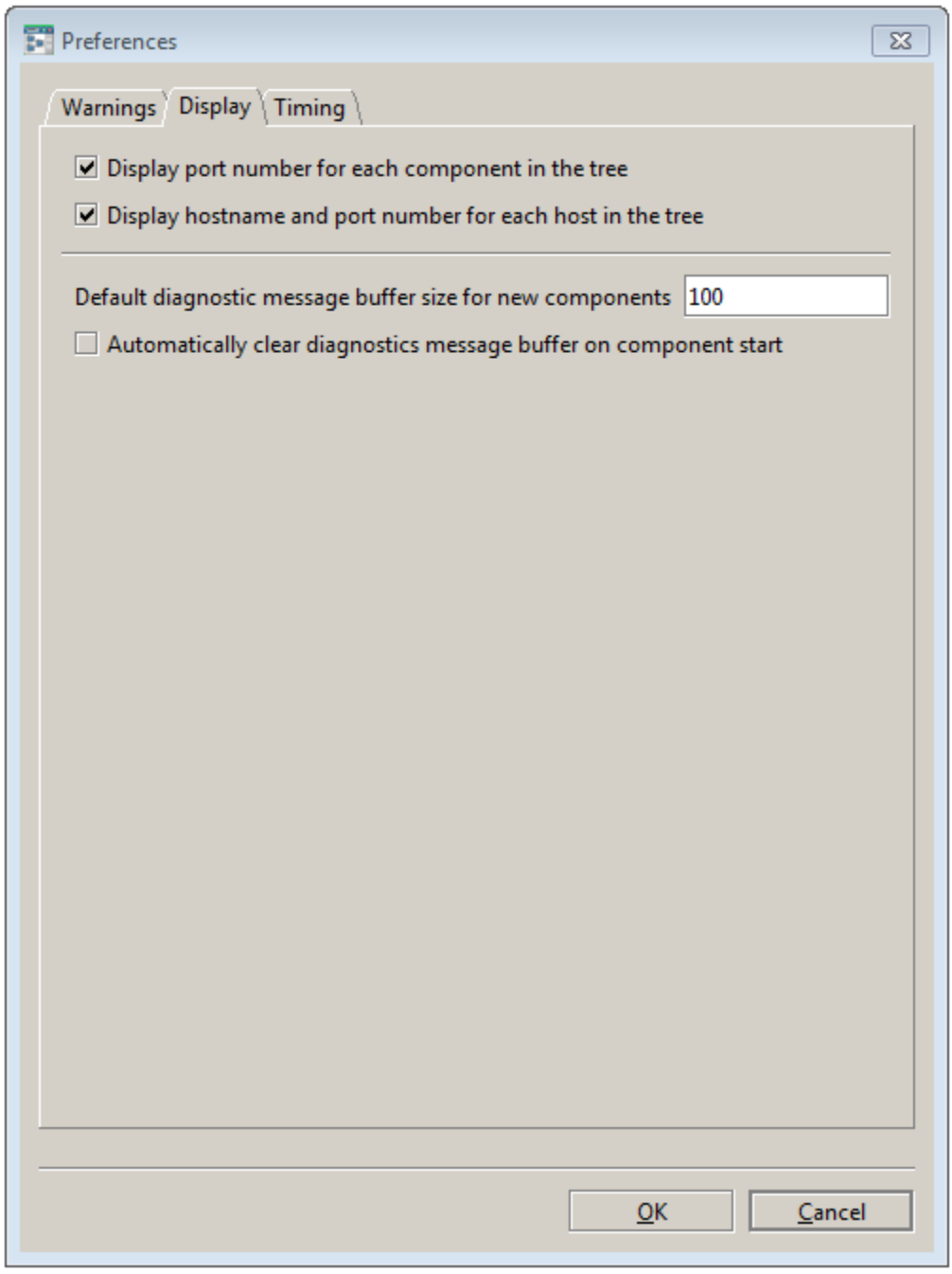
- `Warn before deleting Correlator contents` – Ask for confirmation before deleting all Apama Event Processing Language (EPL), Correlator Deployment Package (CDP), or JMon files in the correlator. The warning will be displayed after clicking the Delete button on the correlator's Inspect tab.

- `Warn before deleting Correlator contents as part of reinitialization` – Ask for confirmation before re-initializing the correlator with the Delete everything loaded in the engine before reinitializing check box is enabled.
- `Warn before deleting specified names from the Correlator` – Ask for confirmation before deleting specific EPL, CDP, or JMon files. The warning will be displayed after clicking the Delete button on the correlator's Inspect tab.

Preferences

Display

The illustration below shows the Display tab.



The following settings may be configured from the Display tab of the **Preferences** dialog:

- Display port number for each component in the tree – If enabled, the listening port for each component will be displayed alongside its name in the Navigation Pane. This makes it easier to check for conflicts (since the port number must be unique per host).
- Display hostname and port number for each host in the tree – If enabled, the actual hostname (as opposed to its EMM descriptive name) and the port its Sentinel Agent is listening on are displayed alongside that host's name in the Navigation Pane. This makes it easier to avoid conflicts in host management.
- Default diagnostic message buffer size for new components – Each component has a Diagnostics tab in its Display Pane, containing a list of the most recent status messages for the component. The

maximum number of messages (after which the oldest will be discarded) may be configured on a per-component basis; this Preferences option specifies the default size of the message buffer that is assigned to new components. By default the buffer size is `100` lines. Minimum size is 1 line.

- Automatically clear diagnostics message buffer on component start – If enabled, removes all log messages for a component when it starts. This is useful during development work, but it should not be enabled in a production system.

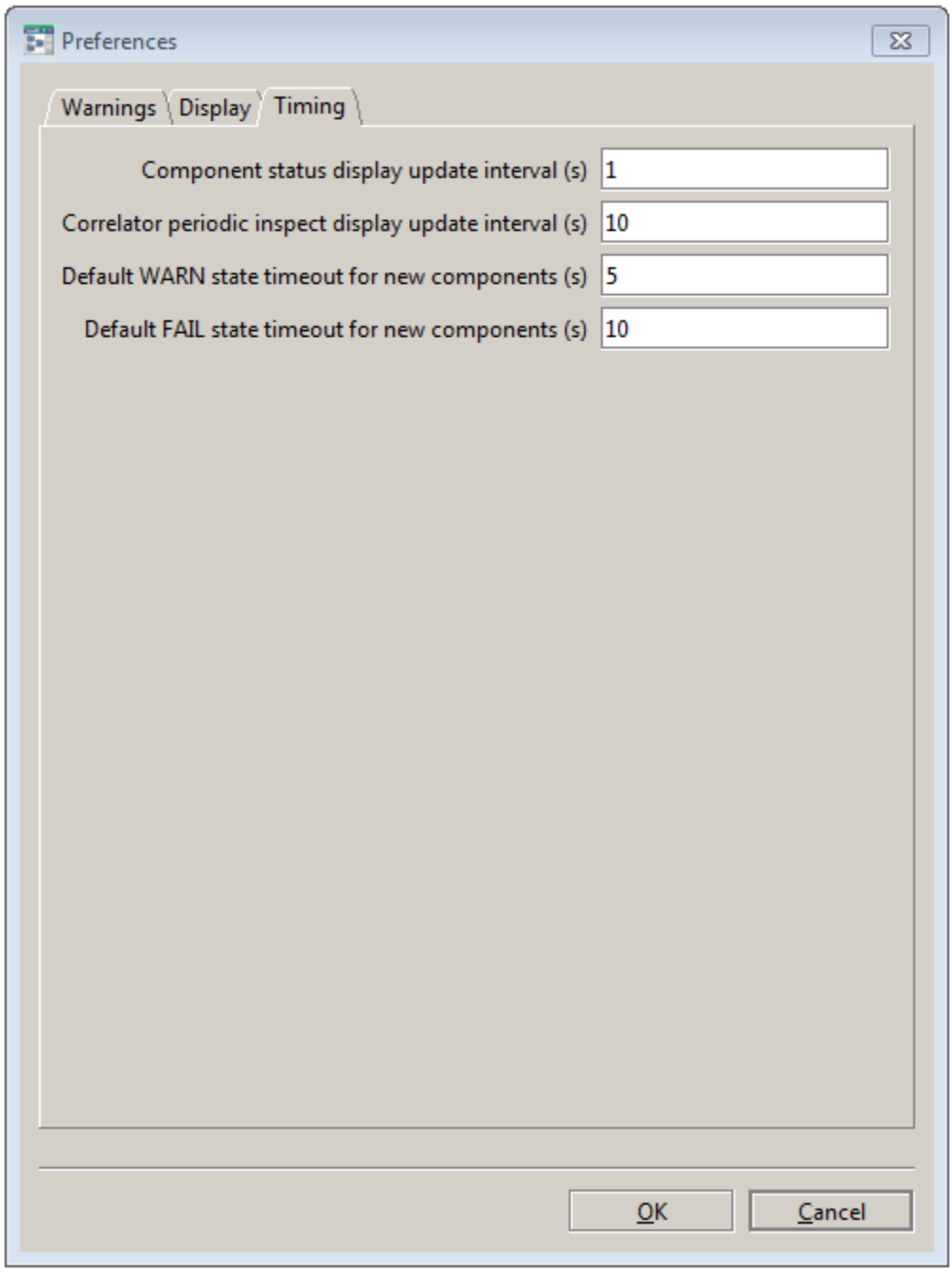
Preferences

Timing

The illustration below shows the Timing tab.

The following settings may be modified from the Timing tab of the **Preferences** dialog:

- Component status display update interval (s) – This parameter specifies how frequently a component's statistics should be updated. By default this is every `1` second.
- Correlator periodic inspect display update interval (s) – This parameter specifies how frequently a correlator's inspect information should be refreshed. By default this is every `10` seconds.
- Default WARN state timeout for new component (s) – This parameter specifies the initial value of `WARN state timeout` used for new components (see "Add correlator" on page 30). By default this is `5` seconds.
- Default FAIL state timeout for new component (s) – This parameter specifies the initial value of `FAIL state timeout` used for new components (see "Add correlator" on page 30). By default this is `10` seconds.



Preferences

Chapter 3: Deploying and Configuring Correlators

Apama correlators are the core event processing and correlation engines for Apama applications. This section describes how to start and manage correlators using the EMM console. You can also start and manage correlators using Apama command line utilities; for more information on this, see "Correlator Utilities Reference" on page 103.

This topic describes how to use the EMM menu items to carry out operations on Apama correlators. However, all these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

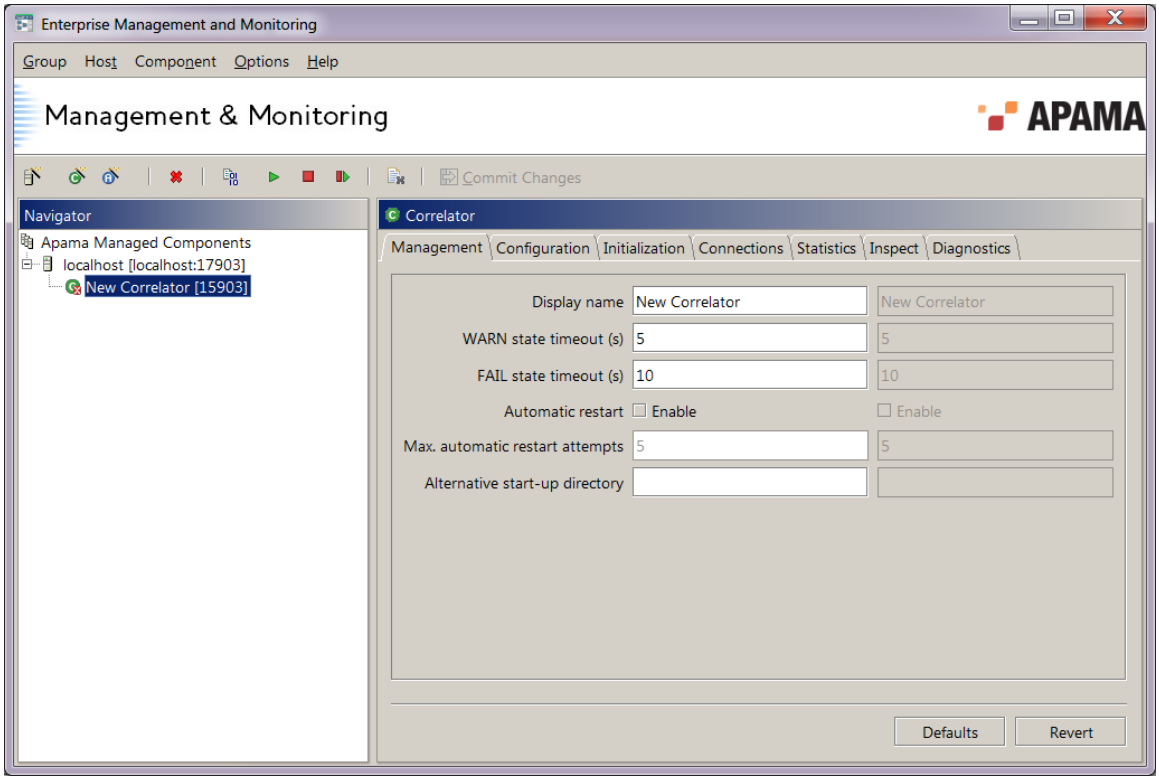
Before operations can be carried out on these components, a Sentinel Agent must be installed and running on the target host, and the EMM console must have been configured to manage that host, as detailed in "Managing hosts" on page 18.

Adding correlators

To define and configure a new correlator:

1. Select the host where you want to add the correlator in the EMM Navigation Pane.
2. Select Component > Add correlator from the EMM menu or click the  button on the tool bar

   This adds a correlator to the host in the Navigation Pane, selects it, and displays the Management tab within the Details Pane;



The other tabs available on this Details Pane for a correlator are the Configuration tab, the Initialization tab, the Connections tab, the Statistics tab, the Inspect tab, and the Diagnostics tab. These are discussed in detail in "The correlator tabs" on page 42.

EMM initializes the new correlator with a set of default options, which are normally safe. The default value for the listening port (which has to be unique per host) is automatically selected so as not to conflict with any other known components.

Before the new correlator can be started for this first time you must 'commit' its configuration using the Commit Changes ( ) button. Note that most of the correlator's configuration options only apply when the component is started up, so changes committed after the component is already running will usually not take effect until it is restarted.

Deploying and Configuring Correlators

The correlator tabs

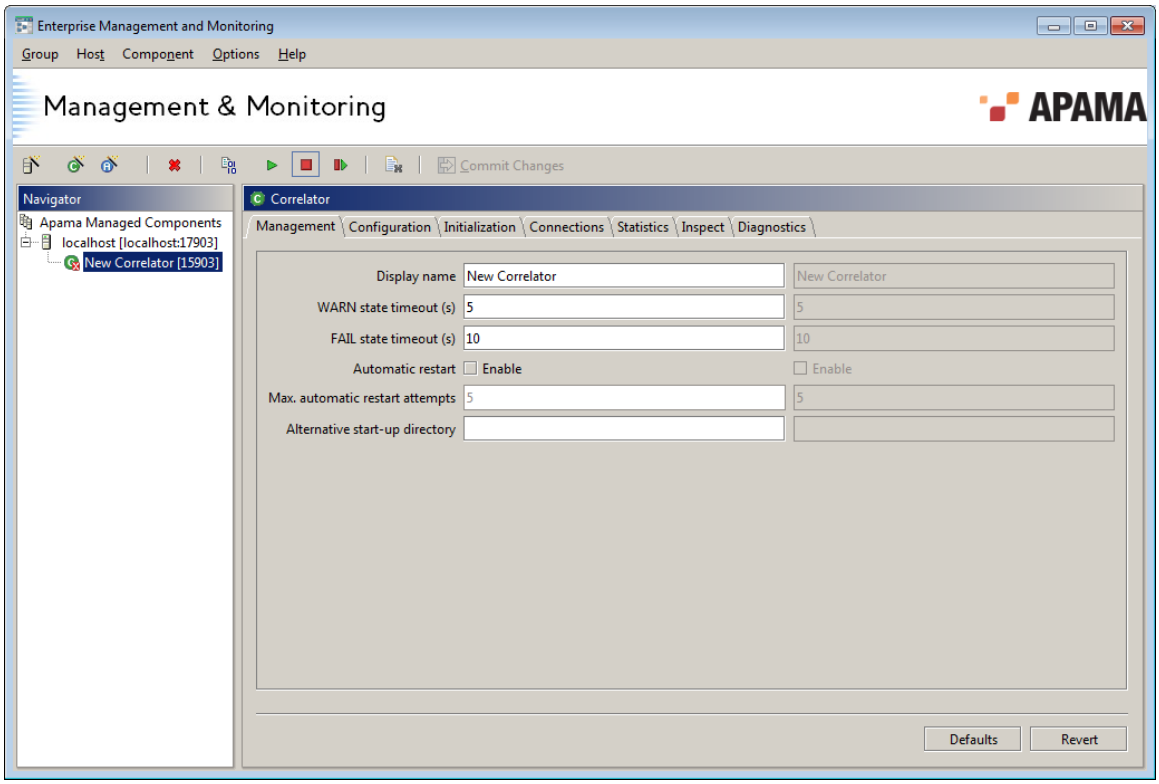
When a correlator is selected in the Navigation Pane, the Details Pane displays the following tabs:

- "Management tab" on page 43
- "Configuration tab" on page 44
- "Initialization tab" on page 45
- "Connections tab" on page 49
- "Statistics tab" on page 53
- "Inspect tab" on page 56

- "Diagnostics tab" on page 58

Management tab

This tab contains a number of parameters that are relevant to managing a correlator.



The parameters on the Management tab are:

- `Display name` – This is the name to associate with this correlator in the Navigation Pane. The name can be changed at any time, even if the correlator has already been started.
- `WARN state timeout (s)` – This is the number of seconds to wait before moving the component into the `WARN` state if it is not changing state (stopping/starting/ restarting) as required. This setting can be changed at any time. The default is 5 seconds.
- `FAIL state timeout (s)` – This is the number of seconds to wait after entering the `WARN` state before moving into the `FAIL` state, if the component has still failed to change state as expected. This setting can be changed at any time. The default is 10 seconds.
- `Automatic restart enable` – Tick to configure EMM to monitor the selected component. If it were to stop unexpectedly, EMM would automatically restart it – after waiting the amount of time required for it to enter the `FAIL` state (and subject to the configured limit on the number of restart attempts described below). This setting can be changed at any time.

Note: Component monitoring and auto-restart is carried out by EMM and requires EMM to be running.

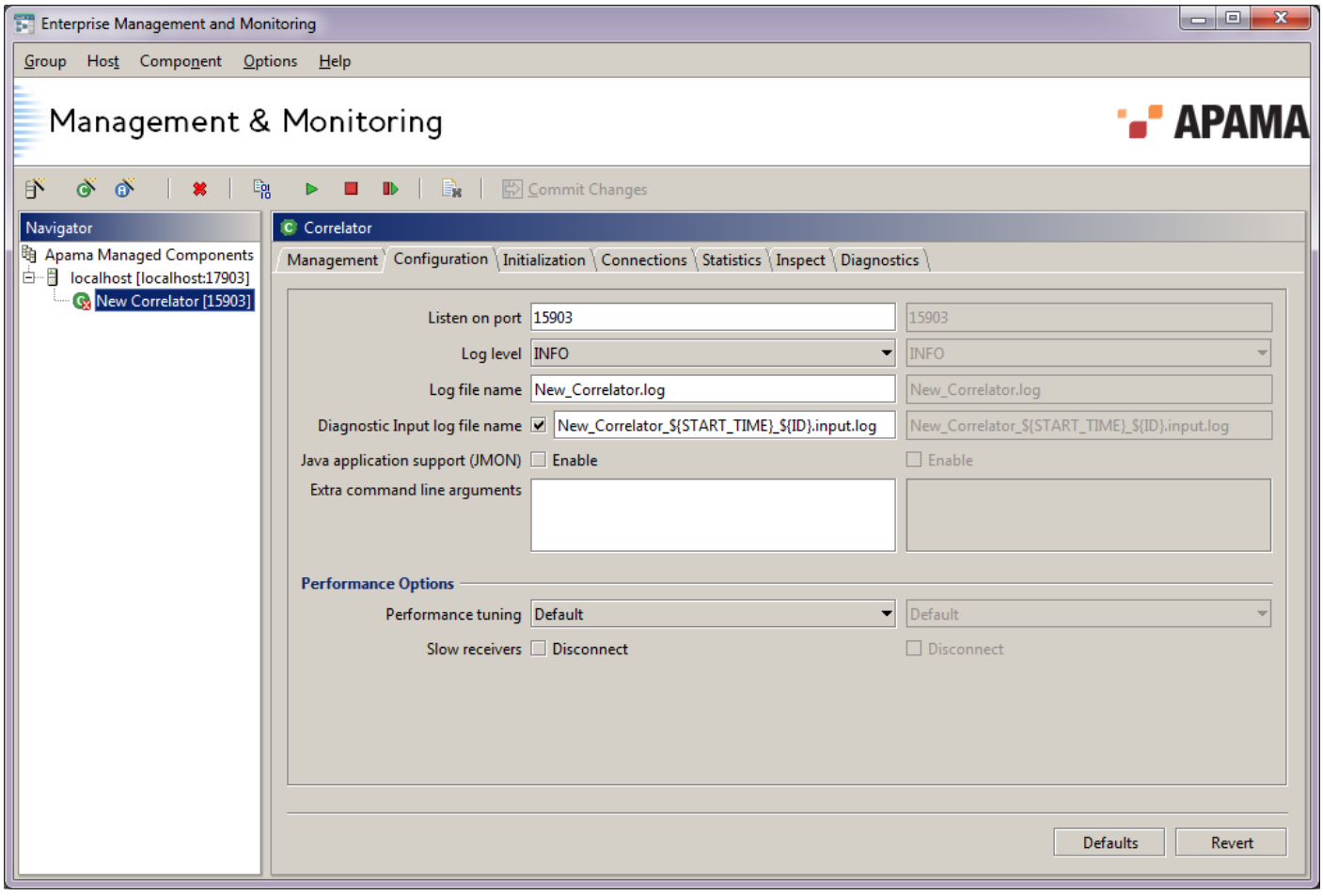
- `Max. automatic restart attempts` – This option is only available if `Automatic restart` is enabled. It specifies the total number of automatic restarts to perform (or attempt) without user intervention. The count is made from when you last enabled `Automatic restart` and committed, or explicitly stopped/started/ restarted the correlator. This setting can be changed at any time.
- `Alternative start-up directory` – By default all components are started from the current working directory of the Sentinel Agent running on the host in question (see "Sentinel Agents" on page 19). If you wish, you can provide an alternative startup folder here.

Click on the Commit Changes button when you are finished customizing the new correlator. This applies these outstanding changes.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

Configuration tab

This tab contains a number of parameters that configure how a correlator operates. As all of these are startup parameters, you should adjust them before you start the component. If the component is already running they will only take effect the next time it is started.



The following parameters are available for a correlator:

- `Listen on port` – Port on which the correlator should listen for monitoring and management requests (default is `15903`). The correlator will fail to start if this port is in use by any other component, or for that matter by any other software that is running on the relevant host.

  The highest permitted port number for correlators and other components managed through EMM is 65534.

- `Log level` – Sets the log level the event correlator should log at. This must be one of `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE` (in increasing order of verbosity). The default level is `INFO`.

- `Log file name` - Sets the filename that the event correlator should write log messages to (on the file system of the host the correlator is running on). It is recommended that an absolute path be provided. See "Specifying paths and filenames in the Details Pane" on page 33 for more information about setting filename options correctly.

- `Diagnostic Input log file name` - Enables and sets the filename for a diagnostic input log. A diagnostic input log collects all messages sent to the correlator and writes then to a file on the file system of the host the correlator is running on. The input log can help Apama technical support diagnose problems with a correlator or an application running on the correlator.

  It is recommended that an absolute path be provided. See "Specifying paths and filenames in the Details Pane" on page 33 for more information about setting filename options correctly. See also: "Replaying an input log to diagnose problems" on page 161.

- `Java application support (JMON)` – Enables support for JMon applications. If this is not set, any attempt to inject a JMon application either using `engine_inject -j` or by adding a `.jar` file `Inject` initialization action will result in an error. The correlator's performance is improved when Java application support is disabled.

- `Extra command line arguments` – This option allows additional unspecified command line arguments to be passed to the correlator. For example these might be special options to pass to the embedded JVM used for JMon applications, or special settings provided to you by Apama Customer Support to address specific issues.

- `Performance tuning` – Select `Slow receivers` to indicate that the correlator should disconnect any receiver whose output queue becomes full; that is, a receiver that cannot consume events fast enough. Without this option, the event correlator would eventually stop processing events altogether, (and block) until some output events are consumed by the receiver, freeing space on its output queue.

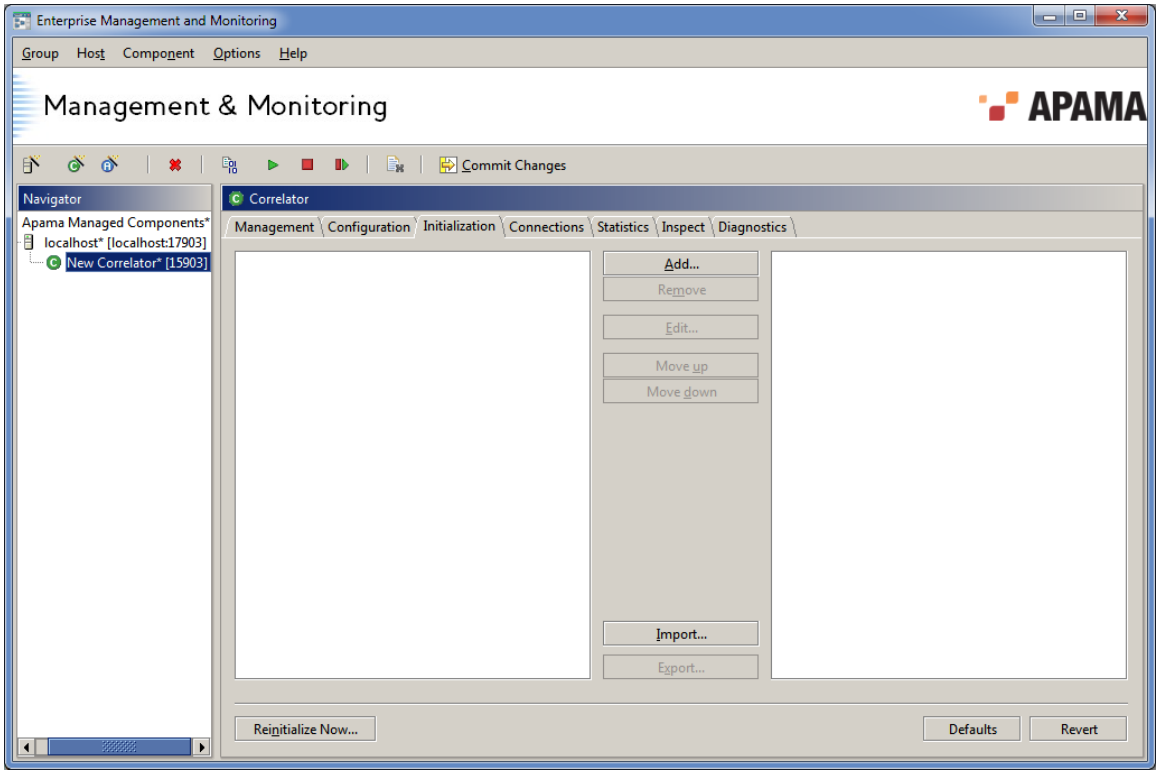
Click Commit Changes when you are finished customizing the new correlator. This applies these outstanding changes for use when the correlator is actually started.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

Initialization tab

This tab allows you to specify the monitors and events that will be automatically executed whenever the component is started by EMM. The files you specify typically have `.mon`, `.evt`, and `.jar` extensions. Note that if the component is started outside EMM and then added to EMM for management, any initialization actions are only executed the next time it is started through EMM.

Note: By default, EMM uses the default encoding of the system on which EMM is running to read Apama Event Processing Language files (`.mon`) and event (`.evt`) files. If you want to inject UTF-8 encoded files, ensure that you use an editor that adds a Byte Order Mark (BOM) at the start of the file. For example, on Windows, the Notepad editor inserts a BOM when you save a file.

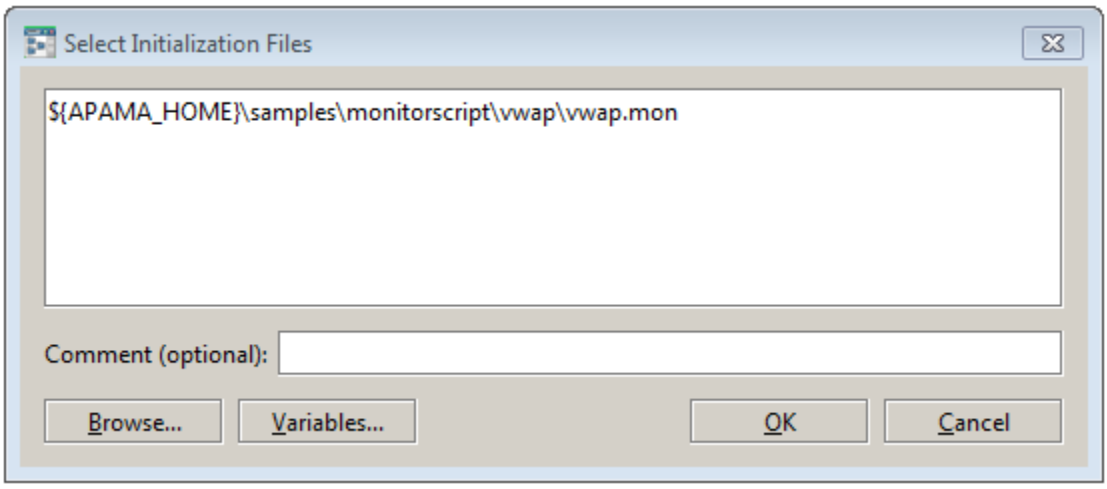


Adding initialization actions

To add initialization actions:

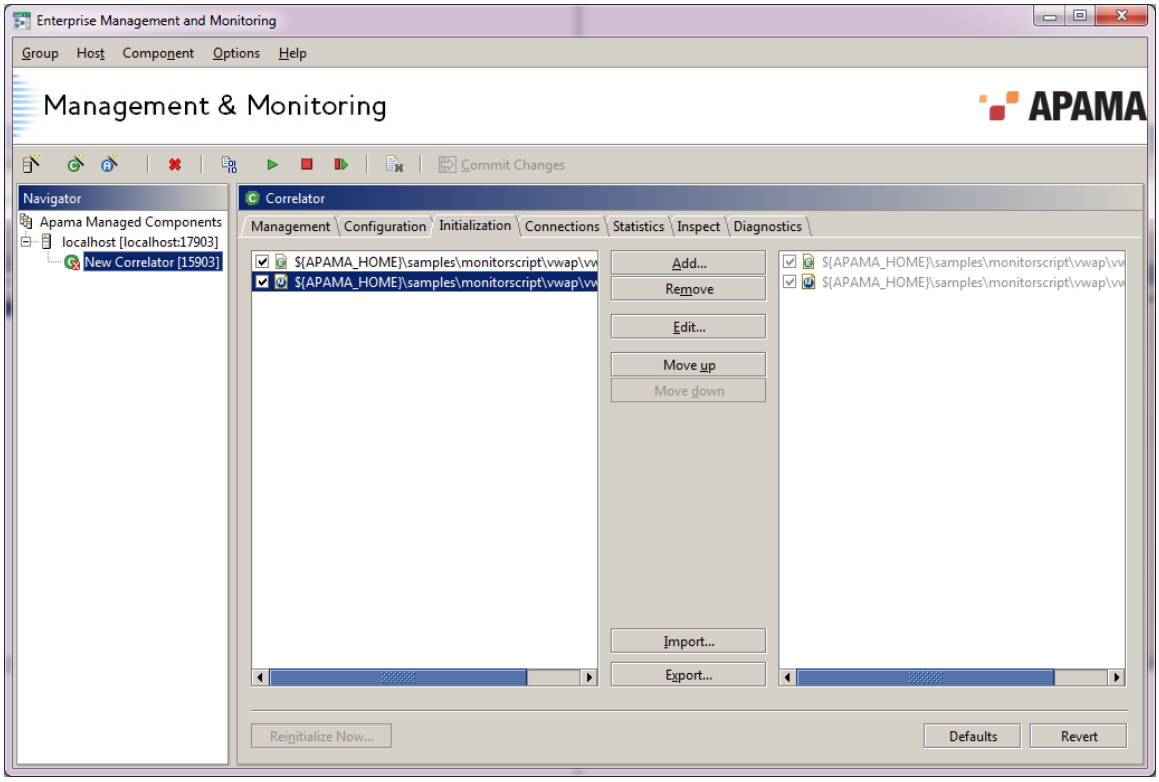
1. On the Initialization tab, click the Add button.

   This displays the **Select Initialization Files** dialog.



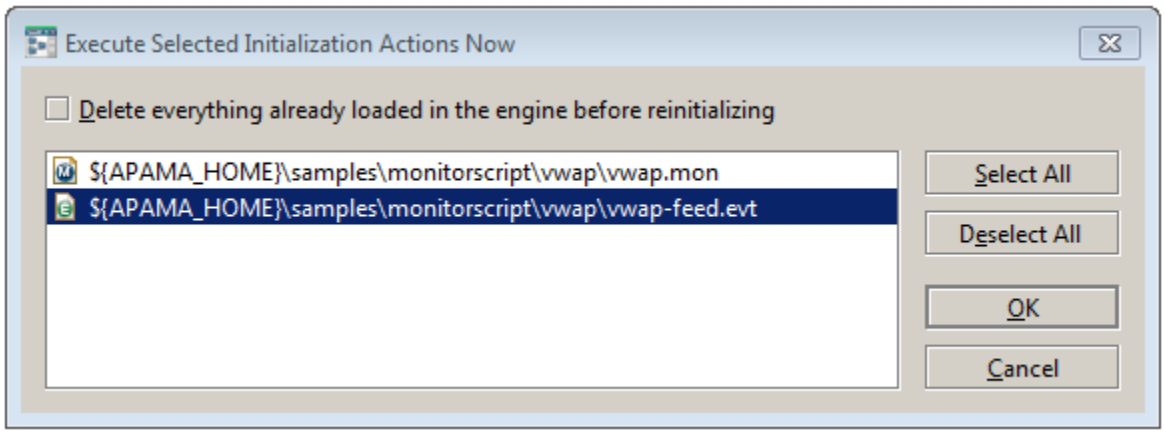
2. Enter the name of the monitor or event file to use for initializing the component. You can also click Browse to navigate to the directory containing the files to use for initializing the component and select the files. Use **Ctrl** and **Shift** to select more than one file name. You can add a comment that will be displayed next to the file name on the Initialization tab.
3. Click Variables to select an Apama variable; this alleviates the need to type a full path name. You can also use your own (non-Apama) environment variables by specifying them in the form: `${env_var:foo}`.
4. When you have specified the name of the file(s) you want to add, click OK.
5. You can also specify a file with a `.txt` extension containing a list of monitors and event files with which to initialize the component. Click Import and enter the name of the file that contains the initialization files. Files for this purpose are formatted with the name of each `.mon`, `.evt`, or `.jar` file on a separate line. Apama Studio exports initialization files in this format.
6. Instead of entering or selecting file names as in Step 2, you can drag and drop files from a Windows Explorer window to the left hand list of the Initialization tab.

   All filenames specified here will be accessed on the local computer on which the EMM console is running – not the machine hosting the correlator (unless of course the correlator is also running on the local machine).
7. After the previous steps, EMM displays the initialization files in the left hand list. The check boxes in the left hand list allow you to select which initialization actions will take effect; actions with unchecked boxes will be ignored. Click the Commit button to apply the changes; this will display the initialization files in the right hand list. The initialization actions will take place the next time the component starts



When an action is selected, the following buttons are also available:

- Remove – Remove the selected initialization action.
- Edit – Allows the comment and/or filename associated with an initialization action to be changed.
- Move up – Move the selected initialization action up in the list. The order is significant because the actions will be executed in the order they appear in the list (from top to bottom).
- Move down – Move the selected initialization action down the list.
- State restore — Restore the entire runtime state of the correlator from a state image file or dump. Due to the nature of Restore operations, only one Restore initialization action may be specified for each correlator, and it will always be executed before any other Initialization action.
- Import – Allows you to specify a text file containing initialization actions.
- Export – Saves the component's initialization actions to a text file.
- Reinitialize Now... — Initializes the Component with the initialization files specified. When you click Reinitialize Now..., EMM displays the **Execute Selected Initialization Actions Now** dialog.

