Developing Adapters

5.2.0

August 2014

This document applies to Apama 5.2.0 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.

Copyright © 2013-2014 Software AG, Darmstadt, Germany and/or Software AG USA Inc., Reston, VA, USA, and/or its Subsidiaries and or/its Affiliates and/or their licensors.

Use, reproduction, transfer, publication or disclosure is prohibited except as specifically provided for in your License Agreement with Software AG.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its Subsidiaries and/or its Affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at http://documentation.softwareag.com/legal/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products." This document is located at http://documentation.softwareag.com/legal/ and/or in the root installation directory of the licensed product(s).

Document ID: PAM-Developing_Adapters-5.2.0-20140808@233876

Table of Contents

Preface

About this documentation

This documentation describes how to use the Apama Integration Adapter Framework (IAF). The IAF is a middleware-independent and protocol-neutral adapter tailoring framework designed to provide for the easy and straightforward creation of software adapters to interface Apama with middleware buses and other message sources. It provides facilities to generate adapters that can communicate with third-party messaging systems, extract and decode self-describing or schema-formatted messages, and flexibly transform them into Apama events. Vice-versa, Apama events can be transformed into the proprietary representations required by third-party messaging systems. It provides highly configurable and maintainable interfaces and semantic data transformations. An adapter generated with the IAF can be re-configured and upgraded at will, and in many cases, without having to restart it. Its dynamic plug-in loading mechanism allows a user to customize it to communicate with proprietary middleware buses and decode message formats.

Preface

How this book is organized

The information in this book is organized as follows:

- "The Integration Adapter Framework" on page 11 describes the architecture of the IAF.
- "Using the IAF" on page 17 describes how to start, stop, and manage the IAF runtime. It also describes the IAF sample applications.
- "C/C++ Transport Plug-in Development" on page 48 presents the C/C++ Transport Plug-in Development Specification and describes how to implement transport layer plug-ins in C and C++.
- "C/C++ Codec Plug-in Development" on page 62 presents the C/C++ Codec Plug-in Development Specification and describes how to implement codec layer plug-ins in C and C++.
- "C/C++ Plug-in Support APIs" on page 83 describes programming interfaces provided with the Apama software that may be useful in implementing transport layer and codec plug-ins for the IAF.
- "Transport Plug-in Development in Java" on page 89 presents the Transport Plug-in Development Specification for Java and describes how to implement transport layer plug-ins in Java.

- "Java Codec Plug-in Development" on page 98 presents the Codec Plug-in Development Specification for Java and describes how to implement codec layer plug-ins in Java.
- "Plug-in Support APIs for Java" on page 111 describes the Java interface for recording status and error log messages from the IAF runtime and any plug-ins loaded within it.
- "Monitoring Adapter Status" on page 116 describes how to provide status information about IAF adapters.
- "Out of and Connection Notifications" on page 130 describes the set of out of band events that can be used to notify adapters when sender and receiver components connect to or disconnect from the IAF.
- "The Event Payload" on page 135 describes how to add a payload field to an Apama event type. A payload field is used to accommodate data that does not comply with the rigid structure of an Apama event.
- "Standard Plug-ins" on page 137 describes the standard Apama plug-ins. These are the File Transport, String Codec and Null Codec plug-ins and are available in both C and Java.
- "Apama File Adapter" on page 164 describes the Apama File adapter and the Apama Database Connector (ADBC) adapter.

Preface

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 5.2 > 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	Contents
What's New in Apama	Describes new features and changes since the previous release.
Installing Apama	Instructions for installing the Developer, Server, or User Apama installation options.