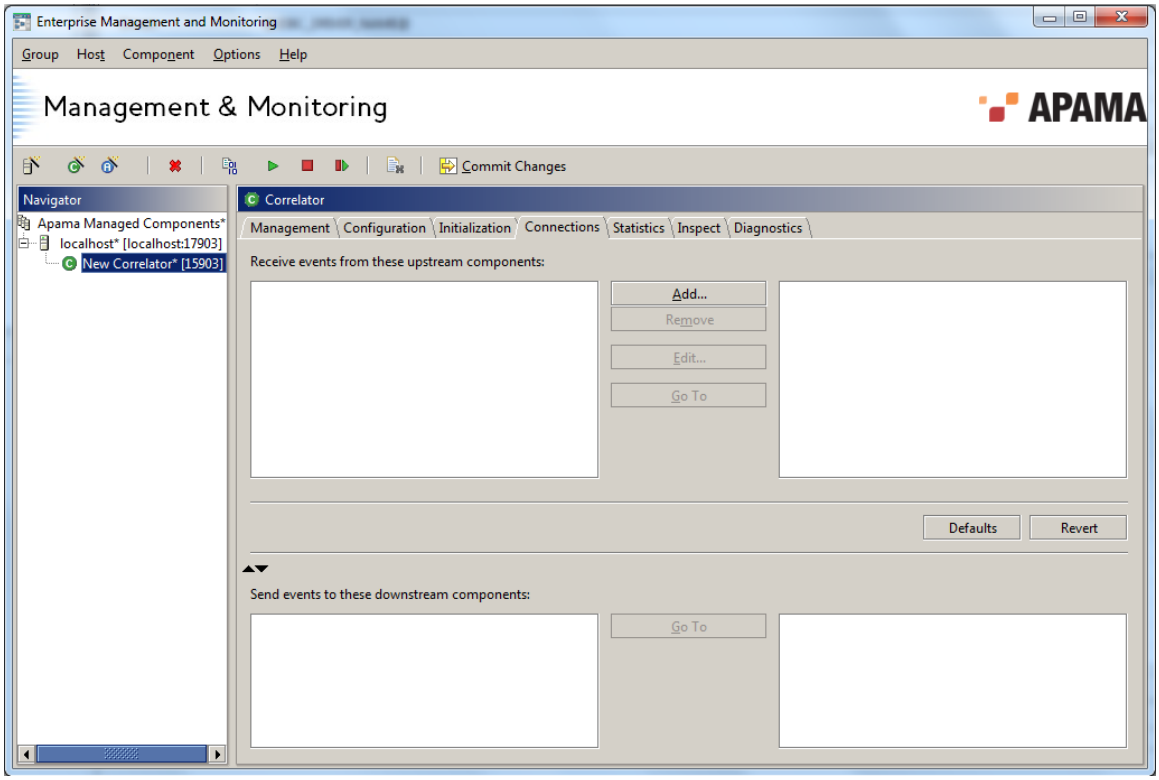


This dialog allows you to specify which actions you want to use and also gives you the opportunity to delete any Apama EPL or JMon files loaded in the correlator before you reinitialize it.

- Defaults — Removes any initialization files that have been added to the component.
- Revert — Removes initialization files that have been added to the component since the last time changes were Committed. Initialization files added prior to the last Commit are retained.

Connections tab

This tab displays any other components that are connected to this component in both upstream and downstream directions. “Upstream” components are those components that this component receive events from; “downstream” components are those that this component sends events to. Because the “owner” of the connection is always the downstream component, the list of downstream connections is for display only.

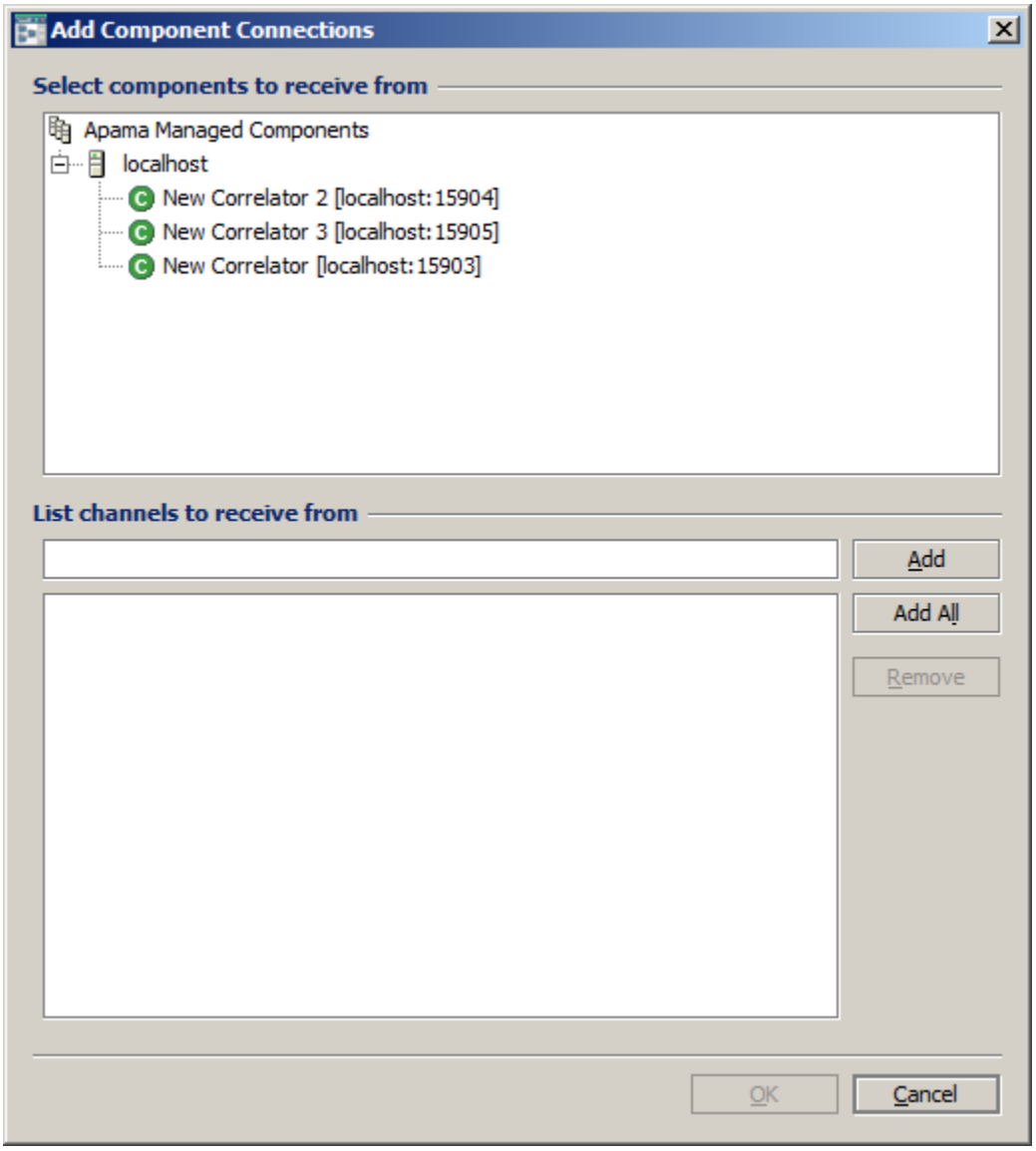


Adding new upstream connections

The Connections tab allows you to add new upstream component connections.

To specify upstream components you want to connect to this component:

1. Click Add to display the **Add Component Connections** dialog.

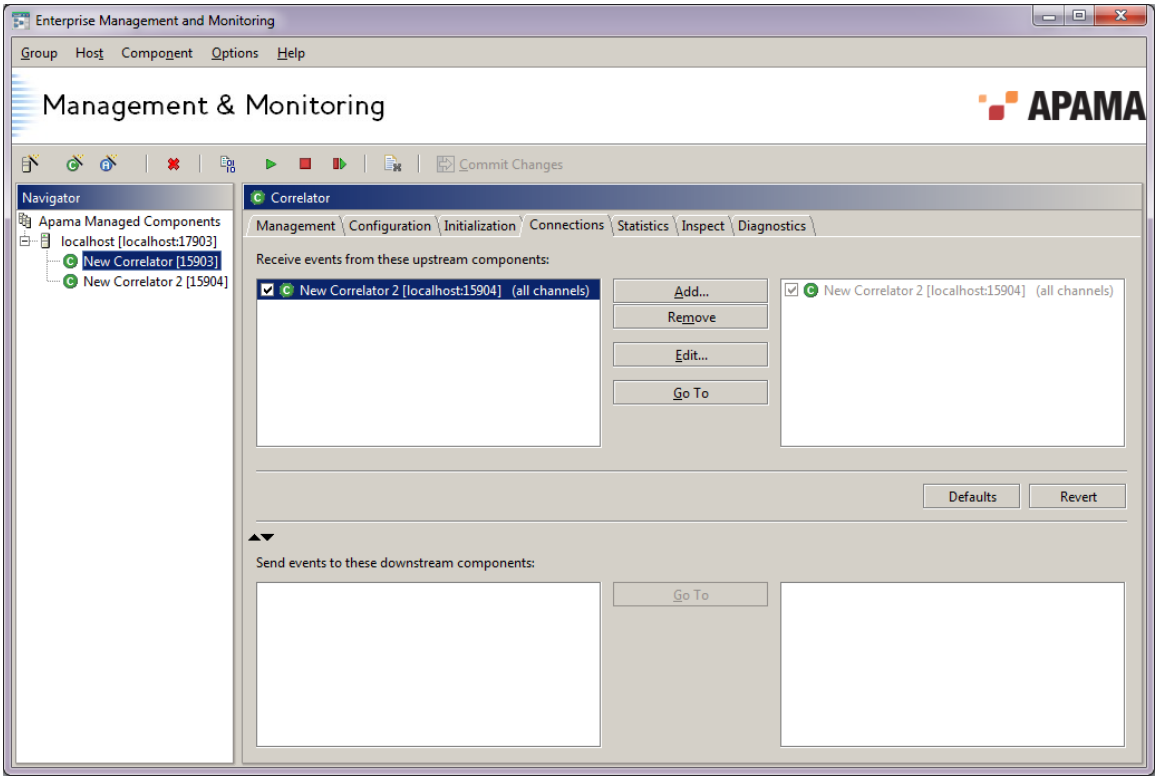


2. From the Select components you want to receive from list, select one or more components.
3. In the List channels to receive from field, enter the name of one or more channels associated with the component you selected in Step 2 and click Add, or click Add All to receive all channels associated with a component.
4. Click OK. The Connections tab will display the components you added in the left hand list.

   The check boxes in the left hand list of the upstream connections allow you select which connections will take effect in the next step. Connections with checked boxes will be enabled; those with unchecked boxes will be disabled.

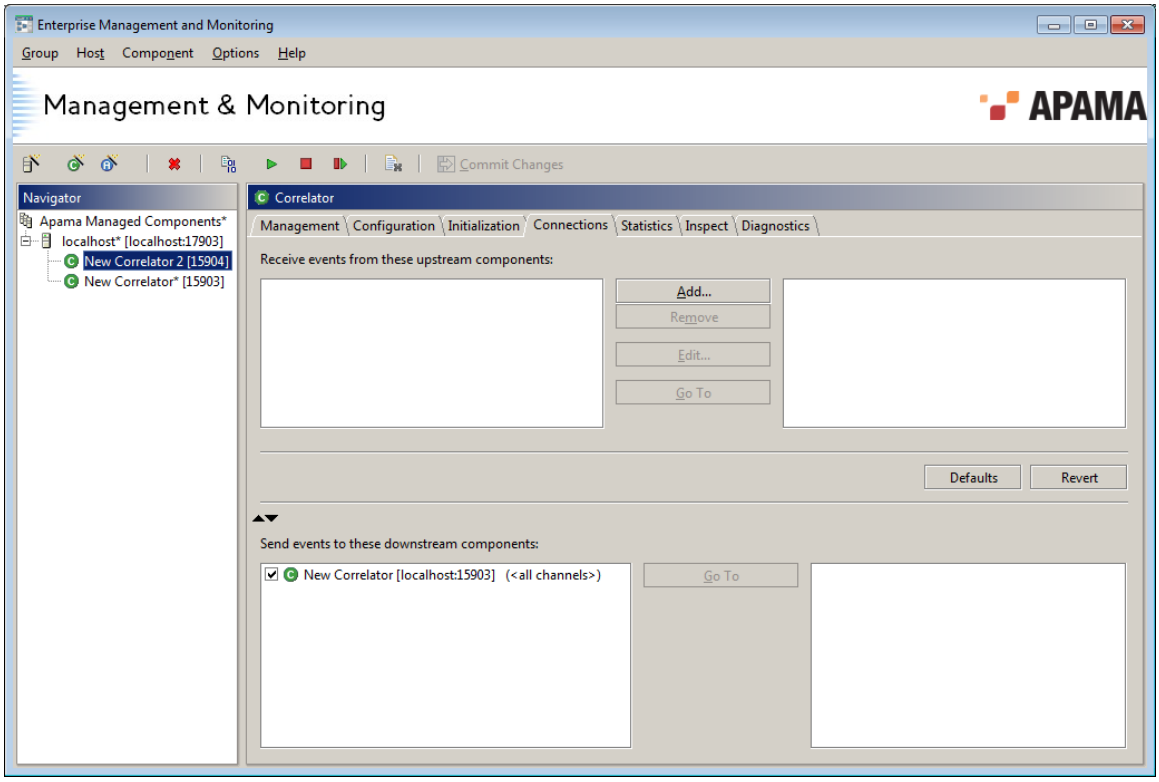
5. Select the Commit changes menu item or the click the Commit Changes ( ) button to make your changes take effect.

   This adds the connected components to the right hand list of the Connections tab.



Downstream connections

Downstream components connected to this component are displayed in the Send events to these downstream components field. The following ilulustration shows a correlator with a downstream connection.



If you select an upstream or downstream component and click the corresponding Go To button, EMM switches the display to show the Connections tab for the connected component.

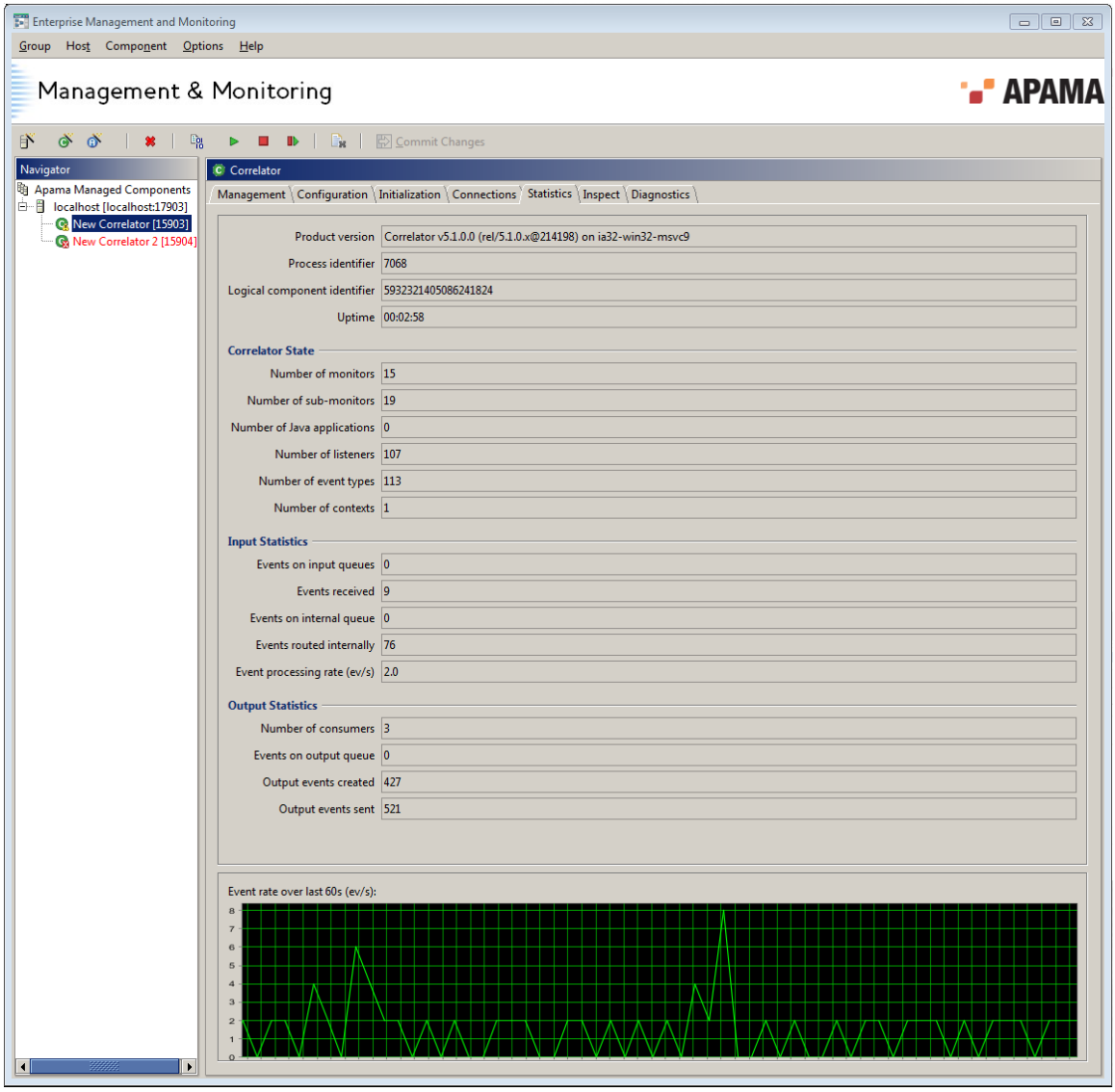
Statistics tab

This tab allows you to monitor of the status of an operational correlator.

Status information is refreshed regularly, by default, once a second. You can change this rate by changing the `Component status display update interval (s)` value on the Timing tab within the **Preferences** dialog. The **Preferences** dialog can be accessed from the Preferences... option from the Options menu on the Menubar.

No status information will be displayed if EMM cannot communicate with the selected correlator. If this is the case it is usually because the correlator has not yet been started or has been stopped. However, it could also be due to network failure. In this situation, statistics will reappear once the connection is restored.

You may need to scroll the display to view all the information on the Statistics tab. The following illustration shows the Statistics tab for an active correlator.



The Statistics tab displays the following information:

- `Product version` — The Apama release number.
- `Process identifier` — The identifier assigned to this component by the operating system.
- `Logical component identifier` – A numeric identifier for this component. This is useful for matching components with information contained in log files.
- `Uptime(s)` – The time in seconds since this correlator was started. This time is maintained and reported by the component itself, so if the correlator was started independently of EMM and only managed by EMM later, the value would still be accurate.
- `Number of monitors` – The current number of Apama monitors injected and instantiated inside the correlator. This figure changes upwards and downwards as monitors are injected, deleted or just expire.
- `Number of sub-monitors` – The number of Apama sub-monitors. Sub-monitors are created by 'spawn' actions within the monitor EPL code. This figure changes upwards and downwards as sub-monitors are spawned, killed or just expire.

- `Number of Java applications` – The number ofJMon applications currently loaded into the correlator. JMon applications do not expire, so this value only decreases when they are explicitly unloaded.
- `Number of listeners` – The number of event listeners created by monitors and sub-monitors in EPL code and by JMon applications.
- `Number of event types` – The total number of event types defined within the correlator. This figure decreases when event types are deleted from the correlator.
- `Number of contexts` – The number of contexts in the correlator. This includes the main context plus any created contexts.
- `Events on input queue` – Across all contexts, the total number of events on input queues.
- `Events received` – The total number of events ever received by the correlator. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of the events seen by the correlator from which the checkpoint was originally made.
- `Events on internal queue` – Across all contexts, the total number of routed events waiting to be processed. The internal routing queue is a high priority queue that is used when events are internally routed by the `route` instruction in EPL code or in JMon applications. For each context, the correlator processes events on the internal queue before processing other events on the input queue.
- `Events routed internally` – Across all contexts, the total number of events that have veen routed since the correlator was started. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of the events routed by the correlator from which the checkpoint was originally made.
- `Event processing rate (ev/s)` – The number of events per second currently being processed by the correlator. This value is computed with every status refresh and is only an approximation.
- `Number of consumers` – The number of event consumers registered with the correlator to receive events emitted by it.
- `Events on output queue` – The number of events waiting on the output queue to be dispatched to any registered event consumers.
- `Output events created` – The total number of output events created by the correlator. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of output events created by the correlator from which the checkpoint was originally made.
- `Output events sent` – The total number of output events dispatched to event consumers by the correlator. This figure varies from the preceding statistic as an output event might be sent to multiple event consumers. This value is preserved by a checkpoint, so if the state of the correlator is restored from a checkpoint file it will reflect the total number of output events sent out by the correlator from which the checkpoint was originally made.

A graph showing the values taken by the `Event rate` statistic over the last 60 seconds is displayed below the figures.

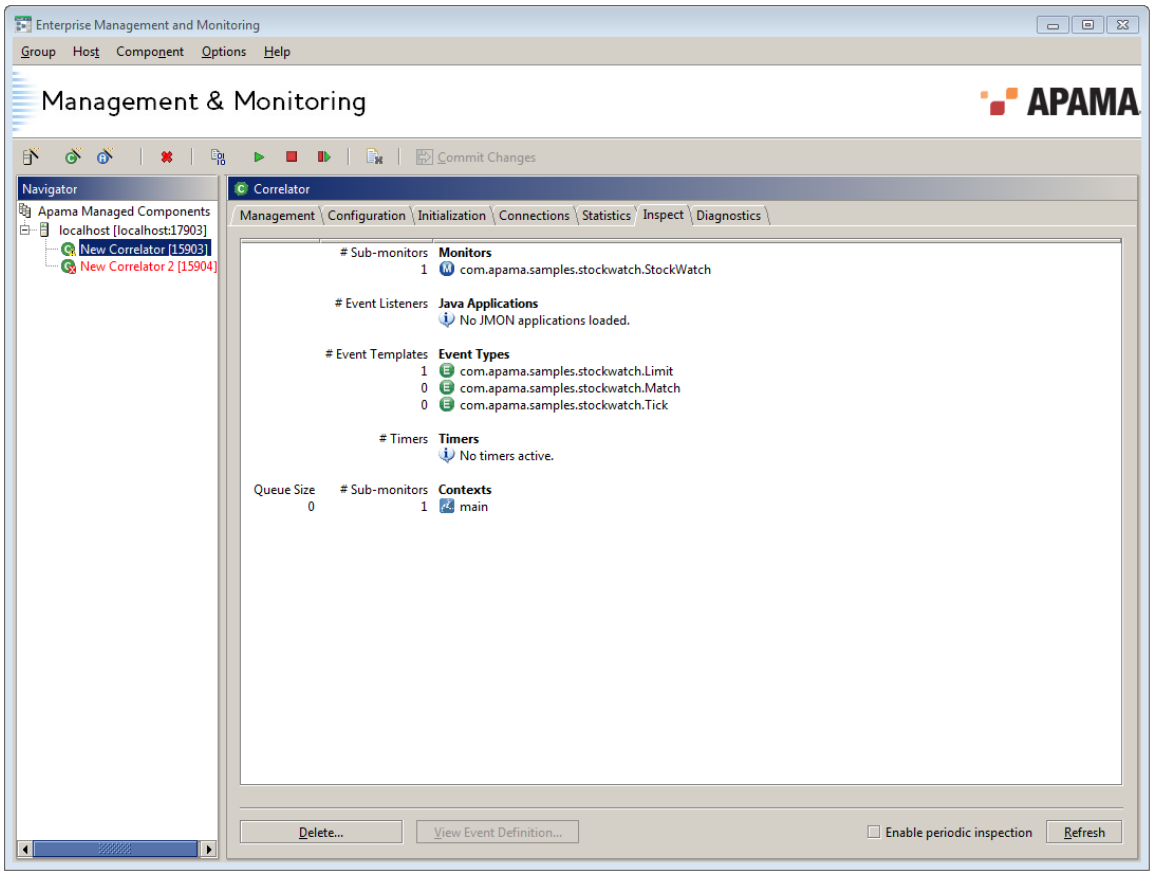
Inspect tab

This tab displays the EPL monitors, JMon applications, and event types currently active within the correlator.

The information on the Inspect tab is retrieved in real time from the selected correlator when you actually select the Inspect tab, and it might take a few seconds to display.

You can configure the Inspect tab to refresh periodically by ticking the Enable periodic inspection checkbox at the bottom-left corner of the tab. By default the information will be refreshed every 10 seconds. You can change this value by changing the `Correlator periodic inspect display update interval(s)` value on the Timing tab within the **Preferences** dialog. The **Preferences** dialog can be accessed from the Preferences… option from the Options menu on the Menu bar

Alternatively you can refresh manually by clicking the Refresh button at the bottom-right corner. The following illustration depicts the Inspect tab for a correlator showing active definitions



The information displayed is organized into the following groups: Monitors, Java Applications, Event Types, and Contexts.

- Monitors — Displays a listing of all the EPL monitors active in the correlator and the number of spawned sub-monitors (if any) for each one.
- Java Applications — Displays a listing of JMon applications loaded into the correlator. Against each one is also displayed the number of active listeners created by that application.

- Event Types — Displays a listing of all the event types the correlator knows about. Against each event type will also be displayed the number of event templates of that type in use at present in active listeners.
- Timers — Displays the current EPL timers active within the system. The four types of timers which may be displayed here are `wait`, `within`, `at`, and `stream`. The `stream` timers are those set up to support the operation of stream networks.
- Contexts — Display the names of the contexts in the correlator and for each correlator, the number of monitor instances (sub-monitors) and the queue size (number of events on the input queue) of each context.

The Inspect tab contains two other buttons:

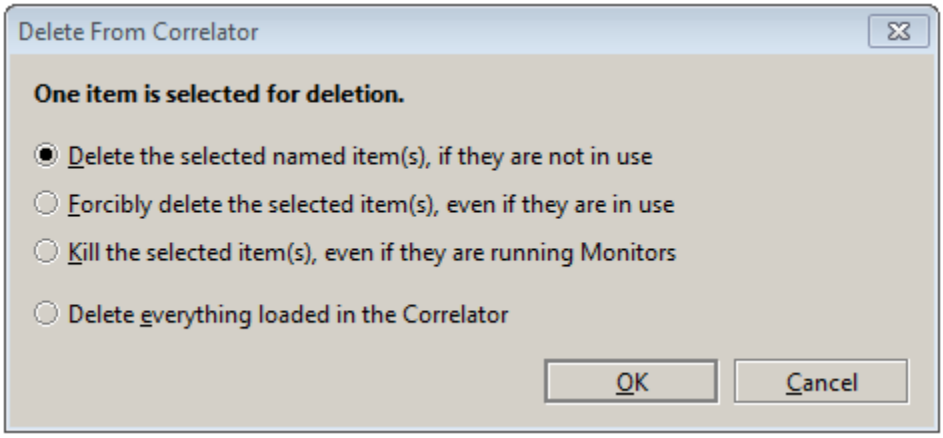
- Delete — Delete an item or items from the correlator.
- View Event Definition — View the code for the definition of a selected Event Type.

Deleting Monitors, Event Types, or JMon applications from a correlator

To delete a monitor, event type, or JMon application from the correlator:

1. In the Inspect tab, select the item or items you want to delete.
2. Click Delete.

   This displays the **Delete from Correlator** confirmation dialog.
3. Select one of the following choices and click OK.
   - Delete the selected named item(s), if they are not in use
   - Forcibly delete the selected item(s), even if they are in use
   - Kill the selected item(s), even if they are running Monitors
   - Delete everything loaded in the correlator



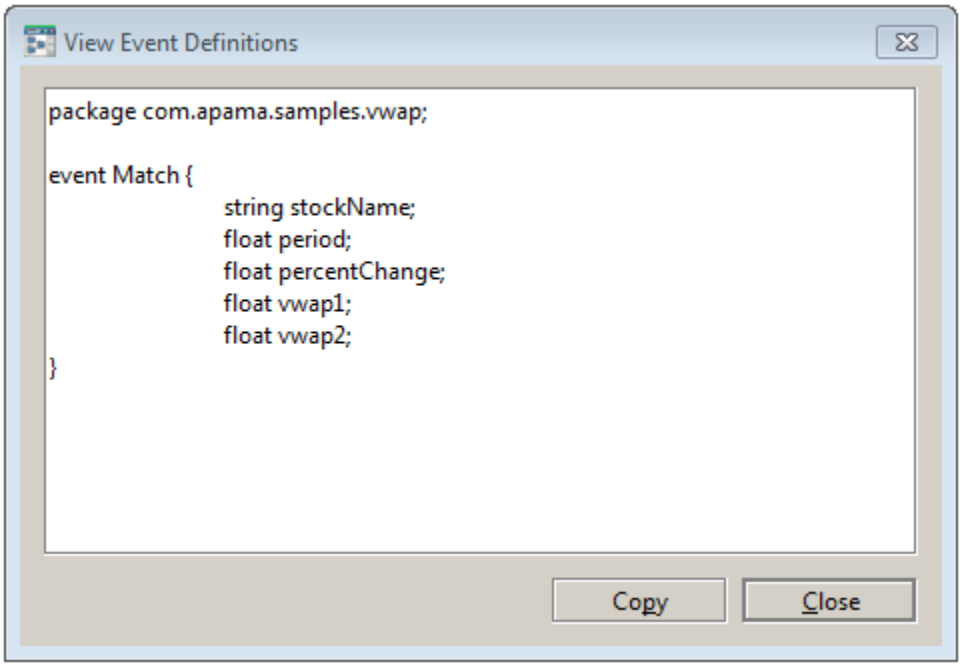
4. Select your choice and click OK.

Displaying event type definition for multiple Event Types

To display the event type definition for one or more Event Types:

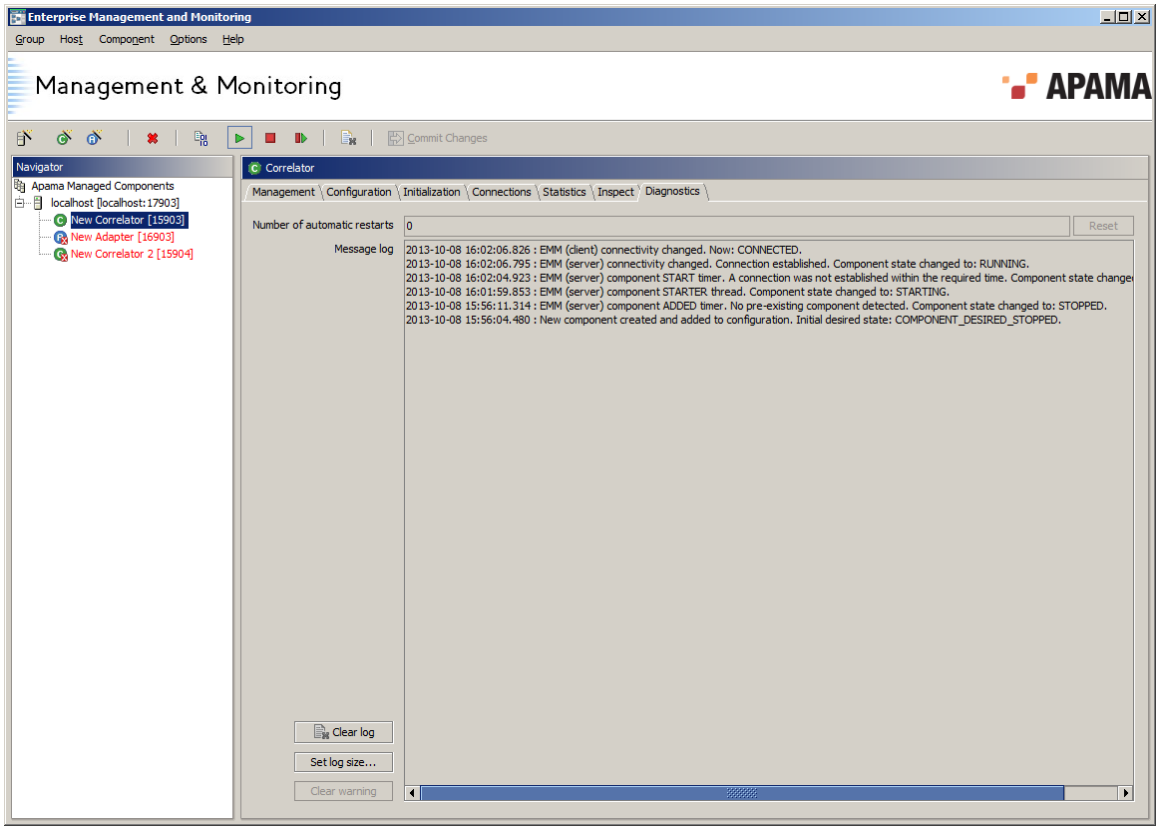
1. In the Inspect tab, select the Event Type or Event Types you are interested in.
2. Click View Event Definition.

   This displays the **View Event Definitions** window.



Diagnostics tab

The Diagnostics tab displays diagnostics information about the changes in the state of the correlator.



It provides two elements of diagnostics information:

- Number of automatic restarts – the total number of automatic restarts since the last time you manually started the component or since you enabled automatic restart. See "Management tab" on page 43 for details of how automatic restart behavior can be configured. The Reset button lets you manually reset the counter and make EMM auto-restart the component again if it is currently in the FAIL state.
- Message log – A log of diagnostic information generated by EMM. Each log entry starts with the date and time of when the event being logged occurred. Note that most entries are tagged as being generated by `EMM (server)` or `EMM (client)`. `EMM (client)` indicates the graphical front-end, whereas `EMM (server)` indicates the backend model.

The Message log has three associated buttons:

- Clear log – Removes all the messages logged for this component.
- Set log size… – Configures the size of the diagnostics log for this component. Note that the default log size assigned to new components can be configured from the Preferences dialog, and is set to `100` entries when Apama is installed. If the size is exceeded the oldest entries are removed to make way for new entries. Minimum size is 1 line.
- Clear warning – Changes the state of the component from `WARN` to `RUNNING`; see "Component status indicators" on page 28 for a full explanation of the meaning of the `WARN` state. This button is only enabled when the component is started and in the `WARN` state.

Chapter 4: Deploying and Configuring Adapters

Adapters allow Apama to interface with external sources of events like message buses, event feeds, or databases. You use the Integration Adapter Framework (IAF) with any of the standard, pre-packaged Apama adapters installed with the product or with custom adapters that you develop. To deploy an adapter, you specify the appropriate information in the adapter's configuration file and then start the IAF using the configuration file. This section describes how to deploy Apama adapters from the EMM console. You can also use Apama command line utilities to start and manage adapters. See also:

- "Using Standard Adapters" in *Connecting Apama Applications to External Components*
- "Developing Custom Adapters" in *Connecting Apama Applications to External Components*

This topic describes how to use the EMM menu items to carry out operations on Apama adapters. However, all these operations can also be carried out from the context menus in the Navigation Pane, and the most useful are also available on the toolbar.

Before operations can be carried out on these components, a Sentinel Agent must be installed and running on the target host, and the EMM console must have been configured to manage that host, as detailed in "Managing hosts" on page 18.

Adding adapters

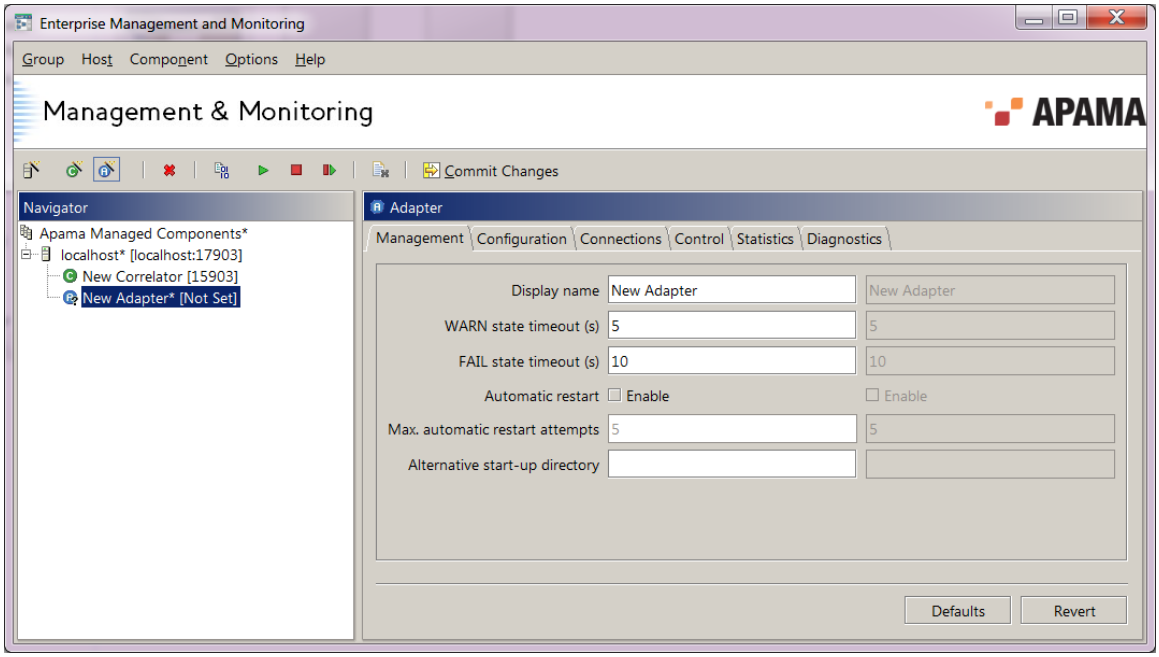
To define and configure a new IAF adapter:

1. Select the host where you want to add the adapter in the EMM Navigation Pane.
2. Select Component > Add adapter from the EMM menu or click the  button on the tool bar.

   This adds an adapter to the host in the Navigation Pane, selects it, and displays the Management tab in the Details Pane. The other tabs available on this Details Pane for an adapter are the Configuration tab, the Connections tab, the Statistics tab, the Control tab, and the Diagnostics tab. These are discussed in detail in "The Adapter tabs" on page 62.

EMM initializes the new IAF adapter with a set of default options, which are normally safe. The default value for the listening port (which has to be unique per host) is automatically selected so as not to conflict with any other known components.

Before the new adapter can be started for this first time you must 'commit' its configuration using the Commit Changes () button. Note that most of the adapter's configuration options only apply when the component is started up, so changes committed after the component is already running will usually not take effect until it is restarted.



Deploying and Configuring Adapters

Configuring adapters

The contents of the Details Pane change according to what is currently selected in the Navigation Pane. When a component is added to a host, it is automatically selected in the Navigation Pane.

Note: As already described, you need to press Commit Changes ( ) before any changes you make to any panel on the Details Pane take effect. The values that are currently in effect are shown within the rightmost grayed out labels.

Deploying and Configuring Adapters

Specifying paths and filenames in the Details Pane

On several of the component Details Pane panels you are asked to provide a path or filename (for example, to specify where logging information should be stored, or where an adapter configuration file should be loaded from).

It is recommended that *absolute paths* and filenames be used where possible – an absolute path being one that specifies the whole path, for example:

`C:\Program Files\SoftwareAG\Apama 5.3\Logging\New_Correlator_1.log` (on Windows)

or

`/opt/SoftwareAG/apama_5.3/configuration/New_Correlator_1.log`(on UNIX)

In contrast, *relative paths* are incomplete, or just consist of a filename on its own, such as:

```
New_Correlator_1.log
```

Important notes:

- Unless otherwise stated, all paths are located on the file system of the *remote host*, on which the Sentinel Agent and managed components are running, rather than the local host where EMM is running.
- Filenames and paths should never be enclosed in quotes, even if they contain spaces.
- If a relative path is provided – as is the case by default for component log files – it is assumed to be relative to the component's `Alternative start-up directory` setting if one was configured, or to the *current working directory* of the Sentinel Agent on that host if not. See "Working directory" on page 21 for details of how the Sentinel Agent's working directory is determined.

Configuring adapters

The Adapter tabs

When an IAF adapter is selected in the Navigation Pane, the Details Pane will display the following tabs:

Management tab

This tab contains a number of parameters that are relevant to managing an IAF adapter, and is identical to that of a correlator.

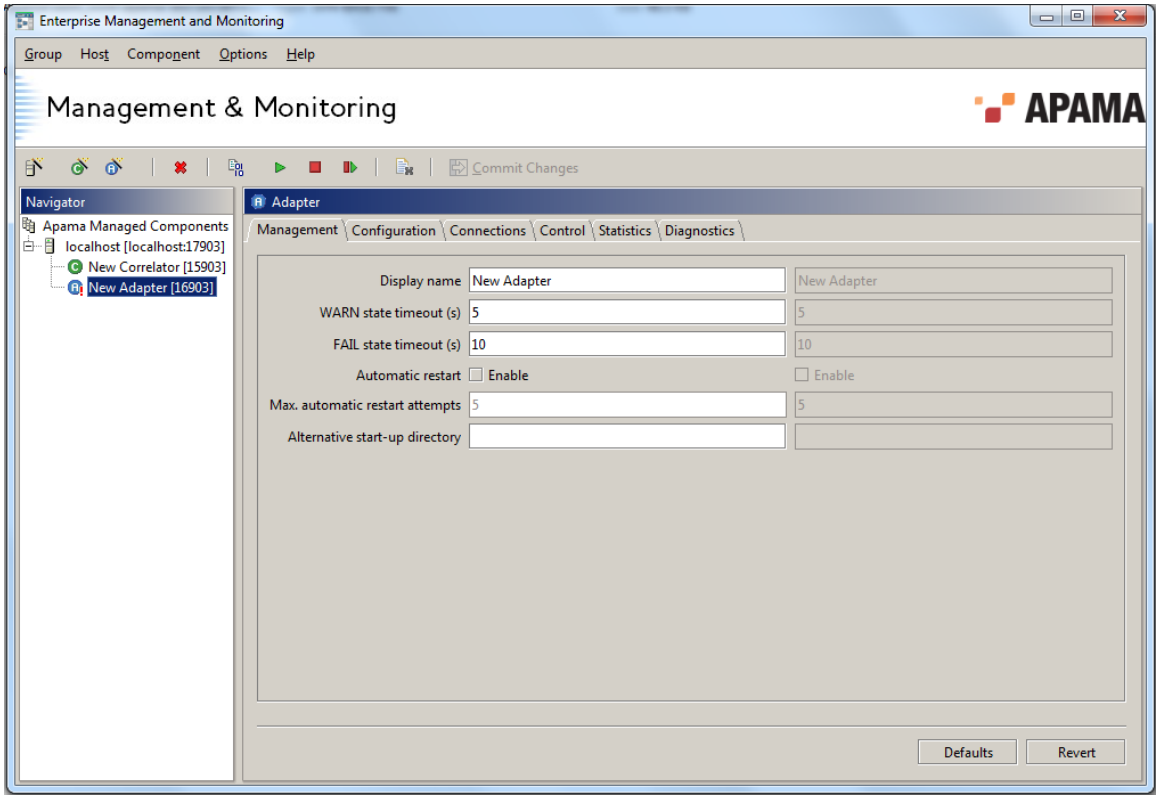
The available parameters are:

- Display name – This is the name to associate with this adapter in the Navigation Pane. The name can be changed at any time, even if the adapter has already been started.
- WARN state timeout (s) – This is the number of seconds to wait before moving the component into the `WARN` state if it is not changing state (stopping/starting/ restarting) as required. This setting can be changed at any time.
- FAIL state timeout (s) – This is the number of seconds to wait after entering the `WARN` state before moving into the `FAIL` state, if the component has still failed to change state as expected. This setting can be changed at any time.
- Automatic restart enable – Tick to configure EMM to monitor the selected component. If it were to stop unexpectedly, EMM would automatically restart it – after waiting the amount of time

required for it to enter the `FAIL` state (and subject to the configured limit on the number of restart attempts described below). This setting can be changed at any time.

Component monitoring and auto-restart is carried out from EMM and requires EMM to be running.

- Max. automatic restart attempts – This option is only available if `Automatic restart` is enabled. It specifies the total number of automatic restarts to perform (or attempt) without user intervention. The count is made from when you last enabled `Automatic restart` and committed, or explicitly stopped/started/ restarted the adapter. This setting can be changed at any time.
- Alternative start-up directory – By default all components are started from the current working directory of the Sentinel Agent running on the host in question (see in "Working directory" on page 21). If you wish, you can provide an alternative startup folder here. Note that this option is only functional if the Sentinel Agent running on the host in question is version 1.1.1 or greater.



Click the Commit Changes button when you are finished customizing the new IAF adapter. This applies the outstanding changes.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

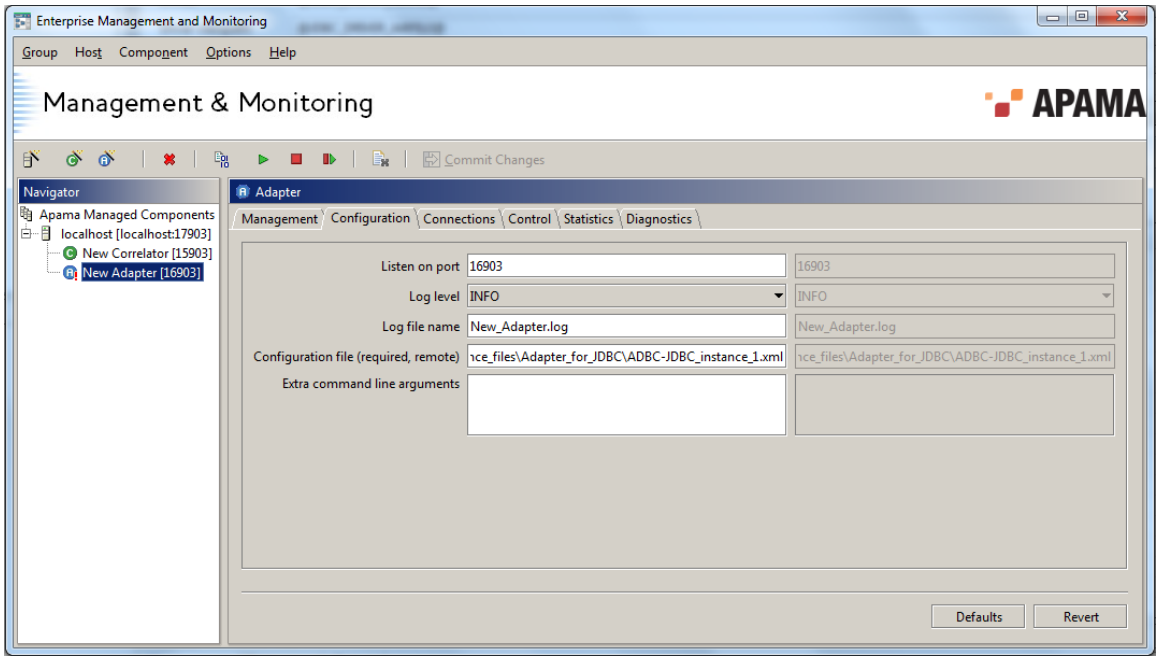
Configuration tab

This tab contains a number of parameters that configure how an IAF adapter operates. As these are all startup parameters, you should adjust them before you start the component. If the component is already running they will only have effect the next time it is started.

The following parameters are available for an IAF adapter:

- Listen on port - Port on which the IAF adapter should listen for monitoring and management requests (default is `16903`). The adapter will fail to start if this port is in use by any other component, or for that matter by any other software that is running on the relevant host.
- Log level - Sets the log level the IAF adapter should log at – must be one of `CRIT`, `ERROR`, `WARN`, `INFO`, `DEBUG` (in increasing order of verbosity). If this value is not set, then the IAF adapter will log at the level specified in the adapter configuration file. If a value is indeed specified, that the EMM value will always override the log level specified in the adapter configuration file.
- Log file name - Sets the filename that the adapter should write log messages to (on the file system of the host the correlator is running on). It is recommended that an absolute path be provided.
- Configuration file (required, remote) – Specifies the path of an IAF adapter configuration file, as described in The IAF configuration file. The IAF adapter cannot be committed or started until a configuration file is specified.

  The file's path is looked up on the file system of the host the adapter is running on, and it is recommended that an absolute path be provided. See "Specifying paths and filenames in the Details Pane" on page 61 for more information about setting filename options correctly.
- Extra command line arguments – This option allows additional unspecified command line arguments to be passed to the IAF adapter. For example these might be special settings provided to you by Apama Customer Support to address specific issues.



Click on the Commit Changes button when you are finished customizing the new IAF adapter. This applies these outstanding changes for use when the adapter is actually started.

The Default button resets all outstanding parameter values to their defaults, while the Revert button reverts the outstanding parameter values to their current committed values.

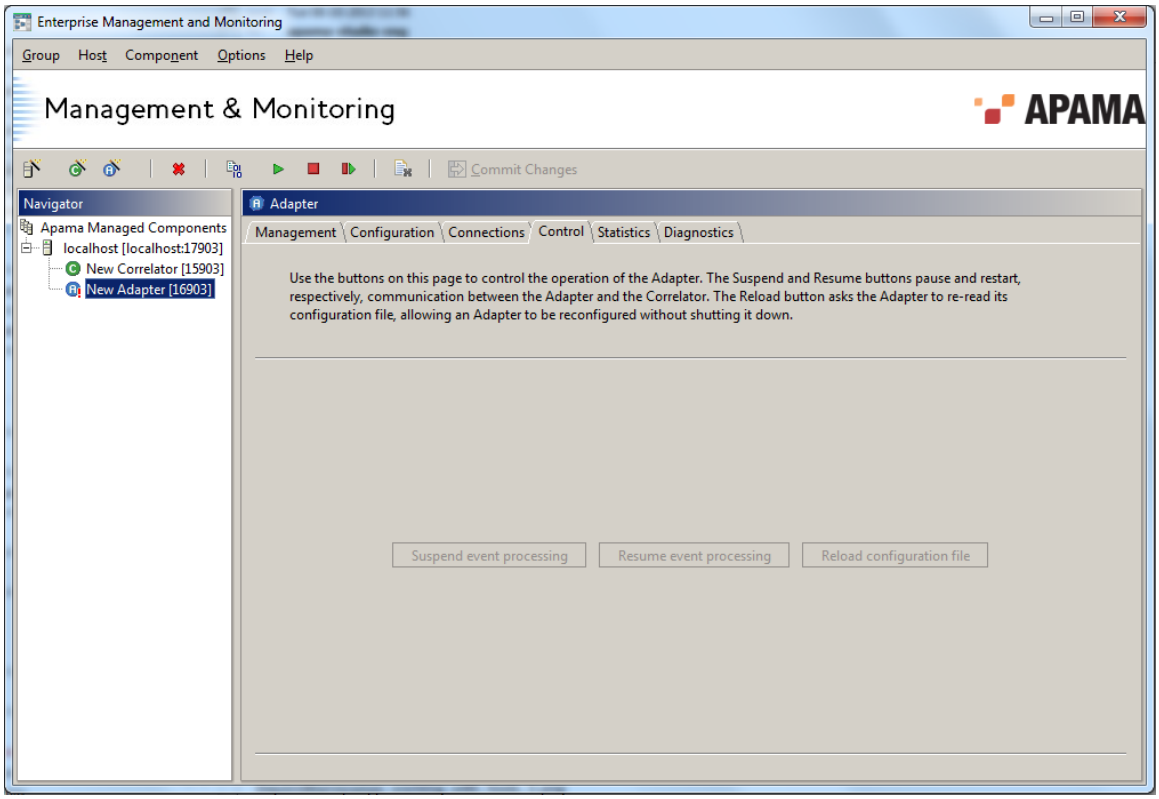
Connections tab

This tab allows you to connect an adapter to upstream components. It also displays downstream components that are connected to the adapter. The tab is similar to the Connections tab for a correlator. For a complete description of the Connections tab, "Connections tab" on page 49.

Control tab

The Control tab provides for control of an IAF adapter. Three operations are supported:

- Suspend event processing – This button will pause a running IAF adapter.
- Resume event processing – This button will resume a running IAF adapter.
- Reload configuration file – This button will ask the IAF adapter to re-read its configuration file, allowing an adapter to be reconfigured without shutting it down.



Statistics tab

This tab allows monitoring of the status of an operational IAF adapter.

Status information is refreshed regularly, by default, once a second. You can change this rate by changing the Component status display update interval (s) value on the Timing tab within the Preferences dialog. The Preferences dialog is available by selecting Options > Preferences… from the EMM menu.

No status information will be displayed if EMM cannot communicate with the IAF adapter selected. This is usually because the adapter has not yet been started or has been stopped. However, it could also be due to network failure. In this case, statistics will reappear once the connection is restored.

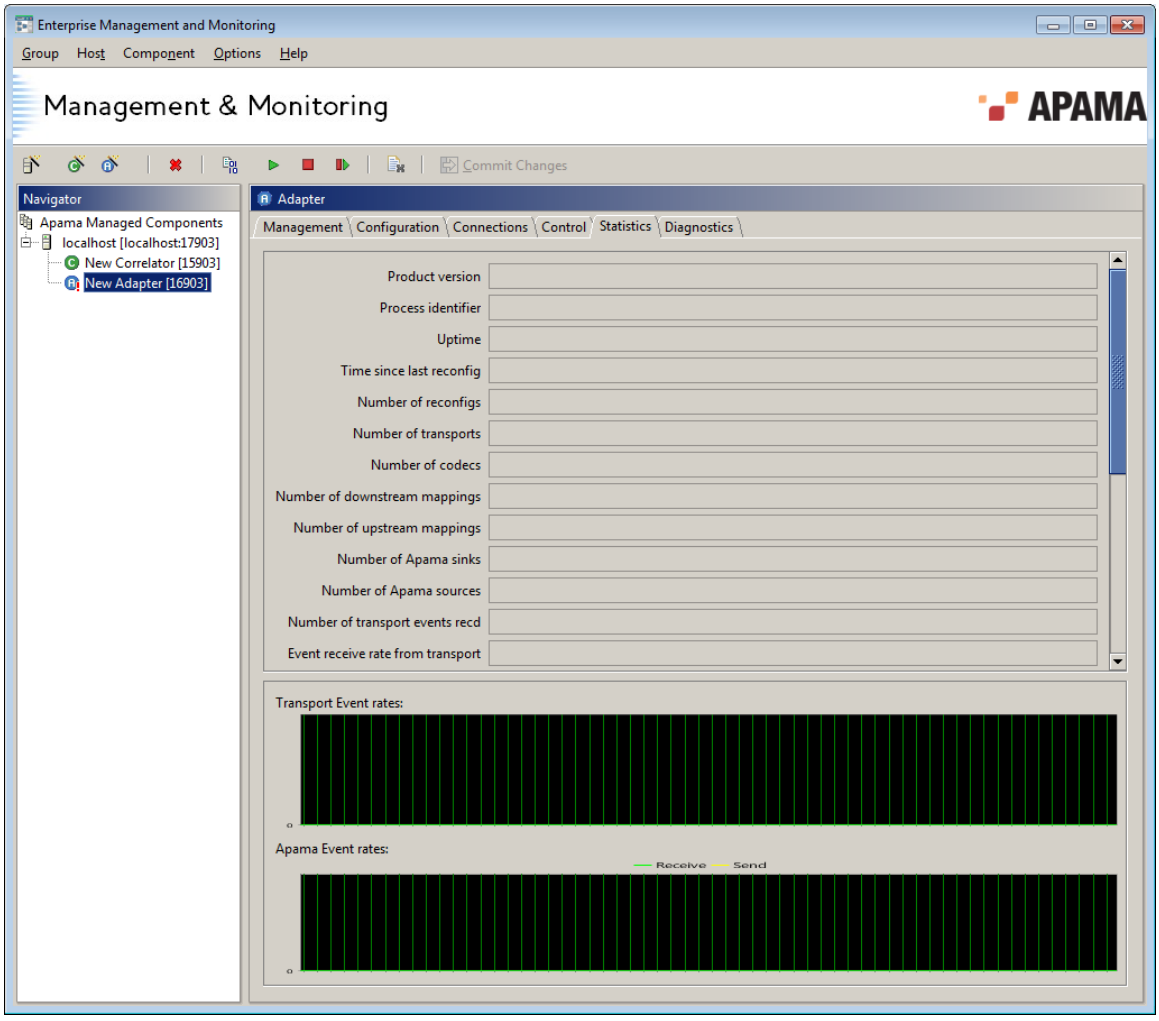
The following statistics are available:

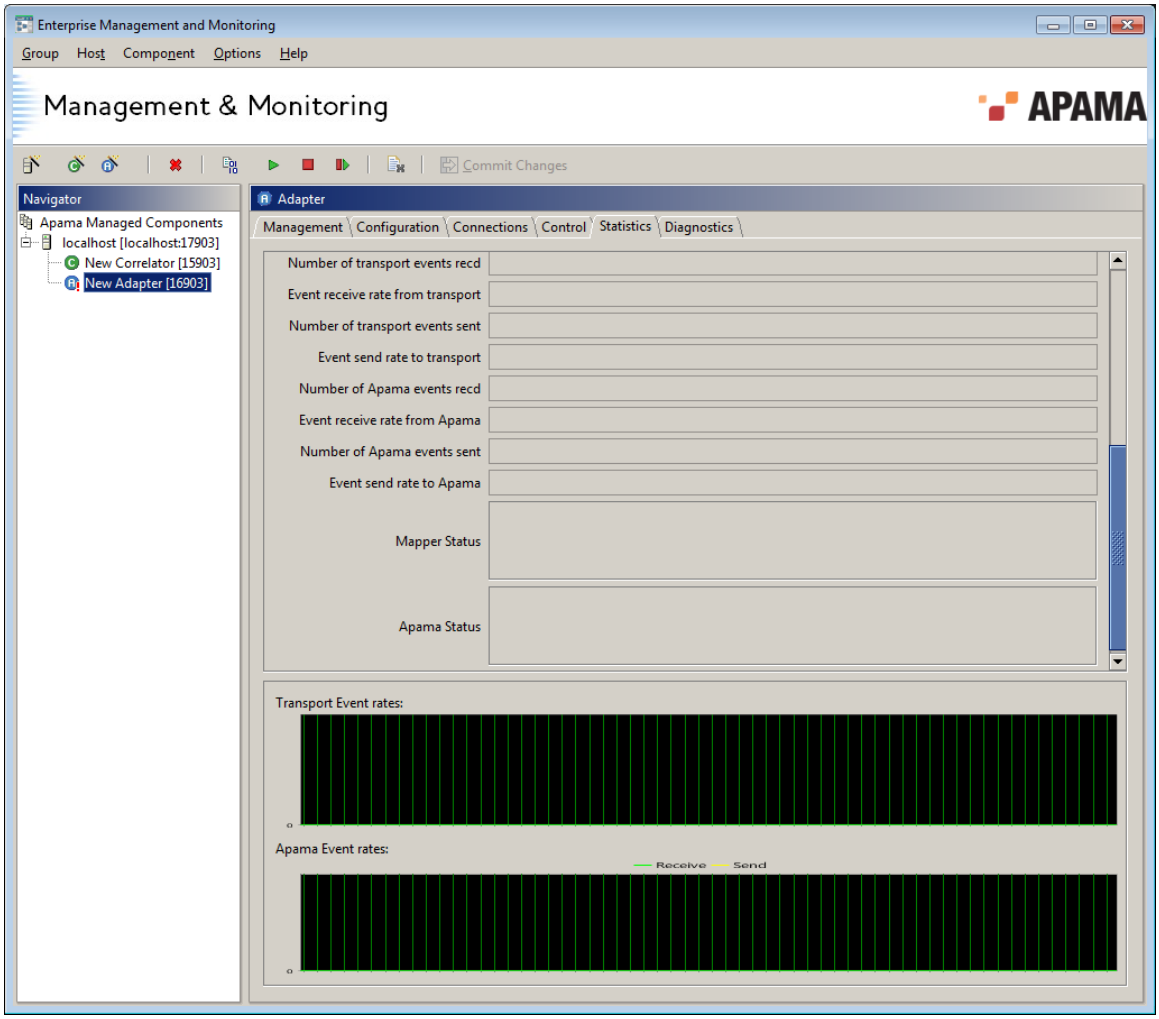
- `Product version` – The Apama release number.
- `Process identifier` – The identifier assigned to this component by the operating system.
- `Uptime` – The time in seconds since this adapter was started. This time is maintained and reported by the component itself, so if the adapter was started independently of EMM and only managed by EMM later, the value would still be accurate.
- `Time since last reconfig` – The time in milliseconds since this adapter was last 'reconfigured', that is, reset with a new configuration file.
- `Number of reconfigs` – The total number of reconfigurations.
- `Number of transports` – The total number of Transport Plug-ins active in this adapter.
- `Number of codecs` – The total number of Codec Plug-ins active in this adapter.
- `Number of downstream mappings` – The total number of mapping rules set up in the adapter's semantic mapper for downstream (from an external message source into Apama events) event mappings.
- `Number of upstream mappings` – The total number of mapping rules set up in the adapter's semantic mapper for upstream (from Apama events into an external message sink) event mappings.
- `Number of Apama sinks` – The number of Apama correlators that the adapter is sending events to.
- `Number of Apama sources` – The number of Apama correlators that the adapter is receiving events from.
- `Number of transport events recd` – The total number of messages received by the adapter's transport Plug-ins from external message sources (that is, downstream).
- `Event receive rate from transport` – The number of messages per second currently being received by the adapter's transport plug-ins. This value is computed with every status refresh and is only an approximation.
- `Number of transport events sent` – The total number of messages sent out by the adapter's Transport Plug-ins to external message sinks (i.e. upstream).
- `Event send rate to transport` – The number of messages per second currently being sent out by the adapter's Transport Plug-ins. This value is computed with every status refresh and is only an approximation.
- `Number of Apama events recd` – The total number of Apama events received by the adapter from Apama for upstream conversion.
- `Event receive rate from Apama` – The number of Apama events per second currently being received by the adapter from Apama. This value is computed with every status refresh and is only an approximation.

- `Number of Apama events sent` – The total number of Apama events sent to Apama by the adapter because of a downstream conversion.
- `Event send rate to Apama` – The number of Apama events per second currently being sent to Apama by the adapter. This value is computed with every status refresh and is only an approximation.
- `Mapper Status` – Indicates whether the adapter's semantic mapper has started.
- `Apama Status` – Indicates whether the adapter has contacted the Apama sinks and sources specified in the Configuration File.

The four event rate statistics are also displayed graphically over the last 60 seconds at the bottom of the Statistics tab in the two graphs: `Transport Event rates` and `Apama Event rates`.

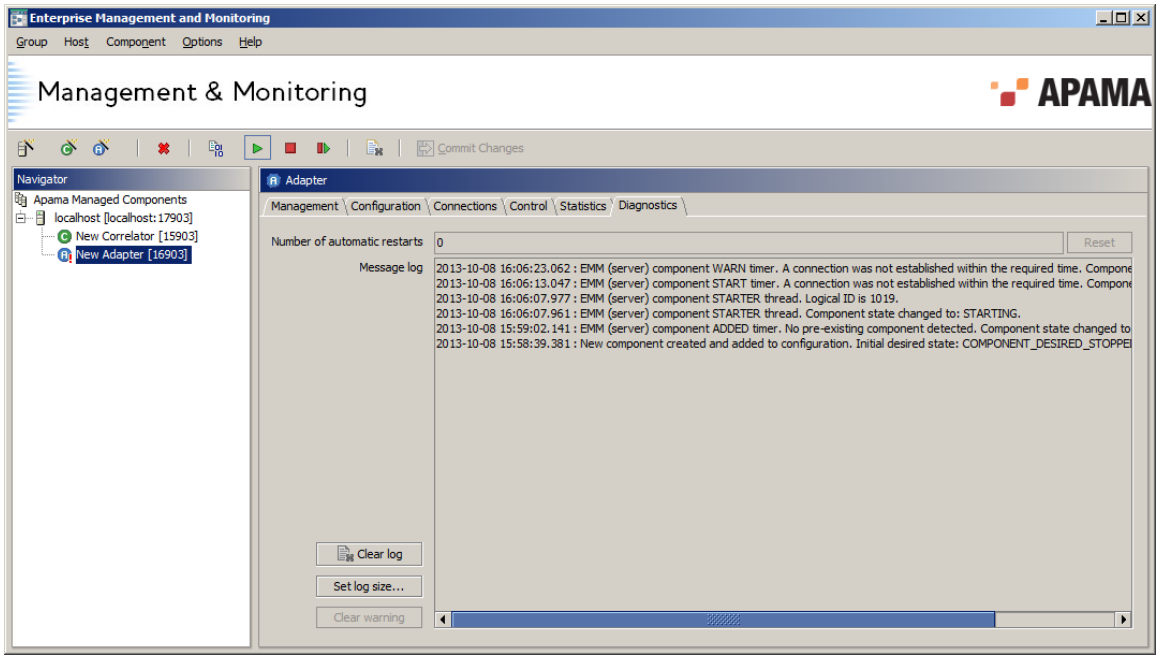
The following two illustrations show the top and bottom portions of the Statistics tab for adapters.





Diagnostics tab

The Diagnostics tab for an IAF adapter is identical to that of a correlator and displays diagnostics information about changes in the state of the adapter.



For complete information on this tab, see "Diagnostics tab" on page 58.

Chapter 5: Deploying and Managing Queries

As mentioned elsewhere, scaling, both vertically (same machine) and horizontally (across multiple machines), is inherent in Apama query applications. Scaled deployments on multiple machines use distributed cache technology to maintain and share application state. Consequently, deployment of Apama query applications includes setting up a distributed cache as well as some kind of messaging. The topics in this section provide instructions for doing this with the recommended platforms.

Overview of deploying and managing query applications

Typically, query application deployments script the start up and management of all Apama query application components outside of the Apama Studio development environment. Apama recommends the use of the Ant export facility of Apama Studio to aid in this.

Queries can also be run from Apama Studio. However, Apama Studio can run only a single correlator deployment. Use Apama macros for Ant to deploy multiple correlator deployments.

Queries can be deployed on a single node, but typically would be deployed across multiple nodes, forming a cluster. While involving more components, a cluster provides:

- Scale out across multiple hosts
- Resiliency against failures
- Continued availability if some nodes fail

Using a cluster will involve the following:

- Some number of correlators that are executing queries
- A BigMemory distributed cache for storing event history
- A JMS bus for distributing events to correlators.

There are likely to also be adapters or a message bus to deliver events generated by queries, and possibly adapters connected to each correlator. See "Deploying and Configuring Adapters" on page 60.

Deploying and Managing Queries

Query application architecture

In a query deployment, incoming events are delivered to correlators, typically via a JMS message bus, such that every event is delivered to one correlator. The correlators store the event history for each query in BigMemory, a distributed cache. On every event, one correlator reads the latest history for the partition or partitions to which the event belongs, and writes that event to the distributed cache for access by other correlators. The entire window history is then evaluated against the query patterns.

Queries can make use of the following technologies to provide a scalable platform:

- JMS queues — these are used to distribute events to multiple correlators, which automatically spreads the load across a number of servers.
- BigMemory distributed cache — this allows state (event history) to be accessed quickly across multiple servers, and replicated to safeguard against hardware failures. This should be configured to give the desired amount of resiliency and scaled appropriately to the deployment.

It is possible to use Apama queries in a standalone mode on a single correlator. This allows easy testing by means of event files. However, all state is stored in-memory, and is lost when the correlator is stopped. Thus, this mode is only recommended for development, not for deployments.

When an event is sent to a cluster of correlators over a JMS queue, this is what happens:

1. Each event goes to one correlator.
2. A received event is handled by one of several processing threads within that correlator.
3. The key of the event is extracted based on the definitions of running queries that use that event.
4. The window of events for that key value is retrieved from the distributed cache.
5. The current event is added to the retrieved window, which is written back to the cache.
6. The event pattern of interest (what you are looking for) is evaluated against the stored window to determine whether there is a match.

Because events are sent to multiple threads in different correlators, small differences in timing across hosts can result in events being processed out of order. If there are a large number of events in the window, the cost of reading and writing the historic window will be excessive. Events for the same key may be processed by different correlators. Consequently, between events, the only state kept by the system is the window of historic event data.

Upon matching an event pattern, queries may send events to other monitors or to adapters. These can be shared adapters across the cluster, or more typically, adapters local to each correlator.

Deploying and Managing Queries

Deploying query applications

Apama recommends that you use the Ant export facility in Apama Studio to help you deploy your query application. The general steps for deploying an Apama query application include:

1. In Apama Studio, configure JMS bus access and distributed cache access. See "Correlator arguments" in *Using the Apama Studio Development Environment*.

2. In Apama Studio, generate an Ant deployment script. Apama Studio places the files it generates in a directory that you specify. See "Exporting to a deployment script" in *Using the Apama Studio Development Environment*.
3. Copy the resultant directory onto each host that will run a correlator.
4. If necessary, edit the `environment.properties` file on each correlator host.
5. Ensure that the BigMemory and JMS servers are running.
6. On each correlator host, run the Ant deployment script to start the correlator.

If the project does not contain a distributed cache configuration, a local in-process MemoryStore will be used to store events. This is not shared or persistent, so only supports a single correlator deployment. If this correlator stops, it will drop all event history data. Apama recommends a BigMemory server and configuration for production use. See "Deploying BigMemory Terracotta Server Array" on page 72 and "Configuring BigMemory driver" on page 73.

Apama does not recommend running multiple correlators on a single machine. The assumption is that each correlator can use all of the CPU resources available. Also, running multiple correlators on one host does not provide any extra resilience. However, it is possible to run multiple correlators on a single machine. To do so:

1. Copy the Apama Studio exported deployment directory to separate directories on the correlator host machine.
2. Edit the `environment.properties` file to specify a different port number for each correlator and for each (if any) adapter in your project.

Deploying and Managing Queries

Running queries on correlator clusters

The following topics describe how to run queries on correlator clusters.

Deploying and Managing Queries

Deploying queries on multiple correlators

When using multiple correlators to deploy an Apama query application, it is the administrator's responsibility to keep the resources of the exported project up to date. If changes are made to a query, if queries are added or removed from a project, then all correlators should be updated to reflect the new state. It is possible to inject queries into a live running correlator, or delete queries from a correlator. Make sure that the injections and deletions are performed on all correlators in the cluster.

Running queries on correlator clusters

Deploying BigMemory Terracotta Server Array

To deploy BigMemory, see the BigMemory Max documentation.

For resilient operations, Apama recommends at least one backup on a separate host. You may want to consider using multiple stripes in order to improve performance. Ensure that the BigMemory server is accessible from all cluster members.

Running queries on correlator clusters

Configuring BigMemory driver

To configure the BigMemory driver:

1. In Apama Studio, add the Distributed MemoryStore adapter bundle to your project. See "Adding adapters to projects" in *Using the Apama Studio Development Environment*.

   In the Distributed MemoryStore Configuration Wizard, specify `ApamaQueriesStore` as the store name.

2. Check that the cluster name is set correctly for the host/port pairs of all of the BigMemory Terracotta Server Array.
3. Set the providerDir property to the Terracotta installation directory.
4. Optionally, edit the on-heap and off-heap storage and other parameters as needed (see "BigMemory Max driver specific details" in *Using the Apama Studio Development Environment*. The `useCompareAndSwap` property should be left in its default `true` configuration for correct behavior of Apama queries.

Running queries on correlator clusters

Using JMS to deliver events to queries running on a cluster

When running queries across multiple correlators in a cluster, as well as configuring all correlators to access the same BigMemory store, Apama recommends that all events are delivered into the cluster using a JMS queue. By using a JMS queue, each correlator will pull events from the JMS queue unless it has a full input queue (that is, it is behind on processing events) or has stopped running (e.g. shut down for maintenance or suffered a hardware failure). In either case, events will continue to be processed by other correlators in the cluster. Correlators can also be added to or removed from the cluster to scale the cluster capacity if desired. It is also possible to use per-correlator adapters for incoming events, but the adapters must co-ordinate so that every event is sent to only one correlator, and should one adapter/correlator pair fail, then other adapters process events that the failed node would have processed. Each event should only be delivered to one correlator, else multiple correlators will store the event in the shared cache, which can result in erroneous matches. Using JMS queues, this happens automatically, giving an 'elastic' system that can be scaled and continues running in the face of failure.

To run queries across multiple correlators in a cluster:

- Configure each correlator to access the same BigMemory store. This is a requirement.
- Use a JMS queue to deliver events into the cluster. This is a recommendation.

When the cluster uses a JMS queue, each correlator pulls events from the queue. If the input queue of one correlator in the cluster becomes full and it cannot pull events from the JMS queue the other correlators continue to do so and continue to process events. A correlator may stop pulling events because the correlator is behind on processing events or because it has stopped running, perhaps for maintenance or because of a hardware failure.

Using a JMS queue makes it easy to scale the cluster capacity by adding or removing correlators.

An alternative to using a JMS queue is to use an adapter for each correlator. For example, by having an IAF-based adapter connected to each correlator, it is possible to send messages to and from a query application without using JMS. A disadvantage of using per-correlator adapters is that the adapters must coordinate the following:

- Each event goes to only one correlator in the cluster. If an event goes to more than one correlator then multiple correlators store the same event in the shared cache. This can result in erroneous matches.
- Should one adapter/correlator pair fail then the other adapters process the events that the failed node would have processed.

Use of a JMS queue automatically ensures that an event goes to only one correlator and that all received events are processed. The result is an 'elastic' system that can be scaled and that continues to run even if a node fails.

Similar to using multiple contexts in a correlator, delivering events through JMS can result in events that occur close together in time being processed in an order that is different than the order in which they were created or sent to the JMS message bus. Queries do not support reliable message delivery. Consequently, if a correlator fails, perhaps because of a hardware failure, all events that had been received by the failed correlator but not yet written to BigMemory may be lost, and output events from queries can be lost. However, if the system is correctly scaled then the number of messages lost will be small.

Configure your JMS bus to have one or more queues, and configure a static JMS receiver connection. See "Getting started with simple correlator-integrated messaging for JMS" in *Connecting Apama Applications to External Components*. You will also need to provide mapping for all event types that flow into the queries. See "Mapping Apama events and JMS messages" also in *Connecting Apama Applications to External Components*.

The queries runtime ensures that after all queries have been injected into the correlator and started, they automatically start to receive events from JMS queues. There is no need to explicitly call `jms.onApplicationInitialized()` as described in "Using EPL to send and receive JMS messages" in *Connecting Apama Applications to External Components*.

For all applications that do not consist entirely of queries, for example, applications that contain additional EPL monitors or Java monitors, then it may be required to delay starting JMS until the application and queries are both ready to process events. The auto-starting of JMS behaviour of queries can be controlled by sending a `QueriesShouldNotAutoStartJMS()` event to the main context. This event can be routed by an application's `onload()` method. If this is done then a monitor in the main context should listen for a `QueriesStarted()` event and should wait until both the application and queries have started. The monitor can then call `jms.onApplicationInitialized()` directly. For example, the following monitor delays starting JMS until queries are started and a `StartMyApp()` event has been processed:

```
using com.apama.queries.QueriesShouldNotAutoStartJMS;
using com.apama.queries.QueriesStarted;
event StartMyApp {
}
```

```
monitor MyApp {
   import "JMSPlugin" as jms;
   action onload {
      route QueriesShouldNotAutoStartJMS();
      on QueriesStarted() and StartMyApp() {
         jms.onApplicationInitialized();
      }
   }
}
```

Running queries on correlator clusters

Mixing queries with monitors and scenarios

It is possible to have both monitors and queries in a project.

Events that are to be processed by queries should be sent to the `com.apama.queries` channel from monitors. Queries may send events to any channel, which EPL monitors may be subscribed to.