Title	Contents
Introduction to Apama	Introduction to developing Apama applications, discussions of Apama architecture and concepts, and pointers to sources of information outside the documentation set.
Using Apama Studio	Instructions for using Apama Studio to create and test Apama projects; write, profile, and debug EPL programs; write JMon programs; develop custom blocks; and store, retrieve and playback data.
Developing Apama Applications in Event Modeler	Instructions for using Apama Studio's Event Modeler editor to develop scenarios. Includes information about using standard functions, standard blocks, and blocks generated from scenarios.
Developing Apama Applications in EPL	Introduces Apama's Event Processing Language (EPL) and provides user guide type information for how to write EPL programs. EPL is the native interface to the correlator. This document also provides information for using the standard correlator plug-ins.
Apama EPL Reference	Reference information for EPL: lexical elements, syntax, types, variables, event definitions, expressions, statements.
Developing Apama Applications in Java	Introduces the Apama in-process API for Java, referred to as JMon, and provides user guide type information for how to write Java programs that run on the correlator. Reference information in Javadoc format is also available.
Building Dashboards	Describes how to create dashboards, which are the end-user interfaces to running scenario instances and data view items.
Dashboard Property Reference	Reference information on the properties of the visualization objects that you can include in your dashboards.
Dashboard Function Reference	Reference information on dashboard functions, which allow you to operate on correlator data before you attach it to visualization objects.
Developing Adapters	Describes how to create adapters, which are components that translate events from non-Apama format to Apama format.
Developing Clients	Describes how to develop C, C++, Java, or .NET clients that can communicate with and interact with the correlator.
Writing Correlator Plug-ins	Describes how to develop formatted libraries of C, C++ or Java functions that can be called from EPL.
Deploying and Managing Apama Applications	Describes how to:

Title	Contents
	• Use the Management & Monitoring console to configure, start, stop, and monitor the correlator and adapters across multiple hosts. • Deploy dashboards over wide area networks, including the internet, and provide dashboards with effective authorization and authentication. • Improve Apama application performance by using multiple correlators, and saving and reusing a snapshot of a correlator's state. • Use the Apama ADBC adapter to store and retrieve data in JDBC, ODBC, and Apama Sim databases. • Use the Apama Web Services Client adapter to invoke Web Services. • Use correlator-integrated messaging for JMS to reliably send and receive JMS messages in Apama applications. • Use Universal Messaging to connect correlators.
Using the Dashboard Viewer	Describes how to view and interact with dashboards that are receiving run-time data from the correlator.

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: The Integration Adapter Framework

There are two ways of integrating with Apama through software. The first is to use the low-level Client Software Development Kits (SDKs) for C, C++ or Java to write your own custom software interface. The second is to instantiate an adapter with the higher-level Integration Adapter Framework (IAF). This document *Developing Adapters* describes how to use the IAF.

For information on using the Client Software Development Kits see "Working with projects" in *Developing Clients*.

This section contains the following topics:

Overview

The IAF is a middleware-independent and protocol-neutral adapter tailoring framework. It is designed to allow easy and straightforward creation of software adapters to interface Apama with middleware buses and other message sources. It provides facilities to generate adapters that can communicate with third-party messaging systems, extract and decode self-describing or schema-formatted messages, and flexibly transform them into Apama events. In the opposite direction, Apama events can be transformed into the proprietary representations required by third-party messaging systems. It provides highly configurable and maintainable interfaces and semantic data transformations. An adapter generated with the IAF can be re-configured and upgraded at will, and in many cases, without having to restart it. Its dynamic plug-in loading mechanism allows a user to customize it to communicate with proprietary middleware buses and decode message formats.

On the other hand, the SDKs provide lower-level client application programming interfaces that allow one to directly connect to Apama and transfer Apama Event Processing Language (EPL) code and events in and out. (The Apama Event Processing Language is the new name for MonitorScript.) The SDKs provide none of the abstractions and functionality of the IAF, and hence their use is only

recommended when a developer needs to write a highly customized and very high performance software client, or wishes to integrate existing client code with Apama in process.

The IAF is available on all platforms supported by Apama, although not all adapters will work on all platforms. For the most up-to-date information about supported platforms and compilers, see Software AG's Knowledge Center in Empower at https://empower.softwareag.com.

Architecture

The first step in integrating Apama within a user environment is to connect the correlator to one or more message/event sources and/or sinks. In the majority of cases the source or provider of messages will be some form of middleware message bus although it could also be a database or other storage based message source, as well as an alternative network-based communication mechanism, like a trading system. The same applies for the sink or consumer of messages.

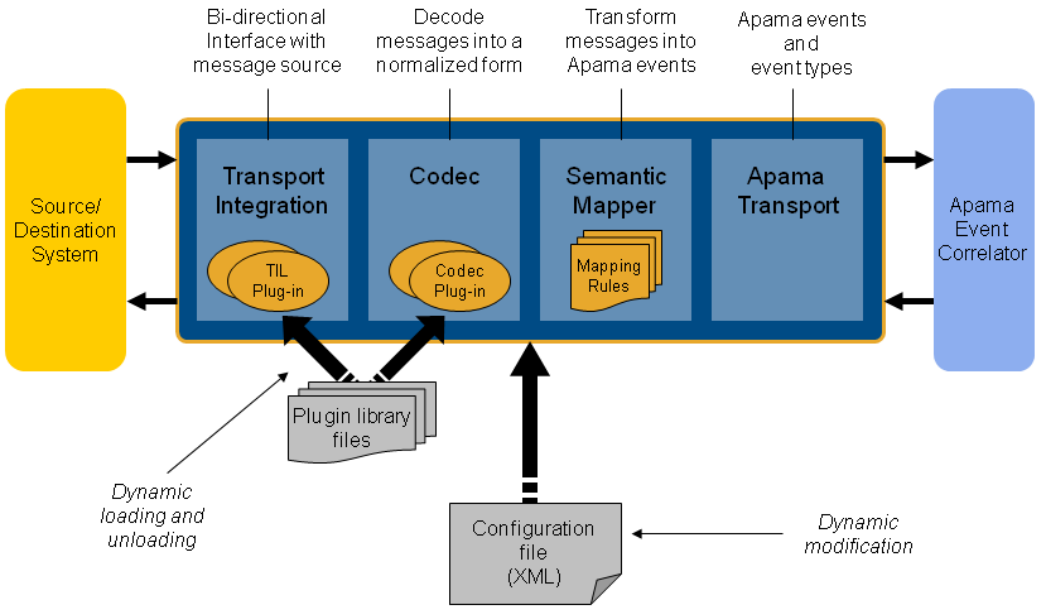
These message sources and/or sinks vary extensively. Typically each comes with its own proprietary communications paradigm, communications protocol, message representation and programming interfaces. Interfacing any software like Apama with a source/sink of messages like a message bus therefore requires the writing of a specialized software adapter or connector, which needs to be maintained should the messaging environment or the message representation change. The adapter needs to interface with the messaging middleware or message source, receive messages from it and decode them, and then transform them into events that Apama can understand and process. The latter transformation is not always straightforward, as the message representation might vary from message to message and require semantic understanding.

Conversely, the events generated by Apama need to be processed in the inverse direction and eventually end up back on the message bus.

Note: In the Apama documentation, a message traveling from a message source through the IAF and into Apama is described as traveling *downstream*, whereas a message output as an alert from Apama and progressing back out through the IAF towards a message sink is described as traveling *upstream*.

In order to facilitate development of software adapters, Apama provides the IAF. In contrast to the SDKs, the IAF is not a programming library. The IAF is effectively a customizable, middleware independent, generic adapter that can be adapted by a user to communicate with their middleware and apply their specific semantic transformations to their messages.

Figure 1. The architecture of an IAF adapter



As illustrated above, the IAF acts as the interface between the messaging middleware and the event correlator. There are four primary components to the IAF:

- *The Transport Layer*. This is the layer that communicates with the user's message source/sink. Its functionality is defined through one or more Apama or user-provided message source/sink specific transport plug-ins, written in C, C++ or Java.
- *The Codec Layer*. The codec layer translates messages from any custom representation into a normalized form and vice-versa. The transformation is carried out by one of its codec plug-ins. These can be provided by Apama or by the user, and may be written in C, C++ or Java. Note that Java codec plug-ins may only communicate with Java transport plug-ins, and C/C++ codec plug-ins with C/C++ transport plug-ins.
- *The Semantic Mapper*. This layer provides functionality to transform the messages received from the message source into Apama events. The Semantic Mapper is a standard component that is configured for use with a particular adapter by means of a set of semantically rich translation mapping rules. These rules define both how to generate Apama events from externally generated messages and how user-custom messages for an external destination may be generated from Apama events.

  An adapter can be configured to bypass this kind of mapping in the Semantic Mapper. Used this way, the Semantic Mapper converts the string form of an Apama event directly to a normalized form and vice-versa.
- *The Apama Interface layer*. This layer abstracts away communication with Apama's event correlator. It injects EPL definitions and event instances into the event correlator and asynchronously receives back events from it.

Additionally, the `engine_send` and `engine_receive` tools can be run against the IAF simply by supplying the port on which the IAF is running. For example, running

```
engine_receive -p 16903
```

connects to the IAF running on the default port and receives all event emitted by it.

The next sections explore the transport, codec and Semantic Mapper layers in more detail.

The transport layer

The *transport layer* is the front-end of the IAF. The transport layer's purpose is to communicate with an external message source and/or sink, extracting downstream messages from the message source ready for delivery to the codec layer, and sending Apama events already encoded by the codec layer on to the message sink.