While queries will automatically scale and share state across a cluster, EPL monitors will not. Thus, be aware that a query may process subsequent events matching a pattern on different nodes. On different nodes, monitors with potentially different state will be executing. Similarly, the state of EPL monitors is not automatically stored in the distributed cache.

Both EPL monitors and Apama queries can make use of actions defined on events, subject to some limitations on the use of `spawn`, `die`, and event listeners. See "Restrictions in queries" in *Developing Apama Applications*.

Running queries on correlator clusters

Handling node failure and failover

A node may stop processing events from time to time. This may be because it is stopped for planned maintenance, or the node failed in some way. In these cases:

- Events that have been delivered to the node but not yet processed will be lost. This will typically be a small window of events.
- Patterns that end with a `wait` operator may be dropped if they are being handled by a node that has stopped. For example, if a query contains the pattern `MyEvent:e -> wait(5.0)`, and the node that receives a matching `MyEvent` stops before 5 seconds are up, then the pattern will not be matched in this instance.
- If using JMS, then events continue to be delivered to and processed by other correlators in the cluster. The failed correlator will not hold up processing on other nodes. Other nodes continue processing events, including matching against events that the failed node had previously received (if they had been processed).
- Any clients connected to the failed correlator will need to re-connect to another correlator. The same set of parameterized query instances is kept in synchronization across the cluster. See "Managing parameterized query instances" on page 76.

Similarly, nodes running a BigMemory Terracotta Server Array may fail. For this reason, BigMemory should be configured with sufficient backups to ensure no data is lost in this case. Consult the Terracotta documentation.

Running queries on correlator clusters

Managing parameterized query instances

When using parameterized queries, Apama recommends that you use one Scenario Service client at a time to manage parameterizations. Use of more than one client can result in undefined behavior if they both attempt to edit a parameterized instance concurrently. You can connect to any correlator in the cluster, and Apama will automatically synchronize the state of parameterized instances across the cluster. This assumes that the same query definitions have been injected into the correlators on all cluster nodes. If a node fails, you will need to connect to another correlator in the cluster.

Deploying and Managing Queries

Creating new query instances by setting parameter values

Use Scenario Browser to set parameter values for a parameterized query and thus create new parameterized query instances, also referred to as parameterizations.

Managing parameterized query instances

Changing parameter values for queries that are running

Use Scenario Browser to change the parameter values for a running parameterized query instance, also referred to as a parameterization.

Managing parameterized query instances

Monitoring running queries

To help you monitor queries that are running on a given correlator, Apama provides data about active queries in DataViews. To display the information provided by these DataViews, you can create a dashboard in which an end user can:

- Monitor query runtime performance.
- Determine whether a query is behaving as intended. For example, you can see how incoming events are distributed across partitions. If you are expecting a particular send and match rate you can see if you are getting the results you expect.
- Ensure that the window size (the number of events in the window) is not too large. The expectation is that your application is designed so that partitioning keeps any given window size as small as possible.

For information about exposing DataViews in dashboards, see *Building and Using Dashboards*.

A running query is either a non-parameterized query instance or a parameterization. For each running query, there is a DataView for each of its input event types. For example, if a query instance has two input event types then there are two DataViews that provide statistics for that query, one for each input event type.

Each DataView:

- Contains data about the activity during the last second of one running query and one of its input event types.
- Contains the fields described in the table below. The value contained in each field is an exponentially weighted moving average (EWMA).
- Is updated every second if the information has changed in the last second.

Statistical Field	Description
`% Threads Active EWMA`	Apama uses multiple threads to process a given query. This is the percentage of those threads that were used within the last second to process the input event type that this DataView provides information for. While there is not a linear correlation, as this percentage goes down, the reliability of the rest of the statistics becomes weaker. This is because a smaller proportion of threads are contributing information.
`Avg. Inbound Event Rate/s EWMA`	The average rate per second at which events of this type are being processed.
`Avg. % of Successful Matches EWMA`	The average percentage of the number of received events that cause a match.
`No. Unique Keys Processed EWMA`	The number of unique query partitions that were accessed for this event type within the past second.
`Avg. Window Size/Key EWMA`	The average window size (number of events that it contains) of each unique partition that was accessed within the past second.

The display name of these DataViews is Correlator Query Statistics.

After a non-parameterized query is injected into the correlator, Apama provides a DataView for each input event type and begins writing data to it. After a non-parameterized query is deleted, Apama no longer makes the DataViews for that query instance available.

For a parameterized query, after a parameterization is created then Apama adds new DataViews and begins populating them. When a parameterization is deleted then Apama no longer provides the DataViews that correspond to that parameterization. If the definition of a parameterized query is deleted then Apama no longer provides DataViews for any parameterizations of that query.

To help you monitor queries that are running across multiple correlators in a cluster, Apama also provides the same type of performance statistics provided for a given correlator but where the underlying data has been aggregated across all the clustered correlators running those queries.

The display name of these DataViews is Cluster-Wide Query Statistics.

This means that for each query running on a correlator two types of monitoring data is provided:

- Statistics generated from data from only that correlator.
- Statistics generated from data aggregated across all correlators in the cluster running that same query.

Deploying and Managing Queries

Chapter 6: Tuning Correlator Performance

This section addresses how to scale up Apama to improve upon the performance of a single event correlator. It describes the Apama features you can use to send events to multiple event correlators to increase an application's capacity.

Scaling up Apama

Apama provides services for real-time matching on streams of events against hundreds of different applications concurrently. This level of capacity is made possible by the advanced matching algorithms developed for Apama's event correlator component and the scalability features of the correlator platform.

Should it prove necessary, capacity can further be increased by using multiple event correlators on multiple hosts. To facilitate such multi-process deployments, Apama provides features to enable connecting components to pass events between them. It is recommended that each correlator is run on a separate host, to assist in the configuration of scaled-up topologies. However, it is possible to run multiple correlators on a single host. There are two methods of configuration:

- Using the configuration tools from the command line or Apama macros for Ant.
- Programmatically through a client programming API.

This guide describes both approaches, but first discusses different ways in which Apama can be distributed and what factors affect the choice of the distribution strategy.

Note: This topic focuses on scaling Apama for applications written in EPL. JMon has less scaling features as it does not support the use of multiple contexts. Java plug-ins can be used if invocation of Java code is required on multiple threads, either directly from EPL or by registering an event handler. See "Using Java plug-ins" in the "Developing Correlator Plug-ins" part of *Developing Apama Applications*. Knowledge of aspects of EPL is assumed, specifically monitors, spawning, listeners and channels. Definitions of these terms can be found in "Getting Started with Apama EPL" in *Developing Apama Applications*.

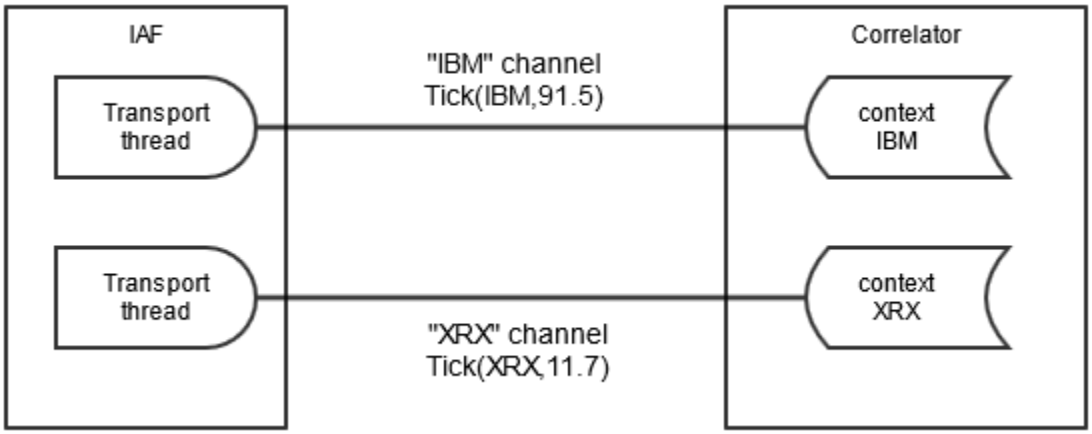
The core event processing and matching service offered by Apama is provided by one or more event correlator processes. In a simple deployment, Apama comprises a single event correlator connected directly to at least one input event feed and output event receiver. Although this arrangement is suitable for a wide variety of applications (the actual size depending on the hardware in use, networking, and other available resources), for some high-end applications it may be necessary to scale up Apama by deploying multiple event correlator processes on multiple hosts to partition the workload across several machines.

Partitioning strategies

Using the patterns and tools described in this guide it is possible to configure the arrangement of multiple contexts within a single correlator or multiple event correlators within Apama (the engine topology). It is important to understand that the appropriate engine topology for an application is firmly dependent on the partitioning strategy to be employed. In turn, the partitioning strategy is determined by the nature of the application itself, in terms of the event rate that must be supported, the number of contexts, spawned monitors expected and the inter-dependencies between monitors and events. The following examples illustrate this.

The `stockwatch` sample application (in the `samples\monitorscript` folder of your Apama installation directory) monitors for changes in the values of named stocks and emits an event should a stock of interest fall below a certain value. The stocks to watch for and the prices on which to notify are set up by initialization events, which cause monitors that contain the relevant details to be spawned. In this example, the need for partitioning arises from a very high event rate (perhaps hundreds of thousands of stock ticks per second), which is too high a rate for a single context to serially process.

A suitable partitioning scheme here might be to split the event stream in the adapter, such that different event streams are sent on different channels. The illustration below shows how this can be accomplished:



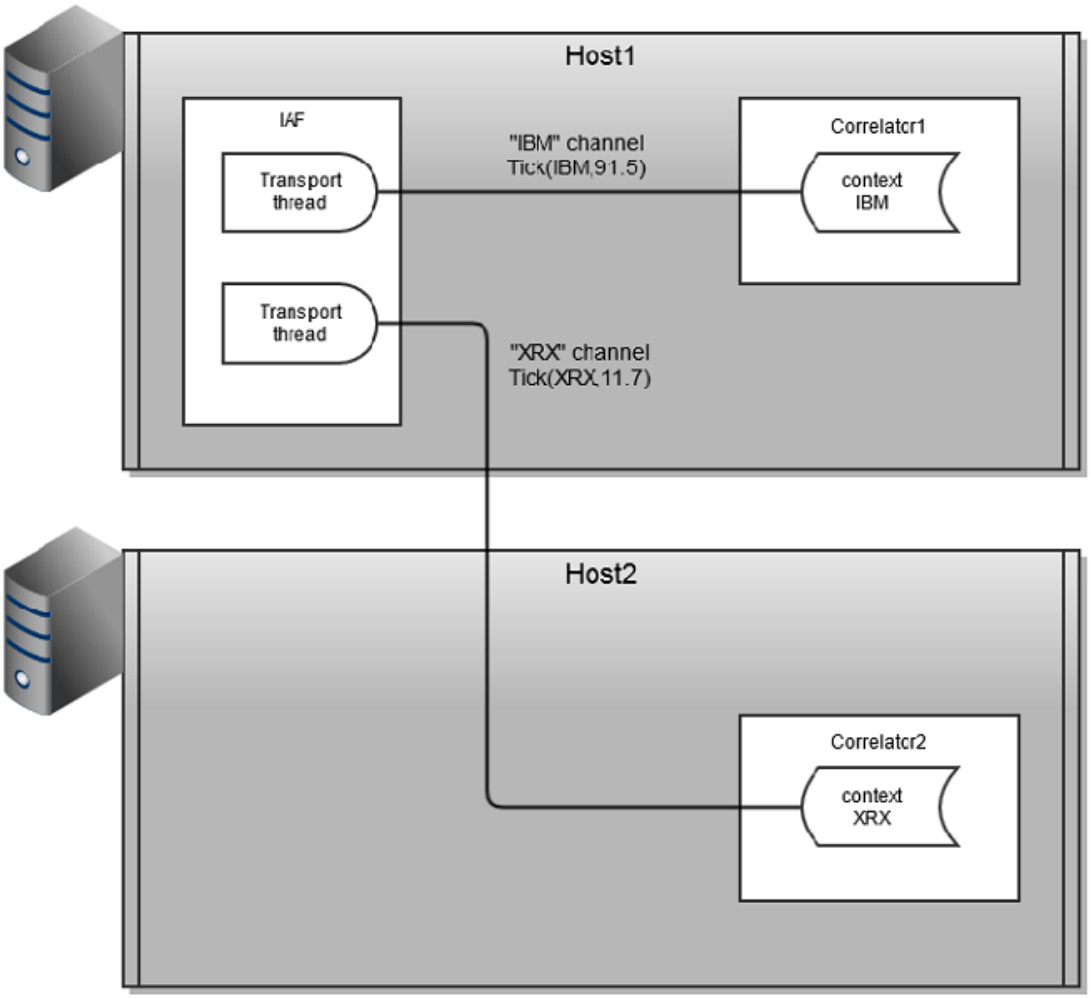
This diagram shows an adapter sending events to different channels based on the symbol of the stock tick. The adapter transport configuration file would specify a `transportChannel` attribute for the stock event that named a field in the `NormalisedEvent` that specified the stock symbol. Either a thread per symbol or a single thread (which could become a bottleneck) managed by the transport, depending on what the system the transport is connecting to allows, is used to send `NormalisedEvent`s to the semantic mapper to be processed. The IAF thus sends the events on the channel in the stock symbol value in the `NormalisedEvent`.

In this example, the stock symbol is either `IBM` or `XRX`. The IAF will send events to all sinks (typically one) that are specified in the IAF's configuration file. In the correlator, all monitors interested in events for a given symbol would need to set up listeners in a context where a monitor has subscribed to that symbol. To achieve good scaling, the application is arranged so that each context is subscribed to only one symbol. For the `stockwatch` application, a separate context per symbol would be created, and the `stockwatch` monitor spawns a new monitor instance to each context. In each context, the monitor instance would execute `monitor.subscribe(stockSymbol);` where `stockSymbol` would have the

value `"IBM"` or `"XRX"` corresponding to the stock symbol it is interested in. This application will scale well, as each event stream (for the different stock symbols) can run in parallel on the same host; this is referred to as scale-up.

Listeners in each context would listen for events matching a pattern, such as `on all Tick(symbol="IBM", price < 90.0)`.

If the number of stock symbols is very large and the amount of processing for each stock symbol is large, then it may be required to run correlators on more than one host to use more hardware resources than are available in a single machine. This is referred to scale-out. To achieve scale-out, connections per channel need to be made between the Apama components using the `engine_connect` tool (or the equivalent call from Ant macros or the client API). The `engine_connect` utility can connect any two Apama components, either correlator to correlator, or IAF to correlator. For best scaling, multiple connections are required between components, which `engine_connect` provides in the parallel mode. The following image shows a scaled out configuration.

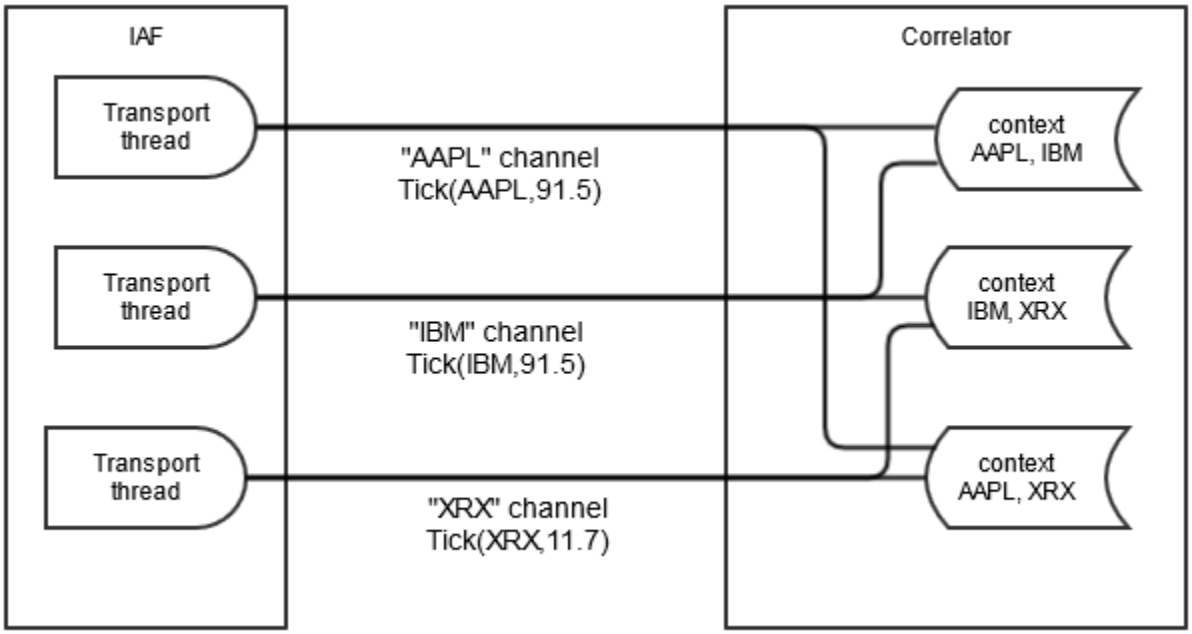


This configuration allows many contexts to run on two hosts and requires use of `engine_connect` to set up the topology.

Now consider a portfolio monitoring application that monitors portfolios of stocks, emitting an event whenever the value of a portfolio (calculated as the sum of stock price * volume held) changes by a percentage. A single spawned monitor manages each portfolio and any stock can be added to/ removed from a portfolio at any time by sending suitable events.

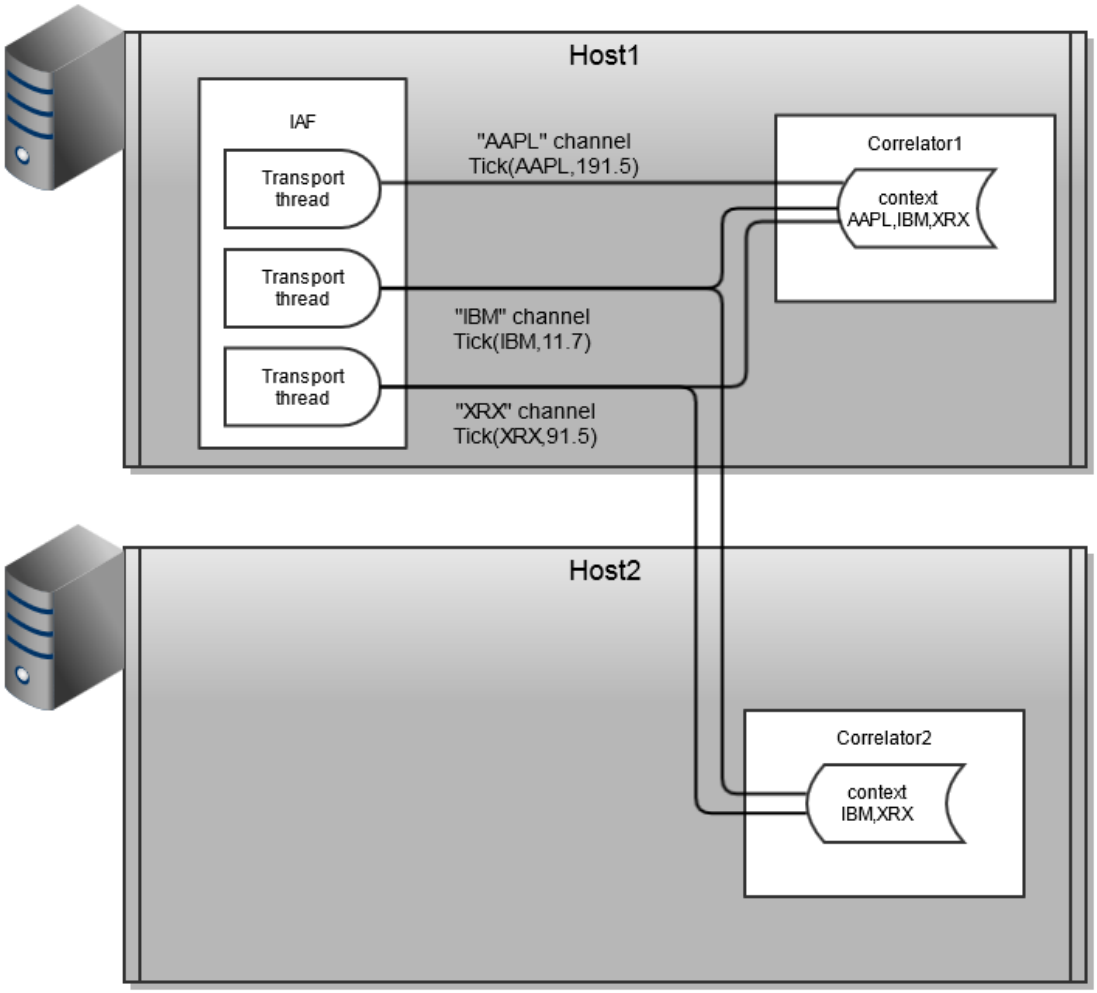
This application potentially calls for significant processing with each stock tick, as values of all portfolios containing that stock must be re-calculated. If the number of portfolios being monitored grows very large, it may not be possible for a single context to perform the necessary recalculations for each stock tick, thus requiring the application to be partitioned across multiple contexts.

Unlike the stockwatch application, it is not possible to achieve scaling to larger numbers of portfolios by splitting the event stream. Each portfolio can contain multiple stocks, and stocks can be dynamically added and removed, thus one event may be required by multiple contexts. In this case, a suitable partitioning scheme is to partition the monitor instances across contexts (as with `stockwatch`) but to duplicate as well as split the event stream to each event correlator. The following images shows the partitioning strategy for the portfolio monitoring application.



Again, each monitor instance is spawned to a new context and subscribes to the channels (stock symbols in this application) that it requires data for. Note that while the previous example would scale very well, this will not scale as well. In particular, if one monitor instance requires data from all or the majority of the channels, then it can become a bottleneck. However, there may be multiple such monitor instances running in parallel if they are running in separate contexts.

Similar to the `stockwatch` application, the portfolio monitoring application may require scale-out across multiple hosts, as shown below.

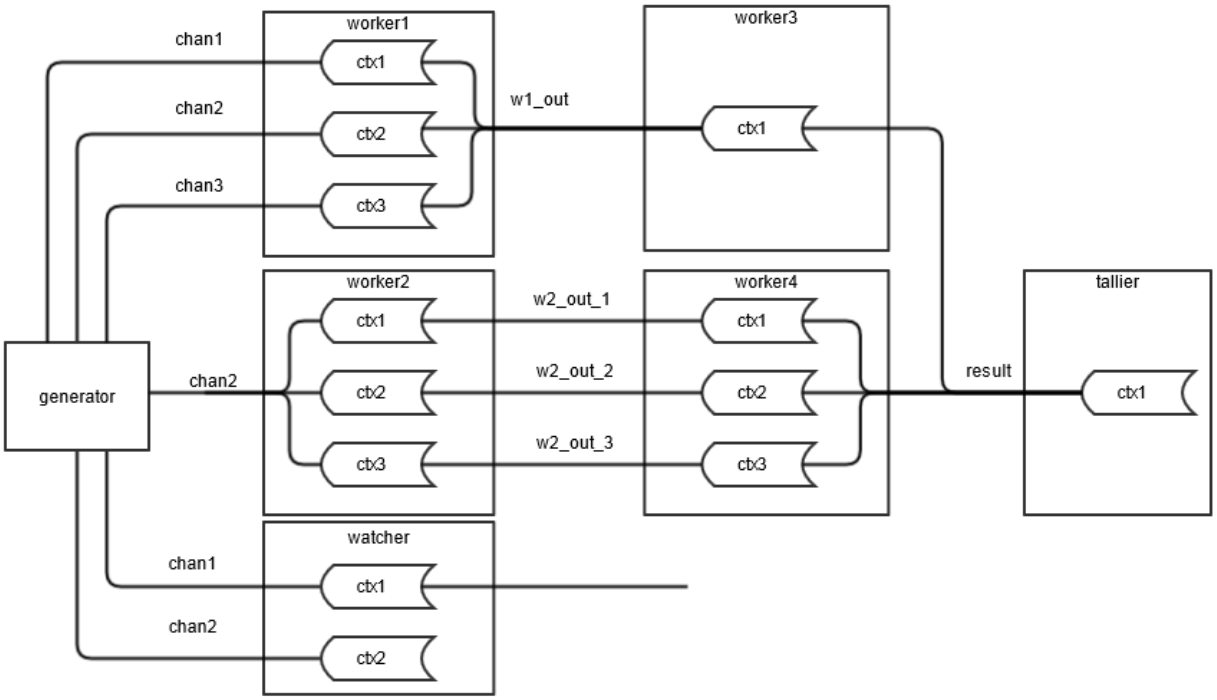


In summary, the partitioning strategy can be thought of as a formula for splitting and duplicating monitors and/or events between event correlators while preserving the correct behavior of the application. In some circumstances, it may be necessary to re-write monitors that work correctly on a single correlator to allow them to be partitioned across correlators, as the following section describes.

Tuning Correlator Performance

Engine topologies

Once the partitioning strategy has been defined, in terms of which events and monitors go to which correlators, it is necessary to translate this into an engine topology. This is achieved by connecting source and target event correlators on separate channels, such that events sent by a source correlator on a specific channel find their way to the correct contexts in the target correlator. A set of two or more event correlators connected in this way is known as a correlator pipeline, as shown in the following image. This figure represents an example topology for a high-end application – the majority of applications use a single correlator only, or have far simpler topologies.



In this image, an event correlator can perform the function of each of the 7 nodes (generator, worker, watcher, tallier). Each target correlator performs some processing before passing the results to a second worker correlator (`worker3`, `worker4`) in the form of events, sent on the channels as marked on the diagram. `tallier` collates the results from these correlators for forwarding to any registered receivers. A final correlator, `watcher`, monitors the events emitted by `generator` on `chan1` and `chan2` and emits events (possibly containing status information or statistical analysis of the incoming event stream) to any registered receivers.

To deploy an application on a topology like that shown above requires separating the processing performed into a number of self-contained chunks. In the previous figure, it is assumed that the core processing can be serialized into three chunks, with the first two chunks split across two correlators each (`worker1`/`2` and `worker3`/`4` respectively) and the third chunk residing on a single correlator (`tallier`). Intermediate results from each stage of processing are passed to the next stage as sent events, which contexts in the connected correlators receive by subscribing to the appropriate channels.

To realize this application structure requires coding each chunk of processing as one or more separate monitors, which send intermediate results as an event of known type on a pre-determined channel. These monitors can then be loaded onto the appropriate correlator. This may require an existing application that grows beyond the capacity of a single event correlator, to be re-written as a number of (smaller) monitors to allow partitioning of the application processing into separate chunks in the manner described above.

Tuning Correlator Performance

Event correlator pipelining

To implement engine topologies comprising multiple event correlators requires a method of connecting correlators in pipelined configurations. This can be achieved in the following ways:

- Directly using the `engine_connect` tool, see "Configuring pipelining with engine_connect" on page 85.
- Indirectly using Software AG's Universal Messaging (UM) message bus. For complex deployments where parts of the application may be moved between Apama correlators, this is likely to be the best alternative. When using UM each correlator connects to the same UM realm. See .
- Programmatically via the C++ client API, see "Configuring pipelining through the client API" on page 91.
- Using the Apama Studio Development Environment, click Connections on the Components tab of the Run Configurations dialog. See .

Configuring pipelining with engine_connect

The `engine_connect` tool allows direct connection of running correlator instances.

The tool is located in the `bin` folder of the Apama installation, as `engine_connect.exe` (on Windows) or `engine_connect` (on UNIX). To run the tool you either need to ensure that the environment variables described in the *Apama Installation Guide* have been set, or else run the tool from an Apama command prompt.

Synopsis

```
Usage: engine_connect [ options ]
To connect a correlator, as an event receiver, to another correlator.
Where options include:
   -h  | --help                This message
   -sn | --sourcehost <host>   Source engine on <host>
   -sp | --sourceport <port>   Source engine is listening on <port>
   -tn | --targethost <host>   Target engine on <host>
   -tp | --targetport <port>   Target engine is listening on <port>
   -c  | --channel <channel>   Listen on output channel <channel>
   -m  | --mode <mode>         Specify legacy or parallel
   -x  | --disconnect          Destroy rather than create connections
   -s  | --qdisconnect         Disconnect if slow (only takes effect on
                               first connection)
   -f  | --filename <file>     Read config data from a file
   -u  | --utf8                Assume config file is in UTF8
   -v  | --verbose             Be more verbose
   -V  | --version             Print program version info
   The target is connected as an event consumer to the source
   Multiple -c options may be given
   To read from stdin use -f -
```

Description

`engine_connect` connects a source correlator (the sender) to a target correlator (the receiver). The target correlator will receive events from the specified channel(s) of the source correlator. Source and target correlators must already be running.

Alternatively, if you specify the `-f` option, `engine_connect` reads connection information from the specified file and sets up each connection found therein (see "Configuring pipelining through the client API" on page 91 for details of the file format). The `engine_connect` utility expects the specified file to be in the local character set. If the configuration file is in UTF-8, specify the `-u` option in addition to the `-f` option. If the filename provided to `-f` is '-' then connection information is read from the standard input device (`stdin`) until end-of-file.

The connection between the source and target correlators is persistent. When one of the correlators stops running then when that correlator restarts it automatically reconnects with the other correlator.

The tool is silent by default unless an error occurs. For verbose progress information use the `-v` option.

Options

Option	Description
`-h`	Display usage information
`-sn` *`host`*	Name of the host on which the source (event sending) correlator is running. The default is `localhost`. However, you can use the default or specify `localhost` only when the source correlator and the target correlator are running on the same host. In all other situations, you must specify the public IP address or the name of the host. This ensures that the host of the target correlator can resolve the name/address of the source correlator host. Non-ASCII characters are not allowed in host names.
`-sp` *`port`*	Port on which the source (event sending) correlator is listening (default is `15903`)
`-tn` *`host`*	Name of the host on which the target (event receiving) correlator is running. The default is `localhost`. However, you can use the default or specify `localhost` only when the source correlator and the target correlator are running on the same host. In all other situations, you must specify the public IP address or the name of the host. This ensures that the host of the source correlator can resolve the name/address of the target correlator host. Non-ASCII characters are not allowed in host names.
`-tp` *`port`*	Port on which the target (event receiving) correlator is listening (default is `15903`)
`-c` *`channel`*	Named channel on which to send/receive events. You can specify the `-c` option multiple times to send/receive events on multiple channels. You must specify the `-c` option at least once for each sender/target pair. Until you do, no events emitted by the sender correlator are received by the target correlator. An event is discarded if it is sent on a channel for which you did not specify the `-c` option.
`-m` *`mode`*	Indicates whether there is one connection (`-m legacy`) between the sender and target correlators or one connection for each specified channel (`-m parallel`). The default behavior is that there is one connection between the sender and target correlators. The utility uses the same connection for every channel. Events sent on any channel are delivered to the default channel in the target correlator and all events are delivered in

Option	Description
	order. You can specify default behavior by specifying `-m legacy` or `--mode legacy`. To create a connection for each specified channel, specify `-m parallel` or `--mode parallel`. Events sent on a named channel are delivered to the same named channel in the target correlator. Events sent on the same channel are delivered in order. Events sent on different channels may be re-ordered. You also specify the `-m` option when you specify the `-x` option to disconnect. If you are using a separate connection for each channel you should specify `-m parallel` when you specify the `-x` option. If you are using one connection for all channels you should specify `-m legacy` when you specify the `-x` option. See also "Avoid mixing connection modes" later in this section.
`-x`	When you specify the `-x` option the behavior depends on whether you also specify the `-c` option. If you specify the `-x` option and you do not also specify the `-c` *`channel`* option then the source correlator stops sending events to the target correlator. Each connection between the source correlator and the target correlator is terminated. If you specify the `-x` option and the `-c` *`channel`* option and the utility is using one connection for each channel then the source correlator terminates only the connection(s) it was using for the channel(s) you specify. Any other connections being used for other channels continue to be used. You can specify the `-x` option with one or more instances of the `-c` *`channel`* option. Remember to also specify `-m parallel`. If you specify the `-x` option and the `-c` *`channel`* option and the utility is using one connection for all channels then the source correlator stops sending events on only the channel(s) you specify. The source correlator continues to send events on any other channels it was already sending events on. If there are no other channels, then the source correlator no longer sends events to the target correlator. However, the connection between the two correlators remains in place. Remember to also specify `-m legacy`.
`-s`	Disconnect if slow (only takes effect on first connection)
`-f`	Read connection information from the named file. If this option is specified, the options `-sn -sp -tn -tp -c` are all ignored. This file must be in the local character set or in UTF-8 format. If it is UTF-8, specify the `-u` option in addition to this option.
`-u`	Indicates that the connection information file is in UTF-8.
`-v`	Requests verbose output

Option	Description
`-v`	Displays program name and version number

Operands

None.

Exit status

The following exit values are returned:

Status	Description
`0`	All connections were established successfully.
`1`	One or more source correlators could not be contacted
`2`	One or more target correlators could not be contacted
`3`	A problem occurred establishing the connection; request invalid
`4`	Target correlator failed to contact the Source
`5`	Some other error occurred

Comparison of legacy and parallel connection modes

Legacy connection mode	Parallel connection mode
0 or 1 connection between two correlators.	Any number of connections between correlators.
Events sent on different channels are delivered in the order in which they are sent.	Events sent on different channels may be delivered in a different order from the order in which they were sent.
Sending an event to a named channel delivers the event to the default channel.	Sending an event to a named channel delivers it to only that channel.
Unlike UM for passing events between correlators.	Similar to UM for passing events between correlators.
Same behavior as releases earlier than Apama 5.2.	New behavior starting with Apama 5.2.

UM has no mechanism for enforcing ordering among events sent on different channels. However, UM is the better alternative when you want to use a large number of channels to send events between components. Without UM, the use of two TCP connections with threads on both ends of the connection might reach the limit of how many channels can have dedicated connections.

Avoid mixing connection modes

Successive command lines that specify the same source/target hosts/ports build on each other. While this makes it possible to mix the legacy and parallel connection modes, you should avoid doing that. Mixing connection modes can cause an event to be delivered twice to the same channel. For example:

```
engine_connect -tp 15902 -sp 15903 -c channelA -c channelB
engine_connect -tp 15902 -sp 15903 -c channelA -c channelC -m legacy
```

The result of the first command is that there is one (legacy) connection for sending/receiving events on `channelA` and `channelB`. The result of the second command is that there is a dedicated connection for sending/receiving events on `channelA` and a dedicated connection for sending/receiving events on `channelC`. Events sent on `channelA` would be delivered twice — once on the legacy connection and once on the dedicated connection.

Examples

Because you can specify command lines that build on each other, you could set up a connection and add named channels later. You can also unsubscribe the channels you've added so that no events are sent or received. The connection remains and you can re-add channels at a later time. However, until you specify the `-c` option for a given connection, no events emitted by the source correlator are received by the target correlator. Consider the following command line:

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904
```

The correlators on `host1` and `host2` are connected but no channels have been subscribed and therefore no events are sent/received. To send and receive events, specify the `-c` option as in the following command line:

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -c CHAN1 -c CHAN2
```

Now the connected correlators can use `CHAN1` and `CHAN2` to send/receive events. To add another channel, execute this command:

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -c CHAN3
```

The correlators are now using `CHAN1`, `CHAN2`, and `CHAN3` to send/receive events. To stop using `CHAN2`, execute the following command. The correlators continue to use `CHAN1` and `CHAN3`.

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -x -c CHAN2
```

To stop sending and receiving events, execute the following command. Note that the correlators remain connected until one of them stops. There is no penalty for this connection.

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -x
```

In this example, the following command is equivalent to the previous command.

```
engine_connect -sn host1 -sp 15903 -tn host2 -tp 15904 -x -c CHAN1 -c CHAN3
```

Tuning Correlator Performance

Connection configuration file

`engine_connect` can take connection information from a file for connecting and disconnecting event correlators. A sample of such a configuration file is shown below, which defines the topology shown in "Example of a pipelined engine topology" on page 84.

```
# comments are allowed prefixed by a '#' - the rest of the line
# is ignored
generator:dopey.apama.com:1234
```

```
worker1:sleepy.apama.com:1234:generator{chan1,chan2,chan3}
worker2:grumpy.apama.com:1234:generator{chan2}
worker3:sneezy.apama.com:1234:worker1{w1_out}
worker4:bashful.apama.com:1234:worker2{w2_out_1,w2_out_2,w2_out_3}
tallier:happy.apama.com:1234:worker3{result},worker4{result}
watcher:doc.apama.com:1234:generator{chan1,chan2}
```

Connection configuration file format

Each entry in the configuration file specifies connection information for a single correlator in the deployment. Entries can be specified in any order. The general format of an entry is:

correlator_name[:*host*][:*port*][:*connection*[,*connection*...]]

where <connection> is defined as:

correlator_name[{*channel_name*[,*channel_name*...]}]

correlator_name is a symbolic identifier for a correlator, used to identify source correlators in target correlator connection information. It can consist of any combination of characters other than whitespace, colon, comma or open/close brace characters, which are reserved as separators. *host* and *port* identify the specific correlator this entry applies to. They can be omitted, in which case the defaults of `localhost` and `15903` are used respectively.

Following this information are details of all connections to source correlators for the current (target) entry. This information is omitted if no correlators sit 'upstream' of the current entry (as with the correlator `generator`, above). If there are multiple upstream source correlators, each name should be separated by a comma (as with `tallier`, above, which takes events from `worker3` and `worker4`).

For each connection it is possible to specify the channel(s) on which the target correlator will listen. If no channels are specified the target correlator will register to receive all events emitted by the source correlator regardless of channel (as with correlators `worker3` and `worker4` which register for all events from `worker1` and `worker2` respectively). One can specify specific channel names by enclosing them in braces and separating multiple channels by commas (as with `watcher` which registers with `generator` for all events on channels `chan1` and `chan2`).

In effect, the configuration file is a convenient way of grouping several calls to `engine_connect`. For example to set up the connections for the correlator `tallier` would require two commands using `engine_connect`:

```
>engine_connect -m parallel -sn sneezy.apama.com -sp 1234 -tn happy.apama.com
    -tp 1234 -c result
>engine_connect -m parallel -sn bashful.apama.com -sp 1234 -tn happy.apama.com
    -tp 1234 -c result
```

Errors in the configuration file

The configuration file can be used to both establish and remove connections in a multi-correlator engine topology. For example, assuming the above file is saved as `topology.dat`, the following commands will first set up then tear down all the connections specified therein:

```
>engine_connect -m parallel -f topology.dat
>engine_connect -m parallel -x -f topology.dat
```

In each of these cases, `engine_connect` will exit with non-zero exit status on the first error it detects in the configuration file. An error message will be printed to standard error (`stderr`).

Re-playing the configuration file

The behavior of `engine_connect` without the `-x` option is additive. This means that successive calls to `engine_connect` will attempt to add the channels specified to any existing connection between the

source and target correlator(s). For example, with reference to the configuration file above, these commands:

```
>engine_connect -m parallel -sn dopey.apama.com -sp 1234 -tn sleepy.apama.com
    -tp 1234 -c foo
>engine_connect -m parallel -f topology.dat
```

will first add a connection from correlator `generator` to `worker1` on channel `foo`, then (from the configuration file) extend that connection so that `worker1` also receives all events from `generator` emitted on channel `chan1`.

Once a connection is set up between two correlators on a channel, any further attempt to set up that connection on the same channel will have no effect. It is therefore possible to re-play the configuration file by invoking `engine_connect` without creating duplicate connections. This can be useful if there is an error in the configuration file signaled when `engine_connect` is called, as the error can be fixed and `engine_connect` re-run without requiring removal of connections that were successfully set up by the first call to `engine_connect`.

Event correlator pipelining

Configuring pipelining through the client API

Apama provides a C++ Client Software Development Kit (SDK). This allows software written in C++ to interface with a running event correlator or group of event correlators. Apama management tools such as `engine_connect` are written using this Client Software Development Kit.

The functionality of the Client SDK is found in the `lib` folder of the Apama installation and consists of the libraries `libengine_client.so.5.3` (on UNIX), or `engine_client5.3.lib` (on Windows). To code against the library use the definitions from the `engine_client_cpp.hpp` header file in the `include` folder.

Detailed information on how to use the integration library is available in "The C Cient Software Development Kit" in *Connecting Apama Applications to External Components*. This section looks at the specific methods that allow a developer to programmatically configure two event correlators to communicate in a pipelined arrangement.

The primary class contained in the library is

```
com::apama::engine::EngineManagement
```

An object instance of this class represents an event correlator within Apama, and allows a developer to:

- inject EPL files,
- delete EPL entities,
- send events into a correlator,
- get a correlator's current operational status,
- connect a receiver of events,
- connect a correlator as a consumer of another correlator.

The last capability is of direct interest here, and is supported through the following methods on the `EngineManagement` class;

```
/**
 * Connect this Event Correlator as an event receiver to another
 * Event Correlator.
```

```
 *
 * @param target The Correlator to connect to
 * @param channels  An array of names representing the channels to subscribe
 *  to. This is a null-terminated array of pointers to zero-terminated
 *  char arrays. If this is null or empty, subscribe to all channels.
 * @param disconnectSlow disconnect if slow. Only the first consumer's
 * disconnectSlow value is used; subsequent consumers added to this.
 * Default is false. EngineManagement object share the connection and
 * thus the disconnect behavior.
 * @param mode the connection mode to use,
 * defaults to legacy (single connection, all
 * events delivered to the default
 * channel). Set to CONNECT_PARALLEL for
 * connection per channel and channel values
 * passed through.
 * @return true if successful
 * @exception EngineException
 */
virtual bool attachAsEventConsumerTo(
    EngineManagement* target, const char* const* channels
       bool disconnectSlow=false, ConnectMode mode = CONNECT_LEGACY) = 0;
/**
 * Unsubscribe as an event receiver from another engine.
 *
 * @param target The Correlator to unsubscribe from.
 * @param channels An array of names representing the channels to
 *   unsubscribe from. This is a null-terminated array of pointers to
 *   zero-terminated char arrays. If this is null or empty unsubscribe
 *   from all channels.
 * @param mode the connection mode to use,
 * defaults to legacy (single connection, all
 * events delivered to the default
 * channel). Set to CONNECT_PARALLEL for
 * connection per channel and channel values
 * passed through.
 * @exception EngineException
 */
virtual void detachAsEventConsumerFrom(
    EngineManagement* target, const char* const* channels,
    ConnectMode mode = CONNECT_LEGACY) = 0;
```

The following sample code illustrates how to connect two correlators in a pipelined arrangement. The calls highlighted in **bold** are the important library calls.

```
  // Method that connects to 2 'Engines' (Correlators) and links them up
  // in a pipeline
  void connect(const char* host1, unsigned short port1,
    const char* host2, unsigned short port2,
    vector<const char*> channels, bool detach,
    int& rc )
  {
    const char* emsg = "";
    EngineManagement* engine1 = NULL;
    EngineManagement* engine2 = NULL;
    try {
      // a bit verbose
      cerr << "Requesting connection from host:" << host1
        << ", port:" << port1 << " to host:" << host2
        << ", port:" << port2 << endl;
      // Ensure valid arguments
      rc = 3; // this just sets a return value
      if ((host1==host2) && (port1==port2)) {
        cerr << "Connecting an engine to itself is disallowed"
          << endl;
        return;
    }
      // Make sure the channel list is NULL-terminated
      channels.push_back(NULL);
      // Attempt to connect to source Correlator
      rc = 1;
```

```
    emsg = "Failed to connect to source engine";
    if (!(engine1 =
      com::apama::engine::connectToEngine(host1, port1))) {
      throw EngineException(emsg);
    }
    // Attempt to connect to target Correlator
    rc = 2;
    emsg = "Failed to connect to target engine";
    if (!(engine2 =
      com::apama::engine::connectToEngine(host2, port2))) {
      throw EngineException(emsg);
    }
    // Connect target to source
    if (detach) {
      emsg = "Detach failed";
      engine2->detachAsEventConsumerFrom(engine1,&channels[0],CONNECT_LEGACY);
    }
    else {
      emsg = "Attach failed";
      if (
        !engine2->attachAsEventConsumerTo(engine1,&channels[0],CONNECT_LEGACY))
      {
        rc = 4;
        emsg = "Target engine could not connect to source engine";
        throw EngineException(emsg);
      }
    }
  }
  catch (EngineException ex) {
    string errmes;
    errmes += emsg;
    errmes += ": ";
    errmes += ex.what();
    throw EngineException(errmes.c_str());
  }
  // Shutdown cleanly
  rc = 5;
  if (engine1) {
    // Disconnect from Apama
    emsg = "Failed to disconnect from source engine";
    com::apama::engine::disconnectFromEngine(engine1);
  }
  if (engine2) {
    // Disconnect from Apama
    emsg = "Failed to disconnect from target engine";
    com::apama::engine::disconnectFromEngine(engine2);
  }
}
```

Event correlator pipelining

Event partitioning

Using `engine_connect` or the Apama client library, it is possible to create topologies of correlators across which an application's monitors can be partitioned. Use the `engine_inject` tool described in "Injecting code into a correlator" on page 119, or by means of the relevant functions of the client library, to load the relevant monitors directly on to the appropriate correlators, specifying the host and port for each correlator.

This scheme is suitable for most applications, as monitors can be loaded once when Apama is brought online. For some applications, however, there is a requirement for a dynamic routing mechanism that (depending on the requirements of the application) continually splits and/or duplicates the incoming event stream and sends it to two or more correlators. Use the IAF

`transportChannel` attribute to specify the channel an event is sent to, and connect that channel to the appropriate correlators.

Event correlator pipelining

Chapter 7: Managing and Monitoring over REST

Apama provides a REpresentational State Transfer (REST) HTTP API with which you can monitor Apama components. The monitoring capabilities are available to third-party managing and monitoring tools or to any application that supports sending and receiving XML documents over the HTTP protocol.

Apama components expose several URIs which can be used to either monitor or manage different parts of the system. Some are exposed by most Apama components. These are the generic management URIs. Some are exposed only by specific types of components. For example a correlator running on the default port of 15903 will expose a URI at `http://localhost:15903/correlator/status`. If an HTTP `GET` is issued against the URI, the correlator will return an XML document with the current status of the correlator.

Most URIs are purely for informational purposes and will only respond to HTTP `GET` requests and interacting with them will not change the state of the component. However, some URIs allow the state of the correlator to be modified. They will respond to one or more of the other HTTP methods. For example the `/logLevel` URI will accept an HTTP `PUT` request containing an XML document describing what the log level of the component should be set to. Note that in this case it will also accept a `GET` request which will report the current log level of the component, in keeping with REST principles.

All requests and responses over these interfaces have the same, simple elements. Those elements are: `prop`, `map` and `list`. All elements have a `name` attribute. The `prop` element simply represents a name-value pair with the name contained in the name attribute and the value being the content of the element. The `map` element is an unordered list of named elements which might be any of the three sets of elements though it is quite typically simply a `map` of `prop` elements. See the `/info` URI as an example. The list is very similar to the `map` element except that here the order is typically regarded as significant. All responses from these URIs have a top-level element with the name "`apama-response`" and similar all requests which are sent to these URIs should have a top-level element with the name "`apama-request`". If there is an error then a response called "`apama-exception`" will be returned.

The `/connections` URL in the Generic Management section is a good example of all these elements being used together. The top level element is a `map` which has two children, both lists, called `senders` and `receivers`. The lists contain a `map` for each sender and receiver. Each sender or receiver has a set of `prop` elements.

Method	URI	Description
GET	`/info`	Summary information about the component including its name, version, etc.
GET	`/ping`	Check if the component is running.

Method	URI	Description
GET	`/deepPing`	Check if the component is running.
GET	`/logLevel`	Display the current log level of the component.
PUT	`/logLevel`	Issues a request to change the log level of the component
GET	`/connections`	Display the connections to the component.
GET	`/info/argv`	Display the arguments that were specified when the component was started.
GET	`/info/envp`	Display the names and values of the environment variables in use.
GET	`/info/categories`	Display the categories available; currently `argv` and `envp`.
GET	`/correlator/status`	Display the runtime operational status of a running correlator.
GET	`/correlator/info`	Display information about the state of a running event correlator.
GET	`/iaf/status`	Summary information about the IAF component.

Generic Management

The Apama REST API `GET` methods return information about correlators, sentinel_agents and IAFs. The `PUT /logLevel` method changes the specified log level.

GET /info

This method returns summary information about the component including its name, version, etc. This is analogous to the data that can be retrieved with the Apama `engine_management` utility, for example, the `hostname` field is exactly what `engine_management --gethostname` gives you.

Response

The XML response to this method looks like this:

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="name">Apama Studio Correlator for Demo - Iceberg
            (Demo - Iceberg:DefaultCorrelator)</prop>
  <prop name="type">correlator</prop>
  <prop name="componentVersion">5.3</prop>
  <prop name="productVersion">5.3</prop>
  <prop name="buildNumber">rel/5.3.x@202939</prop>
  <prop name="buildPlatform">ia32-win32-msvc9</prop>
  <prop name="hostname">NBBEDUSERNAME.location.myco.com</prop>
  <prop name="username">username</prop>
  <prop name="pid">4684</prop>
  <prop name="port">15903</prop>
  <prop name="physicalId">5881164212676086470</prop>
  <prop name="logicalId">5881164212676086470</prop>
  <prop name="currentDirectory">C:\Users\username\SoftwareAG\ApamaWork_5.3_32bit\
             studio_workspace\Demo - Iceberg</prop>
  <prop name="uptime">3970854</prop>
</map>
```

GET /ping and GET /deepPing

Checks if the component is still running. If the component is running the client receives an empty response. A failure is a timeout waiting for the response.

Response

The empty response for the `Ping` and `deepPing` looks like this:

```
<?xml version="1.0"?>
<map name="apama-response"> </map>
```

GET /logLevel

This method displays the log level of a component.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="logLevel">INFO</prop>
</map>
```

PUT /logLevel

This methods sets the log level of a component.

Request

```
<?xml version="1.0"?>
<map name="apama-request">
  <prop name="logLevel">INFO</prop>
</map>
```

GET /connections

Gets the incoming and outgoing messaging connections to the given component, along with the channels subscribed to and whether or not the receivers are slow.

Response

The response for this method looks like the following

```
<?xml version="1.0"?>
<map name="apama-response">
```

```
  <list name="senders">
    <map name="correlator (on port 57042)">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57042</prop>
      <prop name="physicalId">432487434834</prop>
      <prop name="logicalId">432487434834</prop>
    </map>
    <map name="engine_send">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57077</prop>
      <prop name="physicalId">643690913</prop>
      <prop name="logicalId">643690913</prop>
    </map>
  </list>
  <list name="receivers">
    <map name="engine_receive">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57053</prop>
      <prop name="physicalId">69086982542</prop>
      <prop name="logicalId">69086982542</prop>
      <prop name="slow">false</prop>
      <prop name="disconnectIfSlow">false</prop>
      <list name="channels">
        <prop name="">chan1"</prop>
        <prop name="">chan2"</prop>
      </list>
    </map>
    <map name="engine_receive">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">58032</prop>
      <prop name="physicalId">4325769021054</prop>
      <prop name="logicalId">4325769021054</prop>
      <prop name="slow">false</prop>
      <prop name="disconnectIfSlow">true</prop>
      <list name="channels">
        <prop name="">*</prop>
      </list>
    </map>
    <map name="correlator (on port 57042)">
      <prop name="remoteHost">127.0.0.1</prop>
      <prop name="remotePort">57071</prop>
      <prop name="physicalId">432487434834</prop>
      <prop name="logicalId">432487434834</prop>
      <prop name="slow">true</prop>
      <prop name="disconnectIfSlow">false</prop>
      <list name="channels">
        <prop name="">chan1"</prop>
      </list>
    </map>
  </list>
</map>
```

GET /info/argv

This method returns the arguments used when starting the component as a list of name/value pairs.

Response

```
<?xml version="1.0"?>
<list name="apama-response">
  <prop name="">C:\Program Files (x86)\Software AG\Apama 5.3\
            bin\correlator.exe</prop>
  <prop name="">--logQueueSizePeriod</prop>
  <prop name="">0</prop>
  <prop name="">-l</prop>
  <prop name="">C:\Users\username\SoftwareAG\ApamaWork_5.3_32bit\
            license\license.txt</prop>
  <prop name="">--port</prop>
  <prop name="">15903</prop>
  <prop name="">--loglevel</prop>
```

```
  <prop name="">INFO</prop>
  <prop name="">--name</prop>
  <prop name="">Apama Studio Correlator for Demo - Iceberg(Demo -
            Iceberg:DefaultCorrelator)</prop>
  <prop name="">-j</prop>
  <prop name="">--inputLog</prop>
  <prop name="">Default_Correlator_${START_TIME}_${ID}.input.log</prop>
</list>
```

GET /info/envp

This method returns a list the environment variables in use as a list of name/value pairs.

Response

```
<?xml version="1.0"?>
<list name="apama-response">
  <prop name="ALLUSERSPROFILE">C:\ProgramData</prop>
  <prop name="APAMA_HOME">C:\Program Files (x86)\Software AG\Apama 5.3</prop>
  <prop name="APPDATA">C:\Users\username\AppData\Roaming</prop>
  <prop name="COMMONPROGRAMFILES">C:\Program Files (x86)\Common Files</prop>
  <prop name="COMMONPROGRAMFILES(X86)">C:\Program Files (x86)\
            Common Files</prop>
  <prop name="COMMONPROGRAMW6432">C:\Program Files\Common Files</prop>
  <prop name="COMPUTERNAME">NBBEDUSERNAME</prop>
  <prop name="COMSPEC">C:\Windows\system32\cmd.exe</prop>
  <prop name="DRVDIR">C:\DRV</prop>
  <prop name="FP_NO_HOST_CHECK">NO</prop>
  <prop name="HOMEDRIVE">C:</prop>
  <prop name="HOMEPATH">\Users\username</prop>
...
  <prop name="TEMP">C:\Users\username\AppData\Local\Temp</prop>
  <prop name="TMP">C:\Users\username\AppData\Local\Temp</prop>
  <prop name="TYPE">Notebook</prop>
  <prop name="USERDNSDOMAIN">LOCATION.MYCO.COM</prop>
  <prop name="USERDOMAIN">MYCO</prop>
  <prop name="USERNAME">username</prop>
  <prop name="USERPROFILE">C:\Users\username</prop>
  <prop name="WINDIR">C:\Windows</prop>
  <prop name="WINDOWS_TRACING_FLAGS">3</prop>
  <prop name="WINDOWS_TRACING_LOGFILE">C:\BVTBin\Tests\installpackage\
            csilogfile.log</prop>
</list>
```

GET /info/category

This method returns the names categories for which you can get general information.

Response

```
<?xml version="1.0"?>
<list name="apama-response">
  <prop name="categories">ha</prop>
  <prop name="categories">categories</prop>
  <prop name="categories">argv</prop>
  <prop name="categories">envp</prop>
</list>
```

Managing and Monitoring over REST

Correlator Management

The Apama REST API provides URIs to use with the `GET` method in order to return information about running Apama correlators. This information includes data such as the number of listeners, monitors, and contexts in the correlator, the number and types of events, and the number of JMon

applications. For details about the type of information returned by these methods, see "Watching correlator runtime status" on page 131 and "Inspecting correlator state" on page 136.

GET /correlator/status

This is analogous to the information reported by the Apama `engine_watch` utility.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="numConsumers">1</prop>
  <prop name="numOutEventsQueued">0</prop>
  <prop name="numOutEventsUnAcked">0</prop>
  <prop name="numOutEventsSent">62</prop>
  <prop name="uptime">1061116</prop>
  <prop name="numMonitors">15</prop>
  <prop name="numProcesses">16</prop>
  <prop name="numJavaApplications">0</prop>
  <prop name="numListeners">63</prop>
  <prop name="numEventTypes">110</prop>
  <prop name="numQueuedFastTrack">0</prop>
  <prop name="numQueuedInput">0</prop>
  <prop name="numReceived">3</prop>
  <prop name="numFastTracked">101</prop>
  <prop name="numEmits">227</prop>
  <prop name="numProcessed">260</prop>
  <prop name="numSubListeners">63</prop>
  <prop name="numContexts">1</prop>
  <prop name="virtualMemorySize">235624</prop>
  <prop name="numSnapshots">0</prop>
  <prop name="numInputQueuedInput">0</prop>
  <prop name="mostBackedUpInputContext"><none></prop>
  <prop name="mostBackedUpICQueueSize">0</prop>
  <prop name="mostBackedUpICLatency">0.0</prop>
</map>
```

GET /correlator/info

This is analogous to the information reported by the Apama `engine_inspect` utility.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <list name="eventTypes">
    <map name="eventType">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">Match</prop>
      <prop name="eventTemplates">0</prop>
    </map>
    <map name="eventType">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">Trade</prop>
      <prop name="eventTemplates">1</prop>
    </map>
    <map name="eventType">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">VwapWatch</prop>
      <prop name="eventTemplates">2</prop>
    </map>
  </list>
  <list name="timers">
    <map name="timer">
      <prop name="nameSpace"></prop>
      <prop name="name">wait</prop>
      <prop name="timers">1</prop>
    </map>
  </list>
```

```
  <list name="containerTypes"/>
  <list name="monitors">
    <map name="monitor">
      <prop name="nameSpace">com.apama.samples.vwap</prop>
      <prop name="name">Vwap</prop>
      <prop name="subMonitors">2</prop>
    </map>
  </list>
  <list name="javaApplications"/>
  <list name="aggregates"/>
  <list name="contexts">
    <map name="context">
      <prop name="name">main</prop>
      <prop name="subMonitors">2</prop>
      <prop name="queueSize">0</prop>
    </map>
  </list>
</map>
```

Managing and Monitoring over REST

IAF Management

The Apama REST API provides a URI to use with the `GET` method in order to return information about adapters running in the Apama Integration Adapter Framework (IAF). This information includes data such as the number of codecs and transports in use and the number of events sent and received.

`GET /iaf/status`

This returns the same information as the output of the Apama `iaf_watch` utility.

Response

```
<?xml version="1.0"?>
<map name="apama-response">
  <prop name="uptime">1235213</prop>
  <prop name="uptimeSinceLastReconfiguration">43432</prop>
  <prop name="numReconfigurations">3</prop>
  <prop name="numTransports">2</prop>
  <prop name="numCodecs">1</prop>
  <prop name="numDownstreamMappings">5</prop>
  <prop name="numUpstreamMappings">2</prop>
  <prop name="numApamaSinks">1</prop>
  <prop name="numApamaSources">1</prop>
  <prop name="numTransportReceived">441414</prop>
  <prop name="numTransportSent">34134</prop>
  <prop name="numApamaReceived">34134</prop>
  <prop name="numApamaSent">44414</prop>
  <list name="transportStatus">
    <map name="ATITransportStatus">
      <prop name="name">ATITransport</prop>
      <prop name="status">All OK</prop>
    </map>
    <map name="ReutersTransport">
      <prop name="name">ReutersTransport</prop>
      <prop name="status">Stuffed</prop>
    </map>
  </list>
  <list name="codecStatus">
    <map name="NullCodec">
      <prop name="name">NullCodec</prop>
      <prop name="status">Lovely</prop>
    </map>
  </list>
```

```
  <map name="mapperStatus">
    <prop name="name">Semantic Mapper</prop>
    <prop name="status">OK</prop>
  </map>
  <map name="apamaStatus">
    <prop name="name">Apama</prop>
    <prop name="status">OK</prop>
  </map>
</map>
```

Managing and Monitoring over REST

Chapter 8: Correlator Utilities Reference

The Apama event correlator is at the heart of Apama applications. The correlator is Apama's core event processing and correlation engine. This section provides information and instructions for using command-line tools to monitor and manage event correlators.

The command-line tools documented in this book are in the `bin` directory of the Apama installation. By default, the installation directory is `/opt/apama_5.3` on UNIX, and `\Program Files\Software AG\Apama 5.3` on Windows.

For information about EPL, event definitions, monitors, namespaces and packages, see "Getting Started with Apama EPL" in *Developing Apama Applications*.

Starting the event correlator

This topic provides information about starting the event correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

This topic is organized as follows:

Synopsis

To start the event correlator:

- On Windows, run `correlator.exe`.
- On UNIX, run `correlator`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: correlator [ options ]
Where options include:
  -V | --version                   Print version info and exit
  -h | --help                      This message
  -p | --port <port>               Listening on <port>
  -r | --rnames <rname[, rname]*>  UM realm names to connect to
  -f | --logfile <file>            Correlator status log file name is <file>
  -v | --loglevel <level>          Log level is <level>
  -t | --truncate                  Truncate log file at startup
  -N | --name <name>               Component name is <name>
  -l | --license <file>            License filename is <file>
  -m | --maxoutstandingack <num>   Buffer up to <num> secs of events/receiver
  -M | --maxoutstandingkb <num>    Buffer up to <num> kb of events/receiver
  -x | --qdisconnect               Disconnect slow event consumers
       --logQueueSizePeriod <p>    Send info to log every <p> seconds
       --distMemStoreConfig <dir>  Enable Distributed MemoryStore using
                                   configuration files in <dir>
       --jmsConfig <dir>           Enable JMS messaging using configuration
                                   files in <dir>
  -j | --java                      Enable Java application support
  -J | --javaopt <option>          Option to pass to JVM
       --inputLog <file>           Input log file name is <file>
  -XsetRandomSeed <num>            Set the seed of correlator random number
                                   generator to <num>
  -XignoreEnqueue                  Ignore all enqueue statements (for replay)
  --inputQueueSize <num>           Set the capacity of the input queue
  -g | --nooptimize                Disable optimizations
  -P                               Enable persistence
  -PsnapshotInterval=<interval>    The persistent correlator snapshot interval
  -PadjustSnapshot=<true/false>    Automatically adjust the snapshot interval
  -PstoreLocation=<path>           The path where the persistent correlator
                                   stores its recovery datastore
  -PstoreName=<file>               The name of the recovery datastore file
  -Pclear                          Clear contents of recovery datastore if
                                   one exists
  -XrecoveryTime <num>             The time used after the recovery(seconds
                                   since the epoch)
  -noDatabaseInReplayLog           Disallow database dumps to input log
  --runtime <type>                 EPL runtime to use, interpreted (default) or compiled
                                   (UNIX only)
  --scheduler <type>               Scheduler tuning: eventProcessing or heavyCompute
  --frequency <num>                Set the number of clock ticks generated per second
  -Xclock | --externalClock        Use external clocking (&TIME events)
  -Xconfig | --configFile <file>   Use service configuration file <file>
  -UMconfig | --umConfigFile       Use configuration file to configure UM server

The loglevel argument must be one of:
  OFF, CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE
```

```
Note: Without specification of a license file, the correlator runs for
      30 minutes and accepts only local clients.
```

Description

By default, the `correlator.exe` (on Windows) or `correlator` (on UNIX) tool starts an event correlator process on the current host, and configures it to listen on port `15903` for monitoring and management requests.

On start-up, the executable displays the current version number and configuration settings.

To terminate an event correlator process, press **Ctrl-C** in the window in which it was started. Alternatively, you can issue the `engine_management` command with the `--shutdown` option. See "Shutting down and managing components" on page 138. Regardless of which technique you use to terminate the correlator, Apama first tries to shut down the correlator cleanly. If this is not possible, for example, perhaps because of a monitor in an infinite loop, Apama forces the correlator to shut down.

Options

When you start the correlator, there are a number of options that you can specify. They are described in the following table:

Option	Description
`-V | --version`	Displays version information for the correlator.
`-h | --help`	Displays usage information.
`-p` *port* `| --port` *port*	Specifies the port on which the event correlator should listen for monitoring and management requests. The default is `15903`.
`-f` *file* `| --logfile` *file*	Specifies the path of the status log file that the event correlator writes messages to. The default is `stdout`. See "Specifying log filenames" on page 111 and "Descriptions of correlator status log fields" on page 114.
`-v` *level* `| --loglevel` *level*	Specifies the level of information that the event correlator sends to the correlator status log file. In increasing amount of information order, the value must be one of the following: `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`. The default is `INFO`.
`-t | --truncate`	Specifies that if the correlator status log file already exists, the correlator should empty it first. The default is to append to it.
`-N` *name* `| --name` *name*	Assigns a name to the correlator. The default is `correlator`. If you are running a lot of correlators you might find it useful to assign a name to each correlator. A name can make it easier to use the `engine_management` tool to manage correlators and adapters.

Option	Description
`-l file | --license file`	Specifies the path to the file that contains the license string. If a license file is not specified, the correlator will start, but in a restricted mode that prevents connections to other hosts and it will terminate after 30 minutes of operation. If the license file exists but is expired or invalid then the correlator will fail to start.
`-m num | --maxoutstandingack num`	Specifies that you want the correlator to buffer messages for up to *num* seconds for each receiver that the correlator sends events to. The default is `10`. For additional information, see "Determining whether to disconnect slow receivers" on page 116.
`-M num | --maxoutstandingkb num`	Specifies that you want the correlator to buffer the events for each receiver up to the size in kb represented by *num*.
`-x | --qdisconnect`	Specifies that you want the correlator to disconnect receivers that are consuming events too slowly. For details, see "Determining whether to disconnect slow receivers" on page 116. The default is that the correlator does not disconnect slow receivers.
`--logQueueSizePeriod p`	Sets the interval at which the correlator sends information to its log file. The default is 5 seconds. Replace *p* with a `float` value for the period you want. **CAUTION**: Setting `logQueueSizePeriod` to `0` turns logging off. Without correlator logging information, it is impossible to effectively troubleshoot problems. See also "Descriptions of correlator status log fields" on page 114.
`--distMemStoreConfig dir`	Specifies that the distributed MemoryStore is enabled, using the configuration files in the specified directory. Note that the configuration filenames must end with `*-spring.xml` and the correlator will not start unless the specified directory contains at least one configuration file. For more information on a distributed MemoryStore's configuration files, see "Using the Distributed MemoryStore" in *Developing Apama Applications in EPL*.
`--jmsConfig dir`	Specifies that correlator-integrated messaging is enabled using the configuration files in the specified directory. Note that the configuration filenames must end with `*-spring.xml` and the correlator will not start unless the specified directory contains at least one configuration file. For more information on the configuration files for correlator-integrated messaging for JMS, see .

Option	Description
`-j | --java`	Enables support for Java applications, which is needed to inject a Java application or plug-in using `engine_inject -j`.
`-J option | --javopt option`	Specifies an option or property that you want the correlator to pass to the embedded JVM. You must specify the `-J` option for each JVM option. You can specify the `-J` option multiple times in the same `correlator.exe` command line. For example, `-J "-Djava.class.path=path" -J "-Da=value1" -J "-Db=value2" -J "-Xmx400m"` When you use this option to pass the classpath to the JVM Apama prepends the correlator internal `JAR` files to the path you specify. If you set the `CLASSPATH` environment variable and also specify this option when you start the correlator, the path you specify in the correlator start-up command takes precedence.
`--inputLog file`	Specifies the path of the input log file. The event correlator writes only input messages to the input log file. If there is a problem with your application, Software AG Global Support can use the input log to try to diagnose the problem. An input log contains only the external inputs to the correlator. There is no information about multi-context ordering. Consequently, if there is more than one context, there is no guarantee that you can replay execution in the same order as the original execution. See "Replaying an input log to diagnose problems" on page 161.
`--XsetRandomSeed int`	Starts the correlator with the random seed value you specify. Specify an integer whose value is in the range of 1 to 2^{32}.The correlator uses the random seed to calculate the random numbers returned by the `integer.rand()` and `float.rand()` functions. The same random seed returns the same sequence of random numbers. This option is useful when your application uses the `integer.rand()` and `float.rand()` functions and you are using an input log to capture events and messages coming into a correlator. If you need to reproduce correlator behavior from that input log, you will want the correlator to generate the same random numbers as it generated when the original input was captured.
`-XignoreEnqueue`	For internal use only. Instructs the correlator to ignore `enqueue` statements when replaying an input log.
`--inputQueueSize int`	Sets the maximum number of spaces in every context's input queue. The default is that each input queue has 20,000 spaces. If events are arriving on an input queue faster than the correlator can process them the input queue can fill up. You can set the `inputQueueSize` option to allow all input queues to accept more events. Typically, the default input

Option	Description
	queue size is enough so if you find that you need to increase the size of the input queue you should try to understand why the correlator cannot process the events fast enough to leave room on the default-sized queue. If you notice that adapters or applications are blocking it might be because a public context's input queue is full. To determine if a public context's input queue is full, use output from the `engine_inspect` utility in conjunction with the status messages in the correlator log file.
`-g | --nooptimize`	Disables correlator optimizations that hinder debugging. Specify this option when you plan to run the `engine_debug` utility. You cannot run the `engine_debug` utility if you did not specify the `-g` option when you started the correlator. Apama Studio automatically uses the `-g` option when it starts a correlator from a debug launch configuration. However, if you are connecting Apama Studio to an externally-started correlator, and you want to debug in that correlator, you must ensure that the `-g` option was specified when the externally-started correlator was started.
`-P`	Enables correlator persistence. You must specify this option to enable correlator persistence. If you do not specify any other correlator persistence options, the correlator uses the default persistence behavior as described in "Enabling correlator persistence" in *Developing Apama Applications in EPL*. If you specify one or more additional correlator persistence options, the correlator uses the settings you specify for those options and uses the defaults for the other persistence options.
`-PsnapshotInterval=`*`interval`*	Specifies the period between persistence snapshots. The default is 200 milliseconds.
`-PadjustSnapshot=`*`boolean`*	Indicates whether or not the correlator should automatically adjust the snapshot interval according to application behavior. The default is true, which means that the correlator does automatically adjust the snapshot interval. You might want to set this to false to diagnose a problem or test a new feature.
`--PstoreLocation=`*`path`*	Specifies the path for the directory in which the correlator stores persistent state. The default is the current directory, which is the directory in which the correlator was started.
`--PstoreName=`*`filename`*	Specifies the name of the file that contains the persistent state. This is the recovery datastore. The default is `persistence.db`.

Option	Description
`-Pclear`	Specifies that you want to clear the contents of the recovery datastore. This option applies to the recovery datastore you specify for the `-PstoreName` option or to the default `persistence.db` file if you do not specify the `-PstoreName` option. When the correlator starts it does not recover from the specified recovery datastore.
`-XrecoveryTime` *num*	For correlators that use an external clock, this is a time expression that represents the time of day that a correlator starts at when it recovers persistent state and restarts processing. The default is the time expression that represents the time of day captured in the last committed snapshot. This option is useful only for replaying input logs that contain recovery information. To change the default, specify a number that indicates seconds since the epoch.
`-noDatabaseInReplayLog`	Specifies that the correlator should not copy the recovery datastore to the input log when it restarts a persistence-enabled correlator. The default is that the correlator does copy the recovery datastore to the input log upon restarting a persistence-enabled correlator. You might set this option if you are using an input log as a record of what the correlator received. The recovery datastore is a large overhead that you probably do not need. Or, if you maintain an independent copy of the recovery datastore, you probably do not want a copy of it in the input log.
`--runtime` *mode*	On Linux 64-bit systems, you can specify whether you want the correlator to use the compiled runtime or the interpreted runtime, which is the default. Specify `--runtime compiled` or `--runtime interpreted`. The interpreted runtime compiles EPL into bytecode whereas the compiled runtime compiles EPL into native code that is directly executed by the CPU. For many applications, the compiled runtime provides significantly faster performance than the interpreted runtime. Applications that perform dense numerical calculations are most likely to have the greatest performance improvement when the correlator uses the compiled runtime. Applications that spend most of their time managing listeners and searching for matches among listeners typically show a smaller performance improvement. Other than performance, the behavior of the two runtimes is the same except • The interpreted runtime allows for the profiler and debugger to be switched on during the execution of EPL. The compiled runtime does not permit this. For example,

Option	Description
	you cannot switch on the profiler or debugger in the middle of a loop. • The amount of stack space available is different for the two runtimes. This means that recursive functions run out of stack space at a different level of recursion on the two runtimes.
`--scheduler` *type*	When you specify `--runtime compiled` you can also specify the correlator scheduler type. The default is `eventProcessing`, which works best for most applications and provides optimal application latency. For higher throughput, you can specify `heavyCompute` as the scheduler type, which provides higher throughput for applications that spend a lot of time performing calculations in EPL but which may have worse latency.
`--frequency` *num*	Instructs the correlator to generate clock ticks at a frequency of *num* per second. Defaults to `10`, which means there is a clock tick every 0.1 seconds. Be aware that there is no value in increasing *num* above the rate at which your operating system can generate its own clock ticks internally. On UNIX and some Windows machines this is 100 and on other Windows machines it is 64.
`-Xclock | --externalClock`	Instructs the correlator to disable its internal clock. By default, the correlator uses internally generated clock ticks to assign a timestamp to each incoming event. When you specify the `-Xclock` option, you must send time events (`&TIME`) to the correlator. These time events set the correlator's clock. For additional information, see "Determining whether to disable the correlator's internal clock" on page 118.
`-Xconfig` *file* `| --configFile` *file*	Specifies a special configuration file that the correlator uses to initialize its networking. Specify this option only as directed by Apama Technical Support to troubleshoot networking problems. For more information, see "Using the Apama component extended configuration file" on page 169.
`-r` *list* | `--rnames` *list*	Specifies one or more Universal Messaging (UM) realm names that you want this correlator to connect to. Separate names with commas. See . If you specify the `--rnames` option and also the `--umConfigFile` option, the specified UM configuration file takes precedence.
`-UMConfig` *path* or `--umConfigFile` *path*	Specifies the name of a properties file that defines the UM configuration for this correlator. See .

Option	Description
	If you specify the `--rnames` option and also the `--umConfigFile` option, the specified UM configuration file takes precedence.

Exit status

The `correlator.exe` tool returns the following exit values:

Value	Description
`0`	The event correlator terminated successfully.
`1`	An error occurred which caused the event correlator to terminate abnormally.

Correlator Utilities Reference

Specifying log filenames

A correlator can send information to the following log files:

- Correlator status log file

  Upon correlator startup, the default behavior is that the correlator logs status information to `stdout`. To send correlator status information to a file, specify the `-f file` or `--logfile file` option and replace `file` with a log filename. The format for specifying a log filename is described below.

  Before you specify a log filename, you should consider your log rotation policy, which can determine what you want to specify for the log filename. See "Rotating the correlator log file" on page 150.

- Correlator input log file

  When you start a correlator you can specify the `--inputlog file` option so that the correlator maintains a special log file for all inputs. Again, before you specify the log filename, consider the rotation policy for your input log files. See "Rotating an input log file" on page 161.

- Application log files

  You can create log files for packages, monitors, and events in your application. The format you use to specify these log filenames is the same as for status logs and input logs. For details about how to create application log files, see "Setting logging attributes for packages, monitors and events" on page 147.

The format for specifying a log filename is as follows:

```
file[${START_TIME}][${ROTATION_TIME}][${ID}].log
```

The following table describes each part of a log filename specification. You cannot include spaces. You can separate the parts of the filename specification with underscores. You can specify `${START_TIME}`, `${ROTATION_TIME}` and/or `${ID}` in any order. For examples, see "Examples for specifying log filenames" on page 113.

file	Replace *file* with the name of the file that you want to be the log file. If you specify the name of a file that exists, the correlator overwrites it on Windows or appends to it on UNIX. Required. If you also specify ${START_TIME} and/or ${ROTATION_TIME} and/or ${ID}, the correlator prefixes the name you specify to the time the correlator was started and/or the time the log file was rotated (logging to a new file began) and/or an ID, beginning with 001. Be sure to specify a location that allows fast access.
${START_TIME}	Tag that indicates that you want the correlator to insert the date and time that it started into the log filename. Optional, however you probably want to always specify either this option or ${ROTATION_TIME} to avoid overwriting log files. This tag is also useful for correlators that you start from Apama's Management and Monitoring console, because it lets you distinguish the logs from different correlators.
${ROTATION_TIME}	Tag that indicates that you want the correlator to insert the date and time that it starts sending messages to a new log file into the log filename. Optional. If you specify ${ROTATION_TIME} and this log filename specification appears in a correlator start-up command then the name of the initial log file contains the time the correlator started. This tag is also useful for correlators that you start from Apama's Management and Monitoring console, because it lets you distinguish the status logs from different correlators.
${ID}	Tag that indicates that you want the correlator to insert a three-digit ID into the filename of the log file. The ID that the correlator inserts first is 001. Optional. The log ID increment is related only to rotation of log files. See "Rotating the correlator log file" on page 150 and "Rotating an input log file" on page 161. The ID allows you to create a sequence of log files. Each time the log file is rotated, the correlator increments the ID. A sequence of log files have the same name except for the ID. If you also specify ${ROTATION_TIME} then a sequence of log files have the same name except for the rotation time and the ID. Restarting the correlator always resets the ID portion of the log filename to 001.

If you plan to rotate log files then be sure to specify ${ROTATION_TIME} or ${ID}. You can also specify both.

Starting the event correlator

Examples for specifying log filenames

This topic provides examples of specifying log filenames. The format for specifying a log filename is the same in the following cases:

- Starting the correlator and specifying a correlator status log file with the `--logfile` option.
- Starting the correlator and specifying a correlator input log file with the `--inputLog` option.
- Invoking `engine_management -r setLogFile` to rotate the correlator status log.
- Invoking `engine_management -r setApplicationLogFile` to create a log file for a package, monitor or event.

The following specifies that the name of the status log is `correlator_status.log`.

```
--logfile corelator_status.log
```

Suppose that the correlator processes events for a while, sends information to `correlator_status.log`, and then you find that you need to restart the correlator. If you restart the correlator and specify the exact same log filename, the correlator overwrites the first `correlator_status.log` file. To avoid overwriting a status log, specify `${START_TIME}` in the log file name specification when you start the correlator. For example:

```
--logfile correlator_status_${START_TIME}.log
```

This command opens a status log with a name something like the following:

```
correlator_status_2015-03-12_15:12:23.log
```

This ensures that the correlator does not overwrite a log file. Now suppose that you want to be able to rotate the log, so you specify the `${START_TIME}` and `${ID}` tags:

```
correlator_status_${START_TIME}_${ID}.log
```

The initial name of the log file is something like the one on the first line below. If you then rotate the log file then the correlator closes that file and opens a new file with a name like the one on the second line:

```
correlator_status_2015-03-12_15:12:23_001.log
correlator_status_2015-03-12_15:12:23_002.log
```

To specify an application log filename for messages generated in `com.example.mypackge`, you could specify the log filename as follows:

```
mypackage_${ID}_${ROTATION_TIME}.log
```

With that specification, messages generated in `com.example.mypackage` would go to a file with a name such as the one on the first line below. The time in the initial application log filename is the time that the initial log file is created. If you rotate the logs every 24 hours at midnight then the names of subsequent application log files would be something like the names in the second and third lines below.

```
mypackage_001_2015-03-21_18:42:06.log
mypackage_002_2015-03-22_00:00:00.log
mypackage_003_2015-03-23_00:00:00.log
```

UNIX Note

In most UNIX shells, when you start a correlator you most likely need to escape the tag symbols, like this:

```
correlator -l license --inputLog input_\${START_TIME}_\${ID}.log
```

Starting the event correlator

Descriptions of correlator status log fields

The correlator sends information to its status log file every five seconds or at an interval that you set with the `--logQueueSizePeriod` option. When logging at `INFO` level, this information contains the following:

```
Status: sm=2 nctx=2 ls=3 rq=0 eq=0 iq=0 oq=0 icq=12 lcn="input ctx one"
lcq=12 lct=0.8 rx=5 tx=20 rt=7 nc=1 vm=369768 runq=0 si=0.0 so=0.0
srn="apamacluster1_node3" srq=3
```

Where the fields are as follows:

Field	Description of correlator status log field
`sm`	Sub-Monitors — Number of monitor instances, sometimes called sub-monitors. This is the sum of monitor instances in the main context plus monitor instances in any other context.
`nctx`	Number of contexts — For applications developed prior to Apama 4.1, this is always `1`.
`ls`	Listener sum — This is the number of listeners in the main context plus the number of listeners in any other context. This includes each `on` statement and each stream source template, for example, `all Tick(symbol="APMA")` in the following: `stream<Tick> ticks := all Tick(symbol="APMA");`
`rq`	Route queue — Sum of the number of routed events on the input queues of all contexts. A routed event is any event that has been generated by EPL's `route` keyword or JMon's `route()` method. A routed event goes to the front of that context's input queue. This ensures that the correlator processes routed events before processing external events. This number can go up and down, and it tends to be `0` for an idle correlator.
`eq`	Enqueue queue — Number of events on the enqueued events queue. An enqueued event is any event that has been generated with the EPL `enqueue` keyword (not `enqueue...to`) or JMon `enqueue()` method. A separate thread moves events from the enqueued events queue to the input queue of each public context. The size of the enqueued events queue is unbounded. Consequently, it is possible for this queue to use a large amount of memory if one or more input queues are full.
`iq`	Input queue — Sum of the number of entries on the input queues of all contexts. This number excludes routed events. It includes events from external sources, injections of EPL files, delete requests, time ticks, pending spawn-to operations, and enqueued events. This number goes up and down, and tends to be `0` for an idle correlator. It is

Field	Description of correlator status log field
	possible for the total number of input queue entries (`iq`) to be greater than the number of events the correlator has received from external sources (`rx`).
`oq`	Output queue — Number of events on the correlator's output queue. This is the number of events that the correlator has emitted but not yet sent to any receivers. If the correlator is idle, this number is `0`.
`icq`	Sum of enqueued events on the input queues of all public contexts. These events are a subset of the events included in the `iq` count.
`lcn`	The name of the public context whose input queue is most backed up in time. This is not necessarily the public context whose input queue has the largest number of entries. If no public context has entries on its input queue then this value is `"<none>"`. The name of the main context is always `main`.
`lcq`	For the context identified by `lcn`, this is the number of entries on the most backed up input queue.
`lct`	For the context identified by `lcn`, this is the time difference between its current logical time and the most recent time tick added to its input queue.
`rx`	Number of events the correlator has received from external sources since the correlator was started. The correlator increments this count as soon as it receives an event. After incrementing this count, the correlator parses the event to determine if it is an event for which a definition has already been injected. If the correlator can successfully parse the event, the event goes to the input queue of each public context. If the correlator cannot parse the event, the correlator discards the event. This is not the number of events that the correlator has processed. This count does not include routed and enqueued events. This number never goes down; it can only go up.
`tx`	Number of events the correlator has sent to receivers since the correlator was started. This number includes duplicate events sent to multiple receivers. For example, suppose you inject EPL code that emits an event, and there are five receivers that are subscribed to channels that publish that event. In this situation, the `tx` count goes up by five. Although there was 1 event, it was sent five times — once to each subscribed receiver.
`rt`	Route total —Total number of events that have been routed across all contexts since the correlator was started.
`nc`	Number of receivers.
`vm`	Virtual memory — Number of kilobytes of virtual memory being used by the correlator process.
`runq`	Run queue — Number of contexts (public and private) that have work to do but are not currently running. These contexts are waiting for processing resources. This indicator is a measure of the load on the system. When this number is consistently more than 0 and latency is a problem you might want to consider adding CPUs to your configuration.

Field	Description of correlator status log field
`si`	The rate (pages per second) at which pages are being read from swap space.
`so`	The rate (pages per second) at which pages are being written to swap space.
`srn`	Slowest receiver name — The name of the receiver whose queue has the largest number of entries. If no receivers have queue entries then this value is `"<none>"`.
`srq`	Slowest receiver queue — For the receiver identified by `srn`, the slowest receiver, this is the number of entries on its queue.

Logging for correlators with correlator-integrated messaging for JMS enabled

Correlators with correlator-integrated messaging for JMS enabled send additional information to the status log file. For details on this information, see .

Starting the event correlator

Text internationalization issues

Apama translates the contents of the correlator status log from UTF-8 to the local character set before displaying it in a console or terminal.

Starting the event correlator

Determining whether to disconnect slow receivers

The correlator sends events to multiple receivers. Sometimes, a receiver cannot consume its events fast enough for the correlator to continue sending them. When this happens, the default behavior is that the correlator suspends processing until the receiver disconnects or starts consuming events fast enough. In other words, a slow receiver can prevent other consumers from receiving events. However, you might prefer to have the correlator disconnect a slow receiver and continue processing and sending events to other consumers. Information in this section can help you determine whether to disconnect slow receivers.

- "Description of slow receivers" on page 117
- "How frequently slow receivers occur" on page 117
- "Correlator behavior when there is a slow receiver" on page 118
- "Tradeoffs for disconnecting a slow receiver" on page 118

See also "Handling slow or blocked receivers", in the testing and tuning chapter of *Developing Apama Applications*.

Starting the event correlator

Description of slow receivers

Every event that the correlator sends to one of its receivers has a sequence number. After a receiver processes an event, it sends the event's sequence number back to the correlator as an acknowledgment that the receiver processed that event. By default, if the correlator does not receive an acknowledgment within `10` seconds after the correlator sent the event, the correlator marks that receiver as being slow to consume events.

You can control the length of time within which the receiver must acknowledge an event before it is marked as a slow receiver. When you start the correlator, you can specify the `-m` (or `--maxoutstandingack`) option and specify a number.

For example:

```
correlator -l etc\license.txt -m 15
```

If you start the correlator with this command, the correlator marks a receiver as slow if the correlator does not receive an acknowledgment within `15` seconds. If you do not specify the `-m` option, the default is `10` seconds. You should not specify a value under `1` second because doing so raises the risk that the correlator might designate a receiver as slow when it is in fact not slow.

The mechanism that flags a receiver as slow is not precise. If a receiver does not acknowledge an event sequence after `10` seconds (the default setting), the correlator does not immediately designate the receiver as slow. Typically, the designation happens within the next `5` seconds. If you change the value of the `-m` option, the slow designation takes effect between `1` and `1.5` times the value of the `-m` option.

Determining whether to disconnect slow receivers

How frequently slow receivers occur

In practice, sending acknowledgements should not be slow because a dedicated thread sends acknowledgments. Network interruptions are the most common cause of delayed acknowledgments. Of course, network interruptions affect events being sent as well.

Most receivers, including the `engine_receive` tool, normally send acknowledgments `0.1` seconds after the message was sent. Consequently, there is very little chance of a receiver being mistakenly designated as slow. In production, slow receivers should be rare as long as you have done the appropriate load testing before deployment.

If an engine library client blocks in the middle of a `sendEvents` call, the receiver cannot acknowledge messages while the client is blocked. As you know, a receiver is made up of an engine library and a client. Clients receive events by registering a `sendEvents` callback with the engine library. When the engine library gets an event from the correlator, it calls `sendEvents`. Problems that can cause a client to block are typically related to I/O, networking, or synchronization. The `sendEvents` call cannot complete until the problem is resolved. The receiver cannot send the acknowledgement until the `sendEvents` call completes. For example, the `engine_receive` tool is a receiver that is made up of an engine library and a client whose `sendEvents` callback writes events received to a disk file. If the client has to wait for the disk, then it is blocked. The likelihood of a `sendEvents` callback being blocked depends on what the client is doing. If the client is writing to a local disk, the process might block sometimes, but never more than a fraction of a second. However, sending the events over a slow or unreliable network might block for a while if the network, or the remote system cannot keep up with the event rate.

During development of event consumers, however, slow receivers are more likely. This can happen when a newly developed consumer receives an event from the correlator but cannot send the acknowledgment because of a deadlock. Another development problem might be that the event consumer cannot keep up with the load. If you have problems with slow receivers during development, you probably need to evaluate the design of your application.

Determining whether to disconnect slow receivers

Correlator behavior when there is a slow receiver

When the correlator has a slow receiver, it can behave in one of two ways:

- The default behavior is that the correlator blocks further processing. This is because a slow receiver causes the correlator's event output queue to become full. When the queue is full, the correlator stops processing because it has no place to put events. The processing thread stops and does not execute any more EPL code. The transport thread does not send any more events to any of the correlator's other receivers. The correlator resumes processing when the slow receiver disconnects or acknowledges the outstanding sequence number.
- The correlator disconnects the slow receiver, and continues processing events and sending them to its other receivers. To obtain this behavior, you specify the `-x` (or `--qdisconnect`) option when you start the correlator. The correlator sends a message to the receiver to indicate that the correlator is disconnecting the receiver. It is up to the receiver to reconnect.

To ensure that it has received an acknowledgment for every event sent, the correlator buffers each event that it sends until it receives the message's corresponding acknowledgment. When there is a slow receiver, this can use a lot of memory if the correlator is sending a large number of messages.

Determining whether to disconnect slow receivers

Tradeoffs for disconnecting a slow receiver

When you specify the `-x` option when you start the correlator, it means that the correlator always disconnects a slow receiver. There are two main disadvantages to this:

- The correlator loses the events that it sent to that receiver.
- It is possible for the correlator to disconnect a receiver that is temporarily overloaded, and to therefore lose events unnecessarily.

Clearly, losing events can be a very serious problem. This is why the default is that the correlator does not disconnect slow receivers.

The advantage of disconnecting a slow receiver is that the correlator continues processing events.

The correlator always sends a warning message to its status log when it detects a slow receiver. This lets you see where there are potential problems.

If you cannot allow the correlator to lose events, do not specify the `-x` option when you start the correlator.

Determining whether to disconnect slow receivers

Determining whether to disable the correlator's internal clock

When you start the correlator, you can specify the `-Xclock` option to disable the correlator's internal clock. By default, the correlator uses internally generated clock ticks to assign a timestamp to each

incoming event. When you specify the `-Xclock` option, you must send time events (`&TIME`) to the correlator. These time events set the correlator's clock.

Use `&TIME` events in place of the correlator's internal clock when you want to reproduce the historic behavior of an application. For example, Apama Studio's Data Player starts a correlator with a command that specifies the `-Xclock` option. The Data Player then sends `&TIME` events that let you play back events from the database.

A situation in which you might want to generate `&TIME` events is when you want to run experiments at faster than real time but still obtain correct timestamp behavior. In this situation, the correlator uses the externally generated time events instead of real time to invoke timers and wait events.

Disabling the correlator's internal clock, and sending external time events, affects all temporal operations, such as timers and `wait` statements.

Regardless of whether the correlator generates internal clock ticks, or receives external time events, the correlator assigns a timestamp to each incoming event. The timestamp indicates the time that the event arrived on the context's input queue. The value of the timestamp is the same as the last internally-generated clock tick or externally-generated time event. For example, suppose you have the following events and clock ticks:

```
&TIME(1)
A()
B()
&TIME(2)
C()
```

`A` and `B` receive a timestamp of `1`. `C` receives a timestamp of `2`.

For additional information about using external time events, see "Apama Concepts" in *Introduction to Apama*.

Determining whether to disconnect slow receivers

Injecting code into a correlator

To inject EPL files, JMon applications, or Correlator Deployment Packages (CDPs) into the correlator invoke the `engine_inject` tool. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To inject applications into the event correlator:

- On Windows, run `engine_inject.exe`.
- On UNIX, run `engine_inject`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_inject [ options ] [ file1 [ file2 ... ] ]
In order to inject to a correlator.
Where options include:
-h | --help            This message
-n | --hostname <host> Connect to an engine on <host>
-p | --port <port>     Engine is listening on <port>
-v | --verbose         Be more verbose
-u | --utf8            Assume input is in UTF8
-V | --version         Print program version info
-j | --java            Operate on Java applications rather than EPL or CDP
```

```
-c | --cdp              Operate on CDP files rather than EPL or Java
-s | --hashes           Print out hashes of (UTF8-encoded) files, rather
                        than injecting, to obtain hashes for Java or CDP
                        files also specify -j or -c
 Use '-' to read from <stdin>
```

Description

The `engine_inject` tool reads application definitions from the specified file(s) and injects them into an event correlator. If you do not specify a filename, or if you specify `"-"` as the filename, the correlator reads data from the standard input device (`stdin`) until you indicate the end of the file: Ctrl-D on UNIX and Ctrl-Z on Windows.

Application definitions can be monitors scripted in Apama's Event Processing Language. For more information on EPL, see "Introduction to Apama Event Processing Language" in *Developing Apama Applications*. Alternatively, you can specify the `-j` or `-c` options. The `-j` option specifies that you will inject an application written in Java. The `-c` option specifies that you will inject a Correlator Deployment Package file.

When you specify the `-j` option, each file you inject must be a Java archive file (`JAR`) that contains a single JMon application. For more information, see "Overview of JMon Applications" in *Developing Apama Applications*.

When you specify the `-c` option, the file you inject must be an Apama Correlator Deployment Package (CDP). For more information on preparing a CDP, see "Packaging EPL and Java code" on page 124.

By default, the `engine_inject` tool is silent unless an error occurs. To view information about `engine_inject` execution, specify the `--verbose` option.

If you try to inject invalid EPL files or invalid JMon applications, the event correlator generates an error. None of the application data in the invalid file is loaded. The `engine_inject` tool terminates. If you specify multiple EPL or Java files for injection the `engine_inject` tool injects all of them or terminates when it reaches the first file that contains an error. For example:

```
engine_inject 1.mon 2.mon 3.mon
```

If the `2.mon` file contains an error then `engine_inject` successfully injects `1.mon` and then terminates when if finds the error in `2.mon`. The tool does not operate on `3.mon`.

If you try to inject a CDP the correlator processes each EPL file packaged in the CDP separately. If one file in a CDP contains an error then the correlator reports an error for that file and does not run it but it does run the other files in the CDP (if they have no errors). It does not matter which file in the CDP contains the error. That is, the first file in the CDP that the correlator processes can contain an error and the correlator still runs the other files in the CDP if they contain no errors.

Options

The following table describes the options you can specify when you inject applications into the correlator:

Option	Description
`-h`	Displays usage information
`-n` *host*	Name of the host on which the event correlator is running. The default is `localhost`.

Option	Description
	Note: Non-ASCII characters in host names are not supported.
`-p port`	Port on which the event correlator is listening. The default is `15903`.
`-v`	Requests verbose output during `engine_inject` execution.
`-u`	Indicates that input files are in UTF-8 encoding. The default is that the `engine_inject` tool assumes that the EPL files to be injected are in the native character set of your platform. Set the `-u` option to override this assumption. The `engine_inject` tool then assumes that all input files are in UTF-8.
`-V`	Displays version information for the `engine_inject` tool.
`-j`	Indicates that each operand is a Java archive file (`JAR` file) that contains a single JMon application.
`-c`	Indicates that each operand is a Correlator Deployment Package file.
`-s`	Indicates that instead of injecting the specified files you want to print the hashes (UTF8-encoded) for the files. If `engine_inject` is operating on Java or Correlator Deployment Package (CDP) files then you must also specify `-j` or `-c`.

Operands

The `engine_inject` tool takes the following operands:

`[ file1 file2 ... ] ]`	The names of zero or more files that contain application data in Apama EPL, JMon, or Correlator Deployment Package files. If you do not specify one or more filenames, the `engine_inject` tool takes input from `stdin`.

Exit status

The `engine_inject` tool returns the following exit values:

Status	Description
`0`	All definitions were injected into the event correlator successfully.
`1`	No connection to the event correlator was possible or the connection failed.
`2`	Other error(s) occurred while injecting the supplied definitions.

Text encoding

By default, the `engine_inject` tool uses the default system encoding to determine the local character set. The `engine_inject` tool then translates all submitted EPL text from the local character set to UTF-8. Consequently, it is important to correctly set the machine's locale.

However, some input files might start with a UTF-8 Byte Order Mark. The `engine_inject` tool treats such input files as UTF-8 and does not do any translation. Alternatively, you can specify the `-u` option when you run the `engine_inject` tool. This forces the tool to treat each input file as UTF-8.

Correlator Utilities Reference

Deleting code from a correlator

The `engine_delete` tool removes EPL code and JMon applications from the correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To remove applications from the event correlator:

- On Windows, run `engine_delete.exe`.
- On UNIX, run `engine_delete`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_delete [ options ] [ name1 [ name2 ... ] ]
To delete named objects from a correlator.
Where options include:
    -h | --help                This message
    -n | --hostname <host>     Connect to an engine on <host>
    -p | --port <port>         Engine is listening on <port>
    -f | --file <filename>     Read names from <filename>
    -F | --force               Force deletion of names even if they are in use
    -k | --kill                Kill a monitor by name even if it is running
    -a | --all                 Delete everything in the engine - use with care
    -y | --yes                 Don't ask 'are you sure?' when deleting all
    -v | --verbose             Be more verbose
    -u | --utf8                Assume input is in UTF8
    -V | --version             Print program version info
    Use '-' to read from <stdin>
    Multiple -f options may be given
```

Description

The `engine_delete` tool deletes named applications, monitors and event types from an event correlator. Names are the full package names as previously assigned to an application monitor or event type when injected into the event correlator.

To specify the items you want to delete, you can specify any one of the following in the `engine_delete` command line:

- Names of the items to delete.
- The `-f` option with the name of a file that contains the names of the items you want to delete. In this file, specify each name on a separate line.
- Neither of the above. In this case, the `engine_delete` tool reads names from `stdin` until you type an end-of-file signal, (`CTRL-D` on UNIX and `CTRL-Z` on Windows). If you want, you can specify "-" in the command line to indicate that input will come from `stdin`.

The tool is silent by default unless an error occurs. To receive progress information, specify the `-v` option.

The tool permits two kinds of operations — delete and kill. These cause different side-effects and you must use them carefully.

- When you delete a monitor, the correlator tries to terminate all of that monitor's instances. If they are responsive (not in some deadlocked state) each one executes its `ondie()` action, and when the last one exits the correlator calls the monitor's `onunload()` action. This assumes that the monitor you are deleting defines `ondie()` and `onunload()` actions.

  If a monitor instance does not respond to a delete request, the correlator cannot invoke the monitor's `onunload()` action. In this case, you must kill, rather than delete, the monitor instance.

- When you kill a monitor, the correlator immediately terminates all of the monitor's instances, without invoking `ondie()` or `onunload()` actions.

Options

The tool `engine_delete` tool takes the following command line options:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *host*	Name of the host on which the event correlator is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *port*	Port on which the event correlator is listening. Optional. The default is `15903`.
`-f` *filename*	Indicates that you want the `engine_delete` tool to read names of items to delete from the specified file. In this file, each line contains one name. Optional. The default is that input comes from `stdin`.
`-F`	Forces deletion of named event types even if they are still in use — that is, they are referenced by active monitors or applications. A forced delete also removes all objects that refer to the event type you are deleting. For example, if monitor `A` has listeners for `B` events and `C` events and you forcibly delete `C` events the operation deletes monitor `A`, which of course means that the listener for `B` events is deleted. Optional. The default is that event types that are in use are not deleted.
`-k`	Kills all instances of the named monitor regardless of whether an instance is in use. For example, you can specify this option to remove a monitor that is stuck in an infinite loop. Any `ondie()` and `onunload()` actions defined in killed monitors are not executed.
`-a`	Forces deletion of all applications, monitors, and event types. The correlator finishes processing any events on input queues and then does the deletions. Any events sent after invoking `engine_delete -a` are not recognized. Specifying this option does not stop a monitor that is in an infinite loop. You must explicitly kill such monitors. Specifying the `-a` option is equivalent to specifying the `-F` option and naming every object in the correlator. If you want to kill every object in the correlator, shut down

Option	Description
	and restart the correlator. See "Shutting down and managing components" on page 138.
`-y`	Removes the "are you sure?" prompt when using the `-a` option.
`-v`	Requests verbose output.
`-u`	Indicates that input files are in UTF-8 encoding. This specifies that the `engine_delete` utility should not convert the input to any other encoding.
`-V`	Displays version information for the `engine_delete` tool.

Operands

The `engine_delete` tool takes the following operands:

`[ name1 [ name2 ... ] ]`	The name of zero or more EPL or JMon applications, monitors and/or event types to delete from the event correlator. If you do not specify at least one item name, and you do not specify the `-f` option, the `engine_delete` tool expects input from `stdin`.

Exit status

The following exit values are returned:

Status	Description
`0`	The items were deleted from the event correlator successfully.
`1`	No connection to the event correlator was possible or the connection failed.
`2`	Other error(s) occurred while deleting the named items.

Correlator Utilities Reference

Packaging EPL and Java code

The `engine_package` tool assembles EPL files and `.jar` files into a Correlator Deployment Package (CDP). You can inject a CDP file into the correlator just as you inject an EPL file or a JAR file containing a JMon application. CDP files use a proprietary, non-plaintext format that treats files in a manner similar to the way a JAR file treats a collection of Java files. In addition, using a CDP file guarantees that all files, assuming no errors, are injected and are injected in the correct order. See "Injecting code into a correlator" on page 119 for details about how the correlator handles an error in a file that is in a CDP.

While the names of events, monitors, aggregates, and `.jar` files that are contained in a CDP file are visible to the correlator utilities `engine_inspect`, `engine_manage`, and `engine_delete`, the code that defines them is not.

The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama command prompt ensures that the environment variables are set correctly.

Synopsis

To package files into a CDP file:

- On Windows, run `engine_package.exe`.
- On UNIX, run `engine_package`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_package [ options ] [ file1 [ file2 ... ] ]
In order to create a Correlator Deployment Package (CDP) from multiple files.
  Where options include:
    -h | --help            This message
    -V | --version         Print program version info
    -o | --output  <file>  Write the CDP to <file>
    -m | --manifest <file> Create the CDP from the manifest at <file>
    -u | --utf8            Assume input is in UTF-8
```

Description

The `engine_package` tool creates a Correlator Deployment Package (CDP). A CDP file contains one or more files. You specify the name of the CDP file to create as an argument to the `-o` option.

You can specify the files you want to include on the command line or you can use the `-m` option and specify a manifest file that contains the names of the files. The manifest file is a text file; each line in the file specifies a relative or absolute path to a file. Files should be listed in the order in which you want them to be injected into the correlator.

Options

The following table describes the options you can specify when you package files into a Correlator Deployment Package:

Option	Description
`-h`	Displays usage information
`-V`	Displays version information for the engine_package tool.
`-o` *`filename`*	Name of the CDP file to create. Required.
`-m` *`filename`*	Name of the manifest file that lists the files you want to package.
`-u`	Indicates that input files are in UTF-8 encoding. The default is that the `engine_package` tool assumes that the files to be packaged are in the native character set of your platform. Set the `-u` option to override this assumption. The `engine_package` tool then assumes that all input files are in UTF-8.

Operands

The `engine_package` tool takes the following operands:

`[ file1 [ file2 ... ] ]`	The names of the EPL or `.jar` files that contain code. The order in which these files are specified will become the order in which they are injected into the correlator when the CDP file is injected. Instead of listing the files on the command line, you can list them in a manifest file and use the `-m` option.

Exit status

The `engine_package` tool returns the following exit values:

Status	Description
`0`	On success.
`1`	On any error.

Example

The following example describes how to create a Correlator Deployment Package file with multiple monitor files and inject the CDP file into a running correlator.

1. Create a manifest file containing a list of files to include in the CDP. For this example, the file is named `manifest.txt` and each line contains the full path name of an EPL file or `.jar` file:

   ```
   c:\dev\sample\monitor1.mon
   c:\dev\sample\monitor2.mon
   C:\dev\sample\jmon-app.jar
   ```

2. To create the CDP file, call the `engine_package` utility stating the output filename and the manifest file to include in the CDP. (Note, instead of using a manifest file, you can list the files individually in the `engine_package` arguments.)

   ```
   engine_package.exe -o c:\sample.cdp -m c:\dev\sample\manifest.txt
   ```

3. To inject the CDP file, call the `engine_inject` utility with `-c` (or `--cdp`). This injects each file that is included in the CDP file into the correlator.

   ```
   engine_inject.exe -c c:\sample.cdp
   ```

   Sample output from the correlator

   ```
   2012-07-11 13:51:33.156 INFO  [3852] - Injected CDP from file
           c:\sample.cdp (b2f097b02791e5dd4ac73cda38e153e9),
           size 313 bytes, decoding and compile time 0.00 seconds
   ```

Correlator Utilities Reference

Sending events to correlators

The `engine_send` tool sends Apama-format events into an event correlator or IAF adapter. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

If the events you want to send are not in Apama format, you must use an adapter that can transform your event format into Apama event format.

Synopsis

To send Apama-format events to an event correlator or IAF adapter:

- On Windows, run `engine_send.exe`.
- On UNIX, run `engine_send`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_send [ options ] [ file1 [ file2 ... ] ]
In order to send to a Correlator or IAF.
Where options include:
    -h | --help                This message
    -n | --hostname <host>     Connect to an engine on <host>
    -p | --port <port>         Engine is listening on <port>
    -c | --channel <channel>   Send events on <channel> if not specified in event file
    -l | --loop <count>        Loop through input <count> times
    -v | --verbose             Be more verbose
    -u | --utf8                Assume input is in UTF8
    -V | --version             Print program version info
    Use '-' to read from <stdin>
    Loop count < 0 means loop forever
```

Description

The `engine_send` tool sends Apama-format events to an event correlator. In Apama-format event files, you can specify whether to send the events in batches of one or more events or at set time intervals.

The correlator reads events from one or more specified files. Alternatively, you can specify "`-`"or not specify a filename and the correlator reads events from `stdin` until it receives an end-of-file signal (**Ctrl-D** on UNIX and **Ctrl-Z** on Windows).

For details about Apama-format events, see "Event file format" on page 164.

By default, the `engine_send` tool is silent unless an error occurs. To view progress information during `engine_send` execution, specify the `-v` option when you invoke `engine_send`.

You can also use `engine_send` to send events directly to the Integration Adapter Framework (IAF). To do this, specify the port of the IAF, by default this is `16903`.

Options

The `engine_send` tool takes the following command line options:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *host*	Name of the host on which the event correlator to which you want to send events is running. Optional. The default is `localhost`. **Note:** Non-ASCII characters in host names are not supported.
`-p` *port*	Port on which the event correlator is listening. Optional. The default is `15903`.

Option	Description
`-c channel`	For events for which a channel is not specified, this option designates the delivery channel. If a channel is not specified for an event and you do not specify this option, the event is delivered to the default channel, which is the empty string. All public contexts receive events sent to the default channel. All queries receive events sent to the default channel. To send events to only running Apama queries, specify the `com.apama.queries` channel. See "Defining Queries" in *Developing Apama Applications*.
`-v`	Requests verbose output during execution. Optional.
`-V`	Displays version information for the `engine_send` tool. Optional.
`-l loop`	Number of times to cycle through and send the input events. Optional. Replace *loop* with one of the following values: • `0` — Indicates that you want the `engine_send` tool to iterate through and send the input data once. This is the default. • Any negative integer— Indicates that you want the `engine_send` tool to indefinitely cycle through and send the input events. • Any positive integer — Indicates the number of times to cycle through and send the input events. The `engine_send` tool ignores this option if you specify it and the input is from `stdin`.
`-u`	Indicates that input files are in UTF-8 encoding. This specifies that the `engine_send` utility should not convert the input to any other encoding.

Operands

The `engine_send` tool takes the following operands:

`[ file1 [ file2 ... ] ]`	Specify zero, one, or more files that contain event data. Each file you specify must comply with the event file format described in "Event file format" on page 164. If you do not specify any filenames, the `engine_send` tool takes input from `stdin`.

Exit status

The following exit values are returned:

`0`	The events were sent successfully.
`1`	No connection to the event correlator was possible or the connection failed.
`2`	One or more other errors occurred while sending the events.

Operating notes

To end an indefinite cycle of sending events, press **Ctrl-C** in the window in which you invoked the `engine_send` tool.

You might want to indefinitely cycle through and send events in the following situations:

- In test environments. For example, you can use `engine_send` to simulate heartbeats. If you then kill the `engine_send` process, you can test your EPL code that detects when heartbeats stop.
- In production environments. For example, you can use the `engine_send` tool to initialize a large data table in the correlator.

Text encoding

By default, the `engine_send` tool checks the environment variable or global setting that specifies the locale because this indicates the local character set. The `engine_send` tool then translates EPL text from the local character set to UTF-8. Consequently, it is important to correctly set the machine's locale.

However, some input files might start with a UTF-8 Byte Order Mark. The `engine_send` tool treats such input files as UTF-8 and does not do any translation. Alternatively, you can specify the `-u` option when you run the `engine_send` tool. This forces the tool to treat each input file as UTF-8.

Correlator Utilities Reference

Receiving events from correlators

The `engine_receive` tool lets you connect to a running event correlator and receive events from it. Events received and displayed by the `engine_receive` tool are in Apama event format. This is identical to the format used to send events to the correlator with the `engine_send` tool. Consequently, it is possible to reuse the output of the `engine_receive` tool as input to the `engine_send` tool.

The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To receive Apama-format events from an event correlator:

- On Windows, run `engine_receive.exe`.
- On UNIX, run `engine_receive`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_receive [ options ]
To receive and log output events from a remote correlator.
Where options include:
 -h | --help               This message
 -n | --hostname <host>    Connect to an engine on <host>
 -p | --port <port>        Engine is listening on <port>
 -c | --channel <channel>  Listen on output channel <channel>
 -f | --filename <file>    Write to <file> instead of to standard out
 -s | --suppressBatch      Do not include BATCH timestamps in the output
    | --suppressbatch
 -z | --zeroAtFirstBatch   Measure BATCH timestamps from when the first
    | --zeroatfirstbatch   batch arrived, instead of from when
                           engine_receive was started
 -C | --logChannels        Include channels in output
```

```
-r | --reconnect          Automatically (re)connect to the server when available
-x | --qdisconnect        Disconnect from correlator if we don't keep up
-v | --verbose            Be more verbose
-u | --utf8               Write output in UTF8
-V | --version            Print program version info
-Xconfig | --configFile <file> Use service configuration file <file>
   Multiple -c options may be given
```

Description

The `engine_receive` tool receives events from an event correlator and writes them to `stdout` or to a file that you specify. The correlator output format is the same as that used for event input and is described in "Event file format" on page 164.

You can specify one or more channels on which to listen for events from the correlator. The default is to receive all output events. For information about event channels, see "Understanding contexts and channels" in *Introduction to Apama*.

To view progress information during `engine_receive` execution, specify the `-v` option.

You can also use `engine_receive` to receive events emitted by the Integration Adapter Framework (IAF) directly. To do this, specify the port of the IAF, by default this is `16903`.

Options

The tool `engine_receive.exe` (on Windows) or `engine_receive` (on UNIX) takes a number of command line options. These are:

Option	Description
`-h`	Displays usage information. Optional.
`-n` *host*	Name of the host on which the event correlator is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *port*	Port on which the event correlator is listening. Optional. The default is `15903`.
`-c` *channel*	Named channel on which to listen for output events from the correlator. Optional. The default is to listen for all output events. You can specify the `-c` option multiple times to listen on multiple channels.
`-C`	Specifies that you want `engine_receive` output to include the channel that an event arrives on. If you then use the `engine_receive` output as input to `engine_send`, events are delivered back to the same-named channels. See "Event association with a channel" on page 167.
`-f` *file*	Dumps all received events in the specified file. Optional. The default is to write the events to `stdout`.
`-s`	Omits `BATCH` timestamps from the output events. Optional. The default is to preserve `BATCH` timestamps in events.

Option	Description
`-z`	Records the first received batch of events as being received at `0` milliseconds after the `engine_receive` tool was started. Optional. The default is that the first received batch of events is received at the number of milliseconds since `engine_receive` actually started.
`-r`	Automatically (re)connect to the server when available
`-x`	Disconnect from the correlator if the `engine_receive` utility cannot keep up with the events from the correlator.
`-v`	Requests verbose output during `engine_receive` execution. Optional.
`-u`	Indicates that received event files are in UTF-8 encoding. This specifies that the `engine_receive` utility should not convert the input to any other encoding.
`-V`	Displays version information for the `engine_receive` tool. Optional.
`-Xconfig` *`file`*	Specifies a special configuration file that the correlator uses to initialize its networking. Specify this option only as directed by Apama Technical Support to troubleshoot networking problems. For more information, see "Using the Apama component extended configuration file" on page 169.

Exit status

The `engine_receive` tool returns the following exit values:

Status	Description
`0`	All events were received successfully.
`1`	No connection to the event correlator was possible or the connection failed.
`2`	Other error(s) occurred while receiving events.

Text encoding

The `engine_receive` tool translates all events it receives from UTF-8 into the current character locale. It is therefore important that you correctly set the machine's locale. To force the `engine_receive` tool to output events in UTF-8 encoding, specify the `-u` option.

Correlator Utilities Reference

Watching correlator runtime status

The `engine_watch` tool lets you monitor the runtime operational status of a running event correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

Synopsis

To monitor the operation of an event correlator:

- On Windows, run `engine_watch.exe`.
- On UNIX, run `engine_watch`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_watch [ options ]
To periodically poll status from a remote correlator.
Where options include:
  -h | --help               This message
  -n | --hostname <host>    Connect to an engine on <host>
  -p | --port <port>        Engine is listening on <port>
  -i | --interval <ms>      Poll every <ms> milliseconds
  -f | --filename <file>    Write to <file> instead of to standard out
  -r | --raw                Raw (i.e. parser friendly) output
  -t | --title              Add header to output (Raw mode only)
  -o | --once               Poll once then exit
  -v | --verbose            Be more verbose
  -V | --version            Print program version info
```

Description

This tool periodically polls an event correlator for status information, writing the returned status data to `stdout`. For additional progress information use the `-v` option.

The `engine_watch` tool returns the following information:

Status message	Meaning
`Uptime (ms)`	The time in milliseconds since this correlator was started. This figure is unaffected if the state of the correlator is restored from a checkpoint file.
`Number of monitors`	The number of EPL monitor definitions injected into the correlator. This figure changes upwards and downwards as monitors are injected, deleted or just expire. A monitor expires when each of its instances dies, or it has no listeners or streams left, or it causes a runtime error.
`Number of sub-monitors`	The number of EPL monitor instances across all contexts in the correlator. In monitors, `spawn` actions create monitor instances. This figure changes upwards and downwards as monitor instances are spawned, killed or just expire.
`Number of contexts`	The number of contexts in the correlator. This includes the main context plus any user-defined contexts.
`Number of Java applications`	The number of Java applications loaded in the correlator. Java applications do not expire, so this value only decreases when you explicitly unload a Java application.

Status message	Meaning
`Number of listeners`	This is the sum of listeners in all contexts. This includes each `on` statement and each stream source template, for example, `all Tick(symbol="APMA")` in the following: `stream<Tick> ticks := all Tick(symbol="APMA");`
`Number of sub-listeners`	The number of sub-listeners that have been created by listeners across all contexts. A stream source template always has exactly one sub-listener. An `on` statement can have multiple sub-listeners.
`Number of event types`	The total number of event types defined within the correlator. This figure decreases when you delete event types from the correlator.
`Events on input queue`	Total number of events waiting to be processed on all input queues. The main context has its own input queue and any user-defined contexts each have an input queue. This includes private contexts as well as public contexts.
`Events received`	The total number of events ever received by the correlator. A checkpoint preserves this value. If you restore the state of the correlator from a checkpoint file, this number reflects the total number of the events seen by the correlator from which the checkpoint was originally made. Note that if an event is on an input queue, it has been received but not processed.
`Events processed`	The total number of events processed by the correlator in all contexts. This includes external events and events routed to contexts by monitors. An event is considered to have been processed when all listeners and streams that were waiting for it have been triggered, or when it has been determined that there are no listeners for the event.
`Events on internal queue`	Total number of routed events waiting to be processed across all contexts. The internal routing queue in each context is a high priority queue for events that you internally routed with the `route` instruction in EPL. The correlator always processes events on the internal queue before any events on the normal input queue.
`Events routed internally`	The total number of events ever routed internally to the internal queues on this correlator. A checkpoint preserves this value. If you restore the state of a correlator from a checkpoint file, this number reflects the total number of the events routed to the internal queues for the correlator from which the checkpoint was originally made.

Status message	Meaning
`Number of consumers`	The number of event consumers registered with the correlator. Event consumers receive events emitted by the correlator.
`Events on output queue`	The number of events waiting on the correlator's output queue to be dispatched to any registered event consumers.
`Output events created`	The total number of output events created by the correlator. A checkpoint preserves this value. If you restore the state of a correlator from a checkpoint file, this number reflects the total number of output events created by the correlator from which the checkpoint was originally made.
`Output events sent`	The total number of output events that the correlator has sent to event consumers. For example, suppose the correlator created 10 output events and sent each event to two consumers. The number of output events sent is 20. A checkpoint preserves this value. If you restore the state of a correlator from a checkpoint file, this number reflects the total number of output events sent by the correlator from which the checkpoint was originally made.
`Event rate over last interval (ev/s)`	The number of events per second currently being processed by the correlator across all contexts. This value is computed with every status refresh and is only an approximation.
`Events on context input queues`	The total number of events on all context queues in the correlator.
`Most backed up input queue`	The input queue that has the most events waiting to be processed.
`Most backed up queue size`	The number of events on the input queue that has the most events waiting to be processed.
`Slowest receiver`	The receiver with the largest number of incoming events waiting to be processed.
`Slowest receiver queue size`	For the receiver with the largest number of incoming events waiting to be processed, this is the number of events that are waiting.

Options

The tool `engine_watch.exe` (on Windows) or `engine_watch` (on UNIX) takes a number of command line options. These are:

Option	Description
`-h`	Displays usage information. Optional.
`-n host`	Name of the host on which the event correlator is running. Optional. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p port`	Port on which the event correlator is listening. Optional. The default is `15903`.
`-i ms`	Specifies the poll interval in milliseconds. Optional. The default is `1000`.
`-f file`	Writes status output to the named file. Optional. The default is to send status information to `stdout`.
`-r`	Indicates that you want raw output format, which is more suitable for machine parsing. Raw output format consists of a single line for each status message. Each line is a comma-separated list of status numbers. This format can be useful in a test environment. If you do not specify that you want raw output format, the default is a multi-line, human-readable format for each status message.
`-t`	If you also specify the `--raw` option, you can specify the `--title` option so that the output contains headers that make it easy to identify the columns.
`-o`	Outputs one set of status information and then quits. Optional. The default is to indefinitely return status information at the specified poll interval.
`-v`	Displays process names and versions in addition to status information. Optional. The default is to display only status information.
`-V`	Displays version information for the `engine_watch` tool. Optional. The default is that the tool does not output this information.

Exit status

The `engine_watch` tool returns the following exit values:

Status	Description
`0`	All status requests were processed successfully.
`1`	No connection to the event correlator was possible or the connection failed.
`2`	Other error(s) occurred while requesting/processing status.

Correlator Utilities Reference

Inspecting correlator state

The `engine_inspect` tool lets you inspect the state of a running event correlator. This means you can review the applications loaded and running on a correlator. The executable for this tool is located in the `bin` directory of the Apama installation. Running the tool in the Apama command prompt ensures that the environment variables are set correctly.

Synopsis

To inspect applications on a running event correlator:

- On Windows, run `engine_inspect.exe`.
- On UNIX, run `engine_inspect`.

To obtain the following usage message, run the command without any options or with the `-h` option:

```
Usage: engine_inspect [ options ]
To list names of monitors and events in a correlator.
Where options include:
    -m | --monitors            List monitor types
    -j | --java                List Java applications
    -e | --events              List event types
    -t | --timers              List timers
    -x | --contexts            List contexts
    -a | --aggregates          List aggregate functions
    -P | --pluginReceivers     List plugin receivers
    -R | --receivers           List connected receivers
    -r | --raw                 Raw (i.e. parser friendly) output
    -h | --help                This message
    -n | --hostname <host>     Connect to an engine on <host>
    -p | --port <port>         Engine is listening on <port>
    -v | --verbose             Be more verbose
    -V | --version             Print program version info
```

Description

The `engine_inspect` tool retrieves state information from a running event correlator and sends it to `stdout`. By default, the tool outputs information on the monitors, JMon applications, event types and container types currently injected in an event correlator.

You can filter this list by specifying command-line options. When you specify one or more of the `-m`, `-j`, `-e`, `-t`, `-x`, `-P`, or `-R` options, the `engine_inspect` tool displays only the information indicated by the option(s) you specify. See the Options table below for a description of these options.

Options

The `engine_inspect` tool takes the following options.:

Option	Description
`-m`	Displays the names of all EPL monitors in the event correlator and the number of sub-monitors each monitor has spawned.
`-j`	Displays the names of all JMon applications in the event correlator and the number of event listeners each JMon application has created.

Option	Description
`-e`	Displays the names of all event types in the correlator and the number of templates of each type, as defined in listener specifications. This includes each event template in an `on` statement and each stream source template, for example, `stream<A> := all A()`. For more information about event types and listeners, see "Introduction to Apama Event Processing Language" in *Developing Apama Applications* .
`-t`	Displays the current EPL timers active within the system. The four types of timers which may be displayed here are `wait`, `within`, `at`, and `stream`. The `stream` timers are those set up to support the operation of a stream network.
`-x`	Displays the names of any user-defined contexts, how many monitor instances are running in each context, what channels each context is subscribed to, and how many entries are on each context's input queue.
`-a`	Displays a list of the custom (user-defined) aggregate functions that have been injected. You use aggregate functions in stream queries. Apama built-in aggregate functions are not listed.
`-P`	Displays the names of any plug-in receivers, the channels the plug-in is subscribed to, and the number of items on the plug-in's input queue. A plug-in receiver is a correlator plug-in that is subscribed to one or more channels.
`-R`	Displays the names of any external receivers, each receiver's address, the channels each receiver is subscribed to, and the number of entries on each receiver's output queue.

`-r`	Indicates that you want raw output, which is more suitable for machine parsing. Raw output provides the name of each entity in the correlator followed by the number of instances associated with that entity. For a monitor, you get the number of its monitor instances. For a JMon application, you get the number of its listeners. For an event type, you get the number of its templates. For example:
	
	`com.apama.samples.stockwatch.StockWatch 1`
	`Tick 1`

`-h`	Displays usage information.
`-n` *host*	Name of the host on which the event correlator is running. The default is `localhost`. Note: Non-ASCII characters in host names are not supported.
`-p` *port*	Port on which the event correlator is listening. The default is `15903`.
`-v`	Displays process names and versions in addition to application information. Optional. The default is to display only application information.
`-V`	Displays version information for the `engine_inspect` tool.

Exit status

The `engine_inspect` tool returns the following values:

Status	Description
`0`	All status requests were processed successfully.
`1`	No connection to the event correlator was possible or the connection failed.
`2`	Other error(s) occurred while requesting/processing status.

Correlator Utilities Reference

Shutting down and managing components

All Apama components (the correlator, the IAF, and the Sentinel Agent) implement an interface with which they can be asked to shut themselves down, provide their process ID, and respond to communication checks. Running the tool in the Apama Command Prompt ensures that the environment variables are set correctly.

For historical reasons, there are three commands that all do the same thing. You can enter any of these commands to manage any component:

```
engine_management
component_management
iaf_management
```

However, the recommendation is to always use `engine_management`. The only differences in behavior among these commands is as follows:

- `engine_management` and `component_management` default to the local correlator port (`15903`).
- `iaf_management` defaults to the default IAF port (`16903`)

Synopsis

To use the event correlator's management tool:

- On Windows, run `engine_management.exe`.
- On UNIX, run `engine_management`.

When you specify the `-h` command line option, the tool displays the following usage information:

```
Usage: engine_management [ options ]
Where options include:
 -V | --version             Print program version info
 -h | --help                Display this message
 -v | --verbose             Be more verbose
 -n | --hostname <host>     Connect to a component on <host>
 -p | --port <port>         Component is listening on <port>
 -w | --wait                Wait forever for component to start
 -W | --waitFor <num>       Wait <num> seconds for component to start
 -N | --getname             Get the name of the component
 -T | --gettype             Get the type of the component
 -Y | --getphysical         Get the physical ID of the component
 -L | --getlogical          Get the logical ID of the component
 -O | --getloglevel         Get the log level of the component
```

```
-C | --getversion          Get the version of the component
-R | --getproduct          Get the product version of the component
-B | --getbuild            Get the build number of the component
-F | --getplatform         Get the build platform of the component
-P | --getpid              Get the process ID of the component
-H | --gethostname         Get the hostname of the component
-U | --getusername         Get the username of the component
-D | --getdirectory        Get the working (current) directory of the component
-E | --getport             Get the port of the component
-c | --getconnections      Get all the connections to the component
-a | --getall              Get all of the above values
-xs| --disconnectsender <id> <reason> Disconnect sender with physical id <id>
-xr| --disconnectreceiver <id> <reason>Disconnect receiver with physical id <id>
-I | --getinfo <category>  Get component-specific info for <category>
                          Use empty string to get all available categories
                          Multiple -I options may be specified
-d | --deepping            Deep-ping the component
-l | --setloglevel <level> Set logging verbosity to <level>. Available levels
                          are TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT and OFF
-r | --dorequest <req>     Send component-specific request <req>
-s | --shutdown <why>      Shutdown the component with reason <why>
```

Description

Use the `engine_management` tool to connect to a running component. Once connected, the tool can shut down the component or return information about the component. The `engine_management` tool can connect to any of the following types of components. The `engine_management` tool sends output to `stdout`.

- Event correlator
- Adapter
- Sentinel Agent process

Options

When you run the `engine_management` tool, you can specify any of the options described in the following table. By default, the `engine_management` tool connects to a local correlator that is listening on port `15903` (the correlator default port). To obtain all information for a particular component, specify the `-a` option. All options are optional:

Option	Description
`-V`	Displays version information for the `engine_management` tool.
`-h`	Display usage information.
`-v`	Displays information in a more verbose manner. For example, when you specify the `-v` option, the `engine_management` tool displays status messages that indicate that it is trying to connect to the component, has connected to the component, is disconnecting, is disconnected, and so on. If you have having trouble obtaining the information you want, specify the `-v` option to help determine where the problem is.
`-n` *`host`*	Name of the host on which the component is running (default is `localhost`). Non-ASCII characters are not allowed in host names.

Option	Description
`-p port`	Port on which the component you want to connect to is listening The default is `15903`.
`-w`	Instructs the `engine_management` tool to wait for the component to start and be in a state that is ready to receive EPL files. This option is similar to the `-W` option, except that this option (the `-w` option) instructs the tool to wait forever. The `-W` option lets you specify how many seconds to wait. See the information for the `-W` option for an example.

`-W num`	Instructs the `engine_management` tool to wait *num* seconds for the component to start and be in a state that is ready to receive EPL files. If the component is not ready to receive EPL files before the specified number of seconds has elapsed, the `engine_management` tool terminates with an exit code of `1`.
	
	This option is most useful in scripts, when the component you want to operate on has not yet started. For example, suppose a script specifies the following commands:
	
	`correlator.exe options engine_inject some_EPL_files`
	
	It can sometimes take a few seconds for a component to start, and this number of seconds is not always exactly predictable. If the `engine_inject` tool runs before the correlator is ready to receive EPL files, the `engine_inject` tool fails. To avoid this for a local correlator that is listening on the default port, insert the following command between these commands:
	
	`engine_management -W 10`
	
	This lets the `engine_management` tool wait for up to 10 seconds for the correlator's management interface to be available. To set an appropriate wait time for your application, monitor your application's performance and adjust as needed.

`-N`	Displays the name of the component. For example, when you start a correlator, you can give it a name with the `-N` option. This is the name that the `engine_management` tool returns. If you do not assign a name to a correlator when you start it, the default name is `correlator`.
`-T`	Displays the type of the component that the `engine_management` tool connects to. The returned value is one of the following: `correlator`, `iaf`, `sentinel_agent`. If you see that a port is in use, you can specify this option to determine the type of component that is using that port.
`-Y`	Displays the physical ID of the component. This can be useful if you are looking at status log information that identifies components by their physical IDs.
`-L`	Displays the logical ID of the component. This can be useful if you are looking at status log information that identifies components by their logical IDs.
`-O`	Displays the log level of the component. The returned value is one of the following: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `CRIT`, `FATAL`, or `OFF`.

Option	Description
`-C`	Displays the version of the component. For example, when the tool connects to a correlator, it displays the version of the correlator software that is running.
`-R`	Displays the product version of the component. For example, when the tool connects to a correlator, it displays the version of the UNIX software that is running.
`-B`	Displays the build number of the component. This information is helpful if you need technical support. It indicates the exact software contained by the component you connected to
`-F`	Displays the build platform of the component. This information is helpful if you need technical support. It indicates the set of libraries required by the component you connected to
`-P`	Displays the process ID of the correlator you are connecting to. This can be useful if you are looking at log information that identifies components by their process ID.
`-H`	Displays the host name of the component. When debugging connectivity issues, this option is helpful for obtaining the host name of a component that is running behind a proxy or on a multihomed system.
`-U`	Displays the user name of the component. On a multiuser machine, this is useful for determining who owns a component.
`-D`	Displays the working (current) directory of the component. This can be helpful if a plug-in writes a file in a component's working directory.
`-E`	Displays the port of the component.
`-c`	This option is for use by technical support. It displays all the connections to the component
`-a`	Displays all information for the component.
`-xs`	Disconnects the sender that has the physical ID you specify. If you specify a reason, the `engine_management` tool sends the reason to the correlator. The correlator then logs the message, sends the reason to the sender, and disconnects the sender. You can specify the component ID as *`physical_ID/logical_ID`*.
`-xr`	Disconnects the receiver that has the physical ID you specify. If you specify a reason, the `engine_management` tool sends the reason to the correlator. The correlator then logs the message, sends the reason to the receiver, and disconnects the receiver. You can specify the component ID as *`physical_ID/logical_ID`*.
`-I` *`category`*	This option is for use by technical support. It displays component-specific information for the specified category.

Option	Description
`-d`	Ping the component. This confirms that the component process is running and acknowledging communications.
`-l level`	Sets the amount of information that the component logs. In order of decreasing verbosity, you can specify `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CRIT`, or `OFF`.
`-r`	This option sends a component-specific request. For example: `engine_management -r "profiling frequency"` This returns the profiling frequency in Hertz. The following requests are available: • `applicationEventLogging` — Sends detailed application information to the correlator log file. See "Viewing garbage collection information" on page 143. • `flushAllQueues` — Sends a request into the correlator that waits until every event/injection sent or enqueued to a context before the `flushAllQueues` request started has been processed, and every event emitted as a result of those events has been acknowledged. This may block if a slow receiver is connected to the correlator. Events enqueued to a context after the request has started may or may not be processed — thus if you want to see the results of one context enqueueing to a second, which enqueues to a third, you should execute `engine_management -r "flushAllQueues"` three times, to ensure it has been processed by each context. This does not change the behavior of the correlator (the correlator will always flush all queues as soon as it is able to), it just waits for events currently on input queues to complete. • `profiling` — Lets you profile Apama EPL applications. See "Using the profiler command-line interface" on page 144. • `rotateLogs` — For details on this request, see "Rotating all log files" on page 150. • `setApplicationLogFile` — A set of commands lets you set, get, and unset the log files for packages and monitors. See "Setting logging attributes for packages, monitors and events" on page 147. • `setApplicationLogLevel` — A set of commands lets you set, get, and unset logging levels for packages and monitors. See "Setting logging attributes for packages, monitors and events" on page 147. • `setLogFile` — Instructs the component to close the log file it is using and open a new log file with the name you specify. See "Rotating specified log files" on page 151. • `verbosegc` — Enables logging of garbage collection events. See "Viewing garbage collection information" on page 143. Certain other requests for this option are available for use by Apama technical support.

Option	Description
`-s why`	Instructs the component to shut down and specifies a message that indicates the reason for termination. The component inserts the string you specify in its status log file with a `CRIT` flag, and then shuts down.

Exit status

The following exit values are returned:

Status	Description
`0`	All status requests were processed successfully.
`1`	Indicates one of the following: • No connection to the specified component was possible. • The connection failed. • You specified the `waitFor` option and the specified time elapsed without the component starting.
`2`	One or more errors occurred while requesting/processing status.
`3`	Deep ping failed.

Correlator Utilities Reference

Viewing garbage collection information

A handy way to view garbage collection information is to execute the following command:

```
engine_management -r "verbosegc on"
```

This command enables logging of garbage collection events, and is particularly useful in production environments. The additional garbage collection information goes to the correlator log. To disable logging of garbage collection information, execute the following:

```
engine_management -r "verbosegc off"
```

These commands provide an alternative to the following command, which provides a great deal of detailed output in addition to garbage collection information. Again, this output goes to the correlator log.

```
engine_management -r "applicationEventLogging on"
```

To turn this off:

```
engine_management -r "applicationEventLogging off"
```

Shutting down and managing components

Using the profiler command-line interface

You can profile applications written with EPL in Apama Studio. Data collected in the profiler allows you to identify possible bottlenecks in an EPL application. When testing an application, or after you deploy an application, you might find it handy to write a script that includes obtaining profile information. Or, you might want to obtain profile information without the overhead of Apama Studio. In these situations, you can use the command-line interface to the profiler, which is described here.

The command-line interface to the profiler consists of specifying the `-r | --dorequest` option with the `engine_management` utility:

```
engine_management -r "profiling on"
engine_management -r "profiling off"
engine_management -r "profiling get"
engine_management -r "profiling gettotal"
engine_management -r "profiling reset"
engine_management -r "profiling frequency"
```

`on`	Starts to capture the state of all contexts in the correlator.
`off`	Stops capturing profile data.
`get`	Returns the samples collected since the correlator was started or since the profiler was reset. Returned data is in CSV (comma separated values) format. A sample is the state of the correlator at the moment the profiler collects data.
`gettotal`	Returns totals for all contexts.
`reset`	Clears profiling samples collected.
`frequency`	Returns the profiling frequency in Hertz.

If a context is executing, it is typically in the EPL interpreter. However, it might also be doing something such as matching events or collecting garbage. For EPL execution, there is a call stack for each context. For the purposes of the profiler, there is one entry at the top for the monitor name, then comes the listener/`onload` action, and then any actions that is calling, and so on. The only action that the correlator is actually executing is at the bottom of the stack.

A context can be in one or two of the following states:

- CPU — the correlator is executing code in this context.
- Runnable — the correlator has work to do in this context but it has been rescheduled because the correlator is executing code in another context.
- Idle —the correlator has no work to do in this context.
- Non-Idle — the correlator has work to do in this context. When a context is in this state, it is also in one other state, either CPU, Plugin, Blocked, or Runnable).
- Plugin — the correlator is executing a plug-in in this context.

- Blocked — the correlator cannot make progress in this context. It is blocked because of a full queue. The full queue might be the correlator output queue (the context is trying to emit an event) or another context's input queue.

When the profiler takes a sample it examines every context in the correlator. Every entry in each context's call stack results in addition or modification of a line in the profiler output. The Cumulative column is incremented for all samples, and one or more of the other columns is incremented for the lowest (deepest) call stack element according to what states the context is in.

When the correlator is not executing EPL code, there is only one element in the stack, for example, when the correlator is processing an event.

The profiler's resolution is to a EPL action. That is, the profiler does not distinguish between lines within an action. The line number in the output is the first line of the action that generates code. For example, variable declarations without initializers, and comments do not generate code, while statements, and declarations with initializers, do generate code. The profiler treats the body of a listener (the code the correlator executes when the listener fires) as an action with the name `::listenerAction::.`

If you want to profile parts of a single large action, you need to split the action into multiple actions in order to determine where time is spent. Remember that action calls have some cost, so that could skew the results.

The `"profiling get"` or `"profiling gettotal"` request returns samples to `stdout` as lines of comma separated values. Output is sorted by context and then by CPU time. For example:

```
Context Id,Context Name,Location,Filename and line number,Cumulative time,CPU time,
Empty,Non-Idle,Idle,Runnable,Plugin,Blocked,total ticks:573
3,3,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/
create-state.mon:50,556,293,0,556,0,0,263,0
```

In the previous output, nearly all of the time of this context (3) is spent in the listener that starts on line 50 of `create-state.mon`. The time is spread between executing EPL code (293 samples) and executing a plug-in (263 samples). Each context spent similar amounts of time executing EPL and executing plug-ins but in different listeners (notice the different line numbers).

Here is more sample output:

```
3,3,Idle,,1,1,0,1,14,0,0,0
3,3,Only just started profiling,,0,0,0,0,2,0,0,0
3,3,Monitor:processor,,556,0,0,0,0,0,0,0
2,2,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/create-state.mon:
34,556,261,0,556,0,0,295,0
2,2,Idle,,1,1,0,1,14,0,0,0
2,2,Only just started profiling,,0,0,0,0,2,0,0,0
2,2,Monitor:processor,,556,0,0,0,0,0,0,0
4,4,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/create-state.mon:
65,556,296,0,556,0,0,260,0
4,4,Idle,,1,1,0,1,14,0,0,0
4,4,Only just started profiling,,0,0,0,0,2,0,0,0
4,4,Monitor:processor,,556,0,0,0,0,0,0,0
1,main,processor:processor.::listenerAction::,/users/ukcam/cr/dev/4.3.0.0/apama-test/
system/correlator/corba/testcases/correctness/Corr_Corba_cor_543/Input/create-state.mon:
18,556,283,0,556,0,0,273,0
1,main,Idle,,1,1,0,1,14,0,0,0
1,main,Only just started profiling,,0,0,0,0,2,0,0,0
1,main,Monitor:processor,,556,0,0,0,0,0,0,0
```

This output is intended to be imported to a spreadsheet, such as Excel. If you do that, then the values in one sample (one row) provide the following information in the following order:

Column Content	Description
Context ID	ID of the context. A context ID is not present in data returned by `-r "profiler gettotal"`.
Context Name	Name of the context. A context name is not present in data returned by `-r "profiler gettotal"`.
Location	What the correlator is doing or where the correlator is executing code at the moment the sample was collected. The value is one of the following: • `Monitor:`*`monitor_name`* — The top-level entry for the monitor. • *`monitor_name.code_owner.action_name`* — For example, if monitor `monny` calls an action `act` on event `pkg.evie`, this location would be `monny.pkg.evie.act`. If a listener has been triggered, the action name is always `::listenerAction::`. • *`monitor_name`*`.;GC` — Garbage collection. • `Event:`*`event_name`* — Event matching or chastenment of an event of that type • `Idle` — Correlator has no work to do. • There are other possible values that you might rarely see. They are self explanatory.
Filename and line number	If the correlator is executing EPL code, indicates the filename and line number of the beginning of the action that is executing.
Cumulative time	Cumulative time indicates time spent in this location or in something that this location was calling (directly or indirectly). CPU time shows time spent in this location, not the actions it called.I
CPU time	Number of samples in which the correlator is executing the location/action and is not in a plug-in (see Plugin later in this table). CPU time is a subset of Cumulative time. It does not include time spent in the location(s) called by this location.
Empty	Number of samples in which the context was empty.An empty context should happen very rarely. A context might be empty if there is a race between getting the location and the state.
Non-idle	Number of samples in which the context was at this row's location and not idle. Each sample in this count is also in the count for CPU time, Runnable, Plugin, or Blocked.

Column Content	Description
Idle	Number of samples in which the context was idle. This should correspond to a location of Idle or Only just started profiling, which means it is an unknown state. As with other cumulative counters, races can result in misleading results. For example, `Idle` in an action, but those are best ignored and should be small.
Runnable	Number of samples in which the location was the lowest point on the call stack and the context was runnable. Runnable means it could have made progress, but the scheduler determined that the correlator should run something else instead. When all rows contain `0` for this entry it means that the correlator never (or very rarely) had to re-schedule one context to run another context. A non-zero value means this location was running for a long time, and it was suspended so that other contexts could run
Plugin	Number of samples in which the location is executing a correlator plug-in.
Blocked	Number of samples in which the context was unable to make progress. For example, it was trying to emit an event but the correlator output queue was full, or it was trying to enqueue an event to a particular context but that context's input queue was full.

Shutting down and managing components

Setting logging attributes for packages, monitors and events

You can configure per-package logging in two ways:

- Statically, in the extended configuration file when starting the correlator. See "Setting log files and log levels in an extended configuration file" on page 171.
- Dynamically, using the `engine_management setApplicationLogFile/Level` request, described here.

In EPL code, you can specify `log` statements as a development or debug tool. By default, `log` statements that you specify in EPL send information to the correlator log file. If a log file was not specified when the correlator was started, and you have not executed the `engine_management` utility to associate a log file with the correlator, `log` statements send output to `stdout`.

In place of this default behavior, you can specify different log files for individual packages, monitors and events. This can be helpful during development. For example, you can specify a separate log file for a package or monitor you are implementing, and direct log output from only your development code to that file.

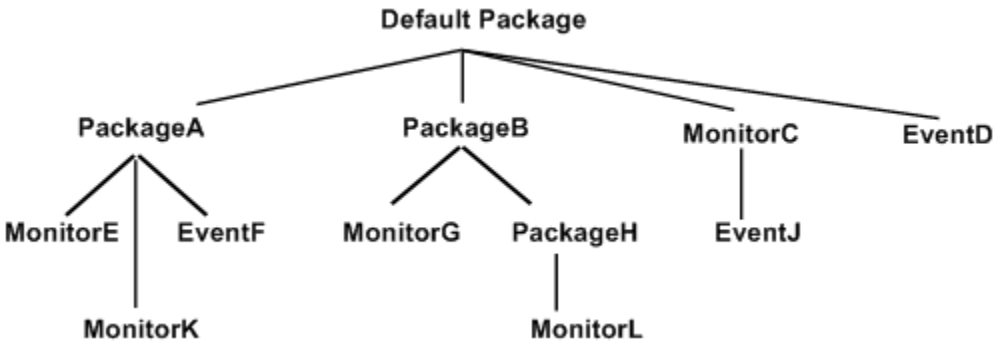
Also, you can specify a particular log level for a package, monitor, or event. The settings of log files and log levels are independent of each other. That is, you can set only a log level for a particular package, monitor or event, or you can set only a log level for a particular element. The following topics provide information for managing individual log files and log levels.

- "Tree structure of packages, monitors, and events" on page 148
- "Managing application log levels" on page 148
- "Managing application log files" on page 149

See also "Rotating the correlator log file" on page 150.

Tree structure of packages, monitors, and events

Packages, monitors and events form a tree as illustrated in the figure below. For each node in the tree, you can specify a log file and/or a log level. Nodes for which you do not specify log settings inherit log settings from their parent node.



The root of the tree is the default package, which contains code that does not explicitly specify a package with the `package` statement. Specified packages are intermediate nodes. Packages can nest inside each other. Monitors and events in specified packages are leaf nodes. If you specify an event type in a monitor, that event is a leaf node and its containing monitor is an intermediate node.

For example, suppose you specify `packageA.log` as the log file for `packageA`. The `packageA.log` file receives output from `log` statements in `MonitorE` and `MonitorK`. If `EventF` contains any `action` members that specify `log` statements, output would go to the `packageA.log` file.

Now suppose that you set `ERROR` as the log level for the default package and you set `INFO` as the log level for `PackageB`. For `log` statements in `MonitorG`, `PackageH`, and `MonitorL`, the correlator compares the log statement's log level with `INFO`. For `log` statements in the rest of the tree, the correlator compares the `log` statement's log level with `ERROR`. For details, see the table in "Managing application log levels" on page 148.

Managing application log levels

To set the log level for a package, monitor or event, invoke the `engine_management` utility as follows:

```
engine_management -r "setApplicationLogLevel logLevel [node]"
```

logLevel	Specify `OFF`, `CRIT`, `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
node	Optionally, specify the name of a package, monitor or event. If you do not specify a node name, the utility sets the log level for the default package.

To obtain the log level for a particular node, invoke the utility as follows:

```
engine_management -r "getApplicationLogLevel [node]"
```

If you do not specify a node, the utility returns the log level for the default package. To remove the log level for a node, so that it takes on the log level of its parent node, invoke the utility as follows. Again, if you do not specify a node, you remove the log level for the default package. The default package then takes on the log level in effect for the correlator. The default correlator log level is `INFO`.

```
engine_management -r "unsetApplicationLogLevel [node]"
```

To manage the log level for an event that you define in a monitor, see "Managing event logging attributes" on page 150.

After the correlator identifies the applicable log level, the log level itself determines whether the correlator sends the `log` statement output to the appropriate log file. The following table indicates which log level identifiers cause the correlator to send the log statement to the appropriate log file.

Log Level in Effect	Log Statements With These Identifiers Go to the Appropriate Log File	Log Statements With These Identifiers are Ignored
`OFF`	None	`CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE`
`CRIT`	`CRIT`	`FATAL, ERROR, WARN, INFO, DEBUG, TRACE`
`FATAL`	`CRIT, FATAL`	`ERROR, WARN, INFO, DEBUG, TRACE`
`ERROR`	`CRIT, FATAL, ERROR`	`WARN, INFO, DEBUG, TRACE`
`WARN`	`CRIT, FATAL, ERROR, WARN`	`INFO, DEBUG, TRACE`
`INFO`	`CRIT, FATAL, ERROR, WARN, INFO`	`DEBUG, TRACE`
`DEBUG`	`CRIT, FATAL, ERROR, WARN, INFO, DEBUG`	`TRACE`
`TRACE`	`CRIT, FATAL, ERROR, WARN, INFO, DEBUG, TRACE`	None

See also "Log levels determine results of log-statements" in the section "Defining what happens when matching events are found" in *Developing Apama Applications*.

Managing application log files

To specify a log file for a package, monitor or event, invoke the `engine_management` utility as follows:

```
engine_management -r "setApplicationLogFile logFile [node]"
```

`logFile`	Specify the path of the log file. You specify the name of an application log file in the same way that you specify the name of a correlator status file or input file. See "Specifying log filenames" on page 111.
`node`	Optionally, specify the name of a package, monitor or event. If you do not specify a node name, the utility associates the log file with the default package.

To obtain the path of the log file for a particular node, invoke the utility as follows:

```
engine_management -r "getApplicationLogFile [node]"
```

If you do not specify a node, the utility returns the log file for the default package. To disassociate a log file from its node, so that the node uses the log file of its parent node, invoke the utility as follows. Again, if you do not specify a node, you disassociate the log file from the default package. The correlator log file is then in effect for the default package. If a log file has not been specified for the correlator, the default is `stdout`.

```
engine_management -r "unsetApplicationLogFile [node]"
```

Managing event logging attributes

If you specify an event type in a monitor, that event does not inherit the logging configuration from the enclosing monitor. It is expected that this will change in a future release. To explicitly set logging attributes for an event type defined in a monitor, invoke the `engine_management` utility and specify an unqualified event type name. Do not specify an enclosing scope, such as `com.apamax.myMonitor.NestedEventType`. For example:

```
engine_management -r "setApplicationLogFile foo.log NestedEventType"
engine_management -r "setApplicationLogLevel DEBUG NestedEventType"
```

Shutting down and managing components

Rotating the correlator log file

Rotating the correlator log file refers to closing the status log file of a running correlator and opening a new status log file. This lets you archive log files and avoid log files that are too large to easily view.

Each site should decide on and implement its own correlator log rotation policy. You should consider

- How often to rotate log files
- How large a correlator log file can be
- What correlator log file naming conventions to use to organize log files.

  There is a lot of useful header information in the log file being used when the correlator starts. If you need to provide log files to Apama technical support, you should be able to provide the log file that was in use when the correlator started, as well as any other log files that were in use before and when a problem occurred.

To rotate the correlator log file and also rotate any other log file the correlator is using (input log file, application log files), see "Rotating all log files" on page 150.

To rotate only the correlator log file, see "Rotating specified log files" on page 151.

Shutting down and managing components

Rotating all log files

To invoke rotation of all log files that the correlator is using, you can do the following:

- Invoke the `engine_management` utility and specify `-r rotateLogs`.

This rotates the correlator status log, the sentinel agent status log, the correlator input log if it is being generated, and any application logs that are being generated. When you invoke this management request then the correlator closes each log file it was using.

If the log filename specification declared `${START_TIME}`, `${ROTATION_TIME}` and/or `${ID}` then the correlator starts new log files with updated names according to the log filename specification, for example, if `${ID}`was specified then the ID portion of a log filename would be incremented by 1.

- In EPL, create a monitor that uses the Management interface correlator plug-in to trigger log rotation on a schedule. See .
- On UNIX only, you can write a `cron` job that periodically sends a `SIGHUP` signal to Apama processes.

  The standard UNIX `SIGHUP` mechanism causes Apama processes to re-open their log files. If the log file names were specified with the `${ROTATION_TIME}` and/or `${ID}` then the re-opened files have names that contain the rotation time and/or the incremented ID.

If you want a log filename to always be the same and so did not declare `${START_TIME}`, `${ROTATION_TIME}` or `${ID}` in the log filename specification then the correlator starts new log files that have the same names as the log files it closed. On Windows, this would overwrite the closed log files so you must move the log files before invoking rotation. On UNIX, log files are appended to if the names are the same.

Shutting down and managing components

Rotating specified log files

Run one of the following utilities to rotate a particular log file. On Windows, set up scheduled tasks that run the utilities. On UNIX, write a `cron` job that periodically runs the utilities. The behavior is the same on both Windows and UNIX, except as noted. The only way to rotate the correlator input log is to rotate all log files. See "Rotating all log files" on page 150.

- The following command instructs the correlator to close its status log file and start using a status log file that has the name you specify. Be sure to enclose the argument after `-r` in quotation marks.

  ```
  engine_management -r "setLogFile log-filename"
  ```

- The following command instructs the correlator to use the specified file as the log file for the specified node, which can be a package, monitor, or event. See also "Setting logging attributes for packages, monitors and events" on page 147.

  ```
  engine_management -r "setApplicationLogFile log-filename [node]"
  ```

  If you use separate log files for particular packages, monitors, or events you might want to rotate those logs at the same time that you rotate the correlator status log. This keeps your Apama log files in sync with each other. See "Rotating all log files" on page 150.

On Windows, when you rotate a log file you must ensure that the new log filename is different from the name of the log file that was in use. Apama takes care of this for you if you specify `${ROTATION_TIME}` and/or `${ID}` in the *log-filename* specification. If the name is not different, the old file is overwritten. If you want to use the same log filename then you must move the file before you rotate it.

On UNIX, a log file is never overwritten. If you rotate a log file and specify the same name then Apama appends messages to the content already there.

Apama does not support automatic log file rotation based on log file size.

The only way to rotate the correlator input log is to rotate all log files. See "Rotating all log files" on page 150.

Shutting down and managing components

Using the command-line debugger

The `engine_debug` tool lets you control execution of EPL code in the correlator and inspect correlator state. This tool is a correlator client that runs a single command from the command line. It is not an interactive command-line debugger.

In general, this tool is expected to be most useful when you are ready to deploy your application or after deployment. During development, the interactive debugger in Apama Studio will probably be most useful to you.

Before you run the `engine_debug` tool, specify the `-g` option when you start the correlator. Specification of this option disables some correlator optimizations. If you run the `engine_debug` utility and you did not specify the `-g` option when you started the correlator, the optimizations hinder the debugging process. For example, the correlator might simultaneously execute multiple statements over multiple lines even if you are using debugger commands to step through the program line by line.

Information about the command-line debugger is organized as follows:

Synopsis

To debug applications on a running event correlator:

- On Windows, run `engine_debug.exe`.
- On UNIX, run `engine_debug`.

To obtain a usage message, run the command with the `help` option.

Description

Debugging a running correlator has some effect on the other programs that connect to that correlator. While you pause a correlator, the expected behavior of connected components is as follows:

- Sending events to the correlator continues to put events on the input queue of each public context. However, since the input queues are not being drained, if an input queue fills up, this will block senders, including the `engine_send` tool and adapters.
- The correlator sends out any events on its output queue. When the output queue is empty, receivers no longer receive events; no contexts are sending events.
- Other inspections of the correlator proceed as normal. For example, `engine_watch`, `engine_management`, and profiling data.
- You can shut down the correlator.
- You can inject monitors while the correlator is stopped. They will not run any of the `onload()` or similar code until the correlator resumes, but the inject call should succeed.
- Java applications continue to run completely independently of whether the correlator is stopped.
- All other requests block until the correlator resumes processing. This includes dumping correlator state, loading, and changing debug or profiling state.

The `engine_debug` tool is stateless. Consequently, during debugging, you can have multiple concurrent connections to the same correlator.

Debug commands

The ordering of arguments to `engine_debug` commands works as follows:

- All arguments before the first command apply to all commands in that command line. This is useful for setting the host and port if you are not using the local defaults.
- All arguments following a command apply to only that command and they override any applicable arguments specified before the first command.
- The arguments to a particular command can be in any order
- When there are multiple commands in a line, the debugger executes them in the order in which they are specified. Execution continues until either all complete, or one fails, which prevents execution of any subsequent commands.

The `engine_debug` tool takes the following commands as options:

Abbv.	Command	Description
	`help [command]`	Displays a usage message. To obtain help for a particular `engine_debug` command, specify that command.
`p`	`status`	Displays the current debugger state, and position if stopped.
`si`	`stepinto`	Steps into an action.

Abbv.	Command	Description
`sot`	`stepout`	Steps out of an action.
`sov`	`stepover`	Steps over an instruction.
`r`	`run`	Begins processing instructions.
`b`	`stop`	Stops processing instructions.
`w [-to int]`	`wait [--timeout timeout]`	Waits for the correlator to stop processing instructions. Specify an integer that indicates the number of seconds to wait. The debugger waits forever if you do not specify a timeout. See "The wait command" on page 158 for more information.
`s`	`stack [--context contextid] | [--frame frameid]`	Displays current stack information for all contexts. The output includes the frame ID associated with each variable. To display stack information for only a particular context, specify the `--context` argument. To display stack information for only a particular frame, specify the `--frame` argument.
`i`	`inspect --instance monitorinstance | --instance monitorinstance --frame frameid | --instance monitorinstance --variable variablename | --instance monitorinstance --frame frameid --variable variablename | --frame frameid | --frame frameid --variable variablename`	Displays the value of one or more variables. Specify a monitor instance and/or a frame ID and/or a variable name to display a list of variables in that monitor or in a particular monitor frame, or to display the value of a particular variable. Obtain monitor instance IDs from `engine_inspect` output or correlator log statements. Obtain frame IDs from `engine_inspect stack` output.
`c`	`context [--context contextid]`	Displays information about all contexts in the correlator or about only the context you specify. Information displayed includes context name, context ID, monitor instances in the context, and monitor instance IDs.
`e`	`enable`	Enables debugging. You must run this in order to do any debugging.
`d`	`disable`	Disables debugging. You must run this to disable debugging. If you do not disable debugging, the correlator runs more

Abbv.	Command	Description
		slowly and continues to stop when it hits breakpoints.
`boe`	`breakonerror enable`	Causes the debugger to pause if it encounters an error.
`boeoff`	`breakonerror disable`	Causes the debugger to continue processing if it encounters an error.

`ba`	`breakpoint add`
	`[--breakonce] --file filename`
	`--line linenumber |`
	`[--breakonce] --owner ownername`
	`--action actionname`
	`--line linenumber`
		Adds a breakpoint at the beginning of the specified line. If you do not specify `--breakonce`, the correlator always pauses at this point when debugging is enabled. You must specify the line number where you want the breakpoint. As usual, this is the absolute offset from the beginning of the file. You must specify either the name of the file that contains the breakpoint or the owner and action name that contains the breakpoint. When the owner is a monitor instance, specify *`package_name.monitor_name`* or just *`monitor_name`* if there is no package.

`bd`	`breakpoint delete`
	`--file filename`
	`--line linenumber |`
	`--owner ownername`
	`--action actionname`
	`--line linenumber |`
	`--breakpoint breakpointid`
		Removes a breakpoint. Specify one of the following:
		• File name and line number
		• Owner name, action name and line number. When the owner is a monitor instance, specify *package_name*.*monitor_name* or just *monitor_name* if there is no package.
		• Breakpoint ID. You can obtain a breakpoint ID by executing the `breakpoint list` command.

`bls`	`breakpoint list`	For each breakpoint in the correlator, this displays the following: • Breakpoint ID • Name of file that contains the breakpoint • Name of the action that contains the breakpoint • Name of the owner of the breakpoint • Number of the line that the breakpoint is on.

Abbv.	Command	Description
		The breakpoint owner is the name of the monitor that contains the breakpoint or the name of the event type definition that contains the breakpoint. If the breakpoint is in an event type definition, the definition must specify an action and processing must create a closure between an event instance and an action call. For information about closures, see "Using action type variables" in *Developing Apama Applications*.

Exit status

The `engine_debug` tool returns the following values:

Status	Description
`0`	Success — All requests were processed successfully.
`1`	Failure — The correlator could not parse the command line, or an exception occurred, such as losing a connection or trying to use a non-existent ID.

Correlator Utilities Reference

Obtaining online help for the command-line debugger

The command-line debugger provides online help. To obtain general information, enter the following:

```
engine_debug help
```

To get help for a particular command, specify that command after the `help` keyword. For example:

```
engine_debug help status
status: Displays the current debugger state, and position if stopped
engine_debug help breakpoint add
breakpoint add [--breakonce] --line linenumber [--file filename |
--owner ownername --action actionname]:
Add a breakpoint at the specified location
```

Using the command-line debugger

Enabling and disabling debugging in the correlator

To use the debugger, you must enable debugging in the correlator. To enable debugging locally on the default port, enter the following:

```
engine_debug enable
```

When you are done debugging, you should disable debugging in the correlator. If you do not, the correlator runs more slowly and continues to pause when it hits a breakpoint. To disable debugging in the local correlator on the default port, enter the following:

```
engine_debug disable
```

You can also use the debugger in a remote correlator by specifying the host name and the port number. For example:

```
engine_debug enable --host foo.bar.com --port 1234
engine_debug disable --host foo.bar.com --port 1234
```

Using the command-line debugger

Working with breakpoints using the command-line debugger

You can use the command-line debugger for:

- "Adding breakpoints" on page 157
- "Listing breakpoints" on page 157
- "Removing breakpoints" on page 158

Adding breakpoints

There are two ways to add a breakpoint. If you know the EPL file name and the line number you can enter something like the following:

```
engine_debug breakpoint add --file filename.mon --line 27
```

When you specify a file name you must specify the exact path you specified when you injected the monitor. For example, suppose you ran the following:

```
engine_inject foo.mon
```

You can then specify `foo.mon` for the file name. Now suppose you ran this:

```
engine_inject c:\foo\bar\baz.mon
```

You must then specify `c:\foo\bar\baz.mon` for the file name.

If you prefer to use the monitor and action name, along with the line number, enter something like this:

```
engine_debug breakpoint add --monitor package.monitor --action actionName --line 27
```

The debugger output indicates the line number where it added the breakpoint. In some cases, the debugger does not set the breakpoint on the line you specified. For example, when a statement runs over multiple lines.

Listing breakpoints

To obtain a list of the breakpoints currently set in the correlator, enter the following:

```
engine_debug breakpoint list
```

Removing breakpoints

To remove a breakpoint by specifying the file name and the line number, enter something like the following:

```
engine_debug breakpoint delete --file filename.mon --line 27
```

To use the monitor name to remove a breakpoint, enter something like this:

```
engine_debug breakpoint delete --monitor package.monitor --action actionName --line 27
```

To delete a breakpoint by using the breakpoint ID that appears in the breakpoint list returned by the debugger, enter something like this:

```
engine_debug breakpoint delete --breakpoint 1
```

Controlling execution with the command-line debugger

When the correlator stops at a breakpoint, you can use the debugger to step over the next line:

```
engine_debug stepover
```

However, you most likely want to step over the line, confirm that the correlator stopped, and learn about the current state of the debugger. You can do this by entering multiple commands in one line. For example:

```
engine_debug stepover wait --timeout 10 status
```

This is the equivalent of the following three commands:

- `engine_debug stepover` — Causes the debugger to step over one line of EPL.
- `engine_debug wait --timeout 10` — Causes the debugger to pause until either a breakpoint is hit, or ten seconds pass.
- `engine_debug status` — Displays the debugger's current status.

Following are more examples of entering multiple commands in one line.

```
engine_debug stepinto wait --timeout 10 status
engine_debug stepout wait --timeout 10 status
```

To instruct the correlator to continue executing EPL code, run the following command:

```
engine_debug run
```

You use the `engine_debug run` command regardless of how the correlator was stopped — a breakpoint was reached, a step operation, a `wait` command.

To stop the correlator, enter the following command:

```
engine_debug stop
```

Using the command-line debugger

The wait command

The `wait` command connects to the correlator to determine if the correlator has suspended processing. If the correlator is in suspend mode, the `wait` command returns immediately and debugging continues. If the correlator is not in suspend mode, the `wait` command remains connected to the correlator. The `wait` command returns when something else suspends the correlator or when the timeout is reached. Operations that can suspend the correlator include reaching a breakpoint,

stepping into or over a line, or some other client explicitly stopping the correlator. If the `wait` command reaches the timeout, it suspends the correlator before it returns.

Stepping can take a variable amount of time. For example, suppose the debugger stops at the end of a listener and you execute a step command. The debugger is now outside the flow of execution until another event comes in. The time that the debugger has to wait for the step to finish is dependent upon when the next matching event arrives.

Controlling execution with the command-line debugger

Command shortcuts for the command-line debugger

Putting multiple commands in the same command line can get verbose. For example, suppose you want to step out of an action on a remote machine. You would need to enter something like this:

```
engine_debug stepout --host foo.bar.com --port 1234 wait --timeout 10
   --host foo.bar.com --port 1234 status --host foo.bar.com --port 1234
```

The command-line debugger provides easier ways to invoke this.

- Any arguments that you specify before the first debugging command apply to the entire command line.
- All individual commands and their arguments have abbreviations.

For example, the following command does the same thing as the previous verbose command:

```
engine_debug -h foo.bar.com -p 1234 sot w -to 10 p
```

The following table lists the abbreviations you can use for command arguments. For abbreviations of commands, see "Debug commands" on page 153

Command	Abbreviation
`--action`	`-a`
`--breakonce`	`-bo`
`--breakpoint`	`-bp`
`--context`	`-c`
`--file`	`-f`
`--frame`	`-fm`
`--host`	`-n`
`--instance`	`-mt`
`--line`	`-l`
`--owner`	`-o`

Command	Abbreviation
`--port`	`-p`
`--raw`	`-R`
`--timeout`	`-to`
`--utf8`	`-u`
`--variable`	`-v`
`--verbose`	`-V`

Using the command-line debugger

Examining the stack with the command-line debugger

When the correlator stops at a breakpoint you can display the stack with the following command:

```
engine_debug stack
```

The results of this command show the number of the frame that contains each variable. In the following example, the frame number is the number before the right parenthesis:

```
0 )
        C:/dev/adbc/apama-test/system/correlator-debug/testcases/
           correctness/Corr_Debug_cor_002/Input/test.mon:35
        foo.baz.test.runtest[master(2)/foo.baz.test(3)]
1 )
        C:/dev/adbc/apama-test/system/correlator-debug/testcases/
           correctness/Corr_Debug_cor_002/Input/test.mon:19
        foo.baz.test.::listenerAction::[master(2)/foo.baz.test(3)]
```

You can use these frame numbers (frame IDs) as arguments to the `engine_debug inspect` command.

To see just the contents of the top frame, run this command:

```
engine_debug stack --frame 0
```

Using the command-line debugger

Displaying variables with the command-line debugger

To list all variables in the current stack frame, enter the following:

```
engine_debug inspect
```

To obtain the value for a variable in the current stack frame, enter the following:

```
engine_debug inspect -variable variableName
```

To obtain the value for a variable further down the stack, run the `stack` command to determine the frame number and then enter the following:

```
engine_debug inspect -variable variableName -frame frameid
```

Using the command-line debugger

Replaying an input log to diagnose problems

When you start the correlator, you can specify that you want it to copy all incoming messages to a special file, called an input log. An input log is useful if there is a problem with either the correlator process or an application running on the correlator. If there is a problem, you can reproduce correlator behavior by replaying the messages captured in the input log. Incoming messages include the following:

- Events
- EPL
- Java
- Correlator Deployment Packages (CDPs)
- Connection, deletion, and disconnection requests

If you are unable to diagnose the problem, you can provide the input log to Software AG Global Support. A support engineer can then feed your input log into a new correlator to try to diagnose the problem.

The information in the following topics describes how to generate and use an input log:

- "Creating an input log file" on page 161
- "Rotating an input log file" on page 161
- "Performance when generating an input log" on page 162
- "Reproducing correlator behavior from an input log" on page 162

See also: "Examples for specifying log filenames" on page 113.

Creating an input log file

To create an input log, specify the following option when you start a correlator:

```
--inputLog filename[${START_TIME}][${ROTATION_TIME}][${ID}].log
```

You specify the name of an input log file in the same way that you specify the name of a correlator status log file. See "Specifying log filenames" on page 111.

In addition, specify any other options that you would normally specify when you start the correlator.

Rotating an input log file

While the input log can get rather large, most file systems can handle large input logs with no special action on your part. However, you might encounter one of the following situations:

- You want to archive your input logs.

- Your operating system enforces a limit on file size.
- The input log has become too large.

In these situations, you can rotate the input log. Rotating the input log means that the correlator closes the current input log and starts sending messages to a new input log.

You should rotate the input log only when you have a specific need to do so. You do not want to have thousands of input logs in a directory since file systems do not handle this efficiently.

If you plan to rotate input logs, specify the `${ID}` tag when you specify the `--inputLog` option when you start the correlator. For examples, see "Examples for specifying log filenames" on page 113.

To rotate the input log, invoke the `engine_management` utility and specify the `-r rotateLogs` option. The name of the new input log is the same as the name of the closed input log except that the correlator increments the ID portion of the input log filename by `1`. See "Rotating all log files" on page 150.

Performance when generating an input log

When the file system that hosts the input log is fast, generating an input log should not have any noticeable effect on correlator performance. Consequently, the recommendation is to always run correlators that send information to input logs. Just make sure you have enough disk space for the input log. You need to monitor repeated use to determine how much space is required.

With the correlator generating an input log, you can implement your application so that it sends a minimum amount of information to the correlator status log. You do not need to log application information because you can always recover application information from the input log. Implementing an application that sends large amounts of application information to the correlator status log can negatively impact performance.

Reproducing correlator behavior from an input log

To use an input log to reproduce correlator behavior, you must do the following:

1. Run the `extract_replay_log` Python utility.
2. Run the `replay_execute` script that the `extract_replay_log` utility generates.

Invoking the extract script

The `extract_replay_log.py` script is in the `utilities` directory in your Apama installation directory. You must have at least Python 2.4 to run this utility. You can download Python from http://www.python.org. If you are using Linux, you probably already have Python installed.

The format for running the `extract_replay_log` utility is as follows:

```
extract_replay_log.py [options] inputLogFile
```

Replace *`inputLogFile`* with the path for the input log you want to extract. If you specify the first input log in a series, the subsequent input logs must be in the same directory as the first input log.

The options you can specify are as follows:

Option	Description

`-o=dir`
or
`--output=dir`
	Specifies the directory that you want to contain the output from the `extract_replay_log` utility. The default is the current directory.

`-l=lang`
or
`--lang=lang`
	Specifies the language of the script that the `extract_replay_log` utility generates. Replace *`lang`* with one of the following:
	• `shell` generates the `replay_execute.sh` UNIX shell script.
	• `batch` generates the `replay_execute.bat` Windows batch file. This is the default.

`-c` or `--correlator`	Specifies that the script that `extract_replay_log` generates should include the command line for starting a correlator. When you run the generated script, the correlator will be started with all of the command line options needed to replay the input log.
`-v` or `--verbose`	Indicates that you want verbose utility output.
`-h` or `--help`	Displays help for the utility.

The `extract_replay_log` utility generates the following:

- A script whose execution duplicates the correlator activity captured by the input log.
- Event files — each one is prefixed with `replay_`.
- EPL and possibly JAR and Correlator Deployment Package (CDP) files — each one is prefixed with `replay_`.

Invoking the replay script

Before you run the replay script, you can optionally edit the generated event files, EPL files, or JAR files to slightly modify the behavior you are about to replay. For example, you might add logging for debugging purposes. However, there are restrictions on what you can change:

- You cannot insert any of the following:
  - calls to `integer.getUnique()` or `rand()`
  - `send`, `emit`, `spawn...to`, `enqueue`, or `enqueue..to` statements
  - `context` constructors
- You cannot change the number of parseable events sent to the correlator. For example, you cannot attach a dashboard component to the input log because the dashboard components work by sending events to the correlator.
- You cannot change the number of event definitions and monitors injected.

Making any of these changes can potentially alter the behavior of later operations.

If you are using the MemoryStore and the correlator reads or writes to a store on disk then to accurately play back execution you must have a copy of that store as it was before the correlator modified it. Also, if you are using the MemoryStore from multiple contexts it is unlikely to replay correctly because the order of interaction with the MemoryStore is not in the input log.

After you have optionally edited the generated files, you are ready to invoke the `replay_execute` script. The `replay_execute` script tries to replay the contents of the input log into the correlator running on the default port.

While the correlator exactly reproduces the activity captured in the input log, it can execute the same activity faster during replay than when it was executed originally. This is because the correlator already has all the events it needs to process; it does not have to wait for any events. Replaying a log is typically significantly faster than original correlator activity. It is possible that you will find that the time it takes to replay a log is not much less than the time it took for the original activity. In this case, it is possible you were running too close to capacity during the original run. If that is the case, you risk not being able to keep up with the event flow during regular correlator execution. If you anticipate higher event flow then you should investigate optimizing your application or running it on a faster computer.

Event file format

You can use the `engine_send` tool to stream a sequence of events through the event correlator. The `engine_send` tool accepts input from one or more data files to support tests or simulations, or from `stdin` to allow dynamic generation of events. In the latter case, you can generate events from user input or by piping output from an event generation program to `engine_send`. In all cases, `engine_send` requires event data formatted as described in this section.

The `engine_receive` tool outputs events in this same file format. This means you can use events generated by the `engine_receive` tool as input to a second event correlator that is executing the `engine_send` tool.

Correlator Utilities Reference

Event representation

A single event is identified by the event type name and the values of all fields as defined by that type. Event type names must be 'fully-qualified' by prefixing the package name into which the corresponding event type was injected, unless the event was injected into the default package.

Each event is given on a separate line, separated by a new-line character. Only single-line comments are allowed. Start each comment line with / / or #. Any blank lines are ignored.

For example, following are three valid events:

```
// This is an event file that contains some sample events.
// Here are three stock price events:
StockPrice("XRX", 11.1)
StockPrice("IBM",130.6)
StockPrice("MSFT", 70.5)
```

For those events, the following event type definition must be injected into the default package:

```
event StockPrice {
   string stockSymbol;
```

```
   float stockValue;
}
```

If the above events were saved in an `.evt` file, `engine_send` would send each event in turn, as soon as the previous event finished transmission. This behavior can optionally be modified in several ways:

- Specifying that batches of events should be sent at specified time intervals.
- Specifying that all events on all queues should be processed before sending the next event.

Event file format

Event timing

In `.evt` files, it is possible to specify

- Time intervals for sending batches of events to the correlator.
- Waiting for all events on all queues at that point in time to be processed before sending the next event.

Adding BATCH tags to send events at intervals

You can specify time intervals for sending batches of events to the correlator. This is achieved by specifying the `BATCH` tag followed by a time offset in milliseconds. For example, the following specifies two batches of events to be sent 50 milliseconds apart.

```
BATCH 50
StockPrice("XRX", 11.1)
StockPrice("IBM", 130.6)
StockPrice("MSFT", 70.5)
BATCH 100
StockPrice("XRX", 11.0)
StockPrice("IBM", 130.8)
StockPrice("MSFT", 70.1)
```

The addition of a 'time' allows simulations of 'bursts' of events, or more random distributions of event traffic. Times are measured as an offset from when the current file was opened. If only one file of events is being read and transferred, then this would be the same as since the start of a run (that is, from the time that the `engine_send` tool starts processing the event data). If multiple files are being read in, the timing starts all over again upon the (re)opening of each file.

If the time given for a batch is less than the current time, or if no time is given following a `BATCH` tag (or if no `BATCH` tag is provided) then the events are sent as soon as they are read in, immediately following the preceding batch.

Using &FLUSHING mode for more predictable event processing order

Sending events in flushing mode can help provide a more predictable event processing order. However, flushing mode is slower than the default behavior.

By default, events are delivered in an optimal way, not waiting for previously sent events to be processed before the next event is delivered to contexts (or other consumers of channels). When flushing mode is enabled the behavior is as follows:

1. The correlator sends an event.

2. The correlator processes all events on all queues at that point in time, repeating this as many times as specified in the flushing specification.
3. The correlator sends the next event.

To enable flushing mode, insert the following line in a `.evt` file:

```
&FLUSHING(n)
```

Replace *n* with an integer that specifies how many times to flush queues in between each event. Set this to the maximum length of a chain of send-to operations between contexts that could occur in your application. If you specify a number that is bigger than required the correlator simply repeats the flush operation, which incurs a small overhead. To disable flushing mode, insert the following line in the `.evt` file:

```
&FLUSHING(0)
```

Enabling or disabling flushing mode affects only the events sent on that connection or from that event file.

When sending `&TIME` events in to a multi-context application, the time ticks are delivered directly to all contexts. This can be different than the way in which events in the `.evt` file are sent in to the correlator and then sent between contexts in an application. This difference can result in processing events at an incorrect simulated time. In these cases, sending `&FLUSHING(1)`, for example, before sending time ticks and events can result in more predictable and reliable behavior.

Event file format

Event types

The following example illustrates how each type is specified in an event representation. Given the event type definitions:

```
event Nested {
  integer i;
}
event EveryType {
  boolean b;
  integer i;
  float f;
  string s;
  location l;
  sequence<integer> si;
  dictionary<integer, string> dis;
  Nested n;
}
```

the following is a valid event representation for an `EveryType` event:

```
EveryType (
  true,               # boolean is true/false (lower-case)
  -10,                # positive or negative integer
  1.73,               # float
  "foo",              # strings are (double) quoted
  (1.0,1.0,5.0,5.0),  # locations are 4-tuples of float values
  [1,2,3],            # sequences are enclosed in brackets []
  {1:"a",2:"b"},      # dictionaries are enclosed in braces {}
  Nested(1)           # nested events include event type name
)
```

Note that this example is split over several lines for clarity; in practice this definition would all be written on the same line.

Types can of course be nested to create more complex structures. For example, the following is a valid event field definition:

```
sequence<dictionary<integer, Nested> >
```

and the following is a valid representation of a value for this field:

```
[{1:Nested(1)}, {2:Nested(2)}, {3:Nested(3)}]
```

Event file format

Event association with a channel

The `engine_send` utility can send an event file that associates channels with events. Likewise, the `engine_receive` utility can output an event file that includes the channel on which an event was received. The event format is the same for both utilities:

```
"channel_name",event_type_name(field_value1[, field_valuen]...)
```

For example, suppose you want to send `Tick` events, which contain a `string` followed by an `integer`, to the `PreProcessing` channel. The contents of the `.evt` file would look like this:

```
"PreProcessing",Tick("SOW", 35)
"PreProcessing",Tick("IBM", 135)
```

A channel name is optional. In a file being sent with the `engine_send` utility, you can mix event representations that specify channels with event representations that do not specify channels. Events for which a channel is specified go to only those contexts subscribed to that channel.

The default behavior is that events are sent on the default channel (the empty string) when a channel is not explicitly specified. Events sent on the default channel go to all public contexts. All running Apama queries receive events sent on the default channel. To change the default behavior for events sent by the `engine_send` utility, you can specify `engine_send -c channel`. If a channel is not explicitly specified for an event then it is sent to the channel identified with the `-c` option . See "Sending events to correlators" on page 126.

Event file format

Using the data player command-line interface

Apama Studio's Data Player lets you play back previously saved event data as you develop your application. During playback, you can analyze the behavior of your application. Or, if you modify the saved event data, you can analyze how your application performs with the altered data. Apama Studio plays back event data that has been stored in standard data formats.

When you are ready to test your application the command-line interface to the Data Player lets you write scripts and unit tests to exercise the API layers. Or, if you just want to play back events to the correlator, using the command-line interface might be easier than using the Data Player GUI in Apama Studio.

To use the command-line interface to the Data Player, you must have already used the GUI interface in Apama Studio. That is, you must have already defined queries and query configurations in

Apama Studio. When you use the command-line interface, you specify query names and query configurations that you created in Apama Studio.

The Data Player relies on Apama Database Connector (ADBC) adapters that are specific to standard ODBC and JDBC database formats as well as the comma-delimited Apama Sim format. Apama release 4.1 and earlier captured streaming data to files in the Sim format. These adapters run in the Apama Integration Application Framework (IAF), which connects the data sources to the correlator. The information here assumes that you are already familiar with the information in "Using the Data Player" in *Using the Apama Studio Development Environment*.

Synopsis

To use the Data Player from the command line, enter `adbc_management.bat` (Windows) or `adbc_management` (UNIX) from the `bin` directory in your `APAMA_HOME` directory:

To obtain the following usage message, run the command with the `help` option:

```
Usage:
adbc_management --query <queryName> --configFile <file> [ options ]
Where options include:
-h | --help                This message
-n | --hostname <host>     Connect to correlator on <host>.
                           Default to localhost.
-p | --port <port>         Connect to correlator on <port>.
                           Default to 15903.
--query <queryName>        Run the query <queryName> specified in
                           query configuration file.
--configFile <file>        Query configuration file to use.
--username <user>          Optional username for database connection.
--password <password>      Optional password for database connection.
--returnType <returnType>  Optional returnType of playback events.
                           Default is native.
--backTest <true|false>    Optional switch to generate time event from
                           data.
                           Use false if correlator is not running
                           -Xclock option. Default is true.
--speed <playBackSpeed>    Optional speed for playing back query.
                            <= 0.0, as fast as possible
                            > 0.0 for some multiple of playback speed.
                            (Ignored if --backTest false is used.)
```

Options

Option	Description
`-h`	Display usage information
`-n`	Name of the host on which the event correlator is running (default is `localhost`). Non-ASCII characters are not allowed in host names.
`-p`	Port on which the event correlator is listening (default is `15903`).
--query *queryName*	Run the specified query, which is defined in the query configuration file that you identify with the `--configFile` option. This is a query you created in Apama Studio in the Data Player Editor. You did this when you clicked on the + button on the action bar. You specified a query name and that is the name you need to specify here

Option	Description
--configFile *file*	Use this query configuration file. Specify the query configuration file associated with your project. In Apama Studio, the query configuration file is always called `dataplayer_queries.xml` (in the project`/config` directory).
--username *user*	The user name to use for the database connection. Optional.
--password *password*	The password to use for the database connection. Optional.
--returnType *returnType*	The type of the playback events returned. The default is `Native`. The only other choice is `Wrapped`. A return type of `Native` means that each matching event is sent as-is to the correlator. When you specify `Wrapped`, each matching event is inside a container event. The name of the container event is `Wrapped` followed by the name of the event in the container, for example, `HistoricalTick`. Event wrapping allows events to be sent to the correlator without triggering application listeners. A separate, user-defined monitor can listen for wrapped events, modify the contained event, and reroute it such that application listeners can match on it.
--backTest <true|false>	This option is equivalent to Apama Studio's Data Player option to "Generate time event from data". When the correlator is running with the `-Xclock` option, time in the correlator is controlled by `&TIME()` events. This is how the Data Player controls the playback speed. If the correlator is not running with the `-Xclock` option, the correlator keeps its own time. The default is true, which means that the correlator is running with the -Xclock option. Set this option to false when the correlator is not running with the `-Xclock` option.
--speed *playBackSpeed*	Specifies the speed for playing back the query. Optional. A float value less than or equal to `0.0` means that you want the correlator to play it back as fast as possible. A float value greater than 0.0 indicates a multiple for the playback speed. To play at normal speed, specify `1.0`.For half normal speed, specify `0.5`.For twice normal speed, specify `2.0`.For 100 times normal speed, specify `100.00`.

Correlator Utilities Reference

Using the Apama component extended configuration file

The Apama component extended configuration file is an optional file that you can use to do the following:

- Bind Apama server components to a particular set of addresses.

- Specify that Apama client connections must be from a particular IP address or range of IP addresses.
- Set environment variables for Apama components.
- Set log files and log levels for EPL root, packages, monitors, or events.

You can specify the optional extended configuration file when you start the correlator. If you do, the settings in the extended configuration file apply to the following Apama components:

- Correlator
- IAF
- Sentinel Agent
- `engine_receive` correlator utility

In an extended configuration file, if a line includes a special character that you want to be treated as a literal, you must escape it by inserting a backslash just before it. A character that follows two consecutive backslashes (\ \) is treated as literal. Single quotes inside double quotes are treated literally. Double quotes inside single quotes are treated literally. See the example in "Sample extended configuration file" on page 173.

Correlator Utilities Reference

Binding server components to particular addresses

To bind Apama server components to a particular address or set of addresses, specify a `BindAddress` line for each address. Specify this in the `[Server]` section of the extended configuration file. For example:

```
[Server]
BindAddress=127.0.0.1:15903
BindAddress=10.0.0.1
```

You can specify as many `BindAddress` lines as you want. Clients can connect to any of the listed addresses.

An IP address is required. If you do not specify a port, the Apama server components use the port that is specified when the correlator is started. This makes it possible to share extended configuration files if you want to restrict connections according to only addresses.

If you do not specify an extended configuration file when you start the correlator, or there are no `BindAddress` entries in the extended configuration file, the Apama components bind to the wildcard address (`0.0.0.0`).

Using the Apama component extended configuration file

Ensuring that client connections are from particular addresses

To ensure that client connections are from particular addresses, add one or more `AllowClient` entries to the extended configuration file in the `[Server]` section. For example:

```
[Server]
```

```
AllowClient=127.0.0.1
AllowClient=192.168.128.0/17
```

An `AllowClient` entry takes an IP address, as in the first example above, or a CIDR (Classless Inter-Domain Routing) address range, as in the second example above. With these example entries in the extended configuration file, the Apama components allow connections from either the localhost (`127.0.0.1`) or IP addresses where the first 17 bits match the first 17 bits of `192.168.128.0`. The Apama components to not accept connections from any other IP addresses. This creates a "whitelist" of allowable IP addresses.

If you specify an extended configuration file when you start the correlator, and if there are any `AllowClient` entries in the extended configuration file then the Apama components do not allow connections from any IP address that does not fall within one of the `AllowClient` ranges specified. If you do not specify an extended configuration file when you start the correlator, or there are no `AllowClient` entries in an extended configuration file that you do specify, the Apama components accept connections from any client.

This feature is intended to prevent mistakenly connecting to the wrong server. It is not intended to prevent malicious intruders since it provides no protection against address spoofing.

Using the Apama component extended configuration file

Setting environment variables for Apama components

You can use the extended configuration file to set environment variables. Put environment variable declarations in the `[Environment]` section. For example:

```
[Environment]
LD_LIBRARY_PATH=/usr/local/mydir
```

If you specify an extended configuration file when you start the correlator, and if there are any environment variable entries in the extended configuration file then the Apama components use those environment variable settings. If you do not specify an extended configuration file when you start the correlator, or there are no environment variable entries in an extended configuration file that you do specify, the Apama components use the environment variable settings specified elsewhere.

Note: You must set variables such as `LD_PRELOAD` and `LD_LIBRARY_PATH` before the affected component starts execution. These environment variables are read-only. If you change them after component execution starts, the component does not recognize the change.

Using the Apama component extended configuration file

Setting log files and log levels in an extended configuration file

You can configure per-package logging in two ways:

- Statically, in the extended configuration file when starting the correlator, as described here.
- Dynamically, using the `engine_management setApplicationLogFile/Level` request. See "Setting logging attributes for packages, monitors and events" on page 147.

To set log files and/or log levels for EPL root, packages, monitors, or events, specify entries in the `[ApplicationLogging]` section of the extended configuration file.

To set the default log file and level for the EPL root package, specify two lines in the following format:

```
RootFile=rootLogFilename
RootLevel=ROOTLOGLEVEL
```

Replace *rootLogFilename* with the name of the log file for the EPL root package. No spaces are allowed in the log file name. Replace *ROOTLOGLEVEL* with TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT or OFF. For example:

```
RootFile=apama\\root.log
RootLevel=ERROR
```

Note that you must insert a backslash to escape special characters. In the previous example, there is a filepath that contains a backslash that must be escaped. Hence, the double backslash. You do not need to specify both a log file and a log level; you can specify one or the other. If you do not specify a log file or log level for the root package, it defaults to the correlator's log file/log level.

To set the default log file and level for an EPL package, monitor, or event, specify two lines in the following format:

```
PackageFile.node=nodeLogFilename
PackageLevel.node=NODELOGLEVEL
```

Replace *node* with the name of the EPL package, monitor, or event you are setting the logging attribute for. If a monitor or event is in a named package and not the root package, be sure to specify the fully qualified name.

Replace *nodeLogFilename* with the name of the default log file for the specified EPL package, monitor or event. No spaces are allowed in the log file name. Replace *NODELOGLEVEL* with TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT or OFF. This is the default log level for the specified node.

For example:

```
PackageFile.com.myCompany.Client=apama\\Client.log
PackageLevel.com.myCompany.Client=DEBUG
```

For a particular node, you do not need to specify both a log file and a log level; you can specify one or the other. If you do not specify a log file or log level for a particular node, it defaults to the settings for a parent node. See "Tree structure of packages, monitors, and events" on page 148.

There is no limit to the number of log settings that can appear in the configuration file. The last line that refers to a given node takes priority over earlier lines. When you set log attributes in the extended configuration file the rules for hierarchical logging apply. See "Setting logging attributes for packages, monitors and events" on page 147.

If you pass an extended configuration file to a correlator when you start that correlator and the configuration file contains an `[ApplicationLogging]` section, the correlator uses the logging settings in that section. If you do not pass a configuration file when you start a correlator, or there are no settings in the `[ApplicationLogging]` section, then correlator initialization does not include any log settings except for the default correlator log.

Whether or not you specify an extended configuration file when you start a correlator, any log settings you specify can be overwritten after initialization by invoking the `engine_management` utility and specifying the `setApplicationLogFile` and/or `setApplicationLogLevel` keywords. See "Managing application log levels" on page 148 and "Managing application log files" on page 149.

Using the Apama component extended configuration file

Sample extended configuration file

Save the extended configuration file as a text file. Blank lines are ignored. For example, the contents of `ApamaExtendedConfig.txt` might be as follows:

```
[Server]
BindAddress=127.0.0.1:15903
BindAddress=10.0.0.1
AllowClient=127.0.0.1
AllowClient=10.0.0.0/24
[Environment]
LD_LIBRARY_PATH=/usr/local/mydir
[ApplicationLogging]
PackageFile.com.myCompany=apama\\apama.log
PackageLevel.com.myCompany=WARN
PackageLevel.com.myCompany.Client=DEBUG
```

Using the Apama component extended configuration file

Starting a correlator with an extended configuration file

To use an Apama component extended configuration file, specify the `-Xconfig` option with the extended configuration file path when you start the correlator. For example, if both the license file and the extended configuration file are in the same directory as the correlator executable, and the name of the extended configuration file is `ApamaExtendedConfig.txt`:

```
run correlator -l license.txt -Xconfig ApamaExtendedConfig.txt
```

Using the Apama component extended configuration file