This layer interfaces with the middleware message bus or message source/sink through the latter's custom programming interface. It receives and dispatches messages from and to it as well as carrying out other proprietary operations. Depending on the nature of the message bus or message source in use, these operations could include opening a database file and running SQL queries on it, registering interest in specific message topics with a message bus, or providing security credentials for authentication. Note that if the IAF is also being used to output messages back to a message sink, then it must also carry out the operations required to enable this; for example, opening and writing to a database file, or dispatching messages onto a message bus.

As this functionality depends entirely on the message source and/or sink the IAF needs to interface with, the transport layer's functionality is loaded dynamically through a custom transport plug-in.

Although Apama provides a set of standard transport plug-ins for popular messaging systems, the user may develop new transport plug-ins. See "C/C++ Transport Plug-in Development" on page 48 and "Transport Plug-in Development in Java" on page 89 for the Transport plug-in Development Specifications for C/C++ and Java, which describe how custom transport plug-ins may be developed in the C, C++ and Java programming languages.

The transport layer can contain one or more transport layer plug-ins. These are loaded when the adapter starts, and the set of loaded plug-ins can be changed while the adapter is running. In addition, a loaded plug-in may be re-configured at any time using the IAF Client tool. If a transport plug-in requires startup or re-configuration parameters, these need to be supplied in the IAF configuration file as documented in "The IAF configuration file" on page 26.

Because a transport layer plug-in effectively implements a custom message transport, this manual uses the terms *transport layer plug-in* and *event transport* interchangeably.

The codec layer

While the transport layer communicates with the custom message sources and sinks and extracts messages, such as stock trade data, from them, the responsibility of the codec layer is to correctly interpret and decode each message into a 'normalized' format on which the semantic mapping rules can be run; similarly in the upstream direction a codec may be responsible for encoding a normalized message in an appropriate format for transmission by particular transport(s).

Message sources like middleware message buses typically use proprietary representation of messages. Messages might appear as strings (possibly human readable or otherwise encoded) or sequences of binary characters. Messages might also be self-describing (possibly in XML or through some other proprietary descriptive format) or else be structured according to a schema available elsewhere.

Producing a universal generic normalized format from these messages requires the codec layer to understand the particular format of the messages. In the upstream direction the codec layer needs to encode the messages correctly according to the destination message sink.

As with the transport layer, in order to enable this custom functionality, the IAF is designed to dynamically load codec plug-ins that are capable of decoding and encoding the messages being received, when supplied with any required configuration properties. Apama provides some generic codec plug-ins, such as the `StringCodec` codec. This can decode most string based name-value representations of messages once it is configured with the syntactic elements used to delimit the elements in a message. In addition the user may develop proprietary codec plug-ins. See "C/C++ Codec Plug-in Development" on page 62 and "Java Codec Plug-in Development" on page 98 for the Codec plug-in Development Specifications for C/C++ and for Java, which describe how custom codec plug-ins may be developed using the C, C++ and Java programming languages.

An adapter can load multiple codec plug-ins (to deal with different message types). These are loaded at startup and the set of loaded codecs can be changed while the adapter is running. Individual codec plug-ins may also be re-configured at any time. If a codec plug-in requires startup or re-configuration parameters, these need to be supplied in the IAF configuration file as documented in "The IAF configuration file" on page 26.

This manual uses the terms *codec layer plug-in* and *event codec* interchangeably.

The Semantic Mapper layer

The Semantic Mapper maps and transforms incoming messages into Apama events that can be passed into the event correlator. Conversely, it can accept incoming Apama events and map them into messages that can be sent upstream on the user's message sink.

Apama events are rigidly defined. Every event must be structured according to a well-defined type definition. Therefore all events are of a specific named type, where this defines the number of fields (or parameters) in the event, their order, the name of each field, and its type. Furthermore it is possible to define which fields are relevant for querying in EPL event expressions, and which are not. See *Developing Apama Applications in EPL* for further information on event type definitions and EPL event expressions. This rigorous format permits the event correlator to be highly optimized and contributes towards Apama's scalable performance.

The source messages that are to be passed into Apama as events (or the sink messages that Apama needs to generate) might match this specification, in which case the mapping will be straightforward. However, they might also differ in several ways, some of which are listed here:

- The messages might be self-describing, and need to be parsed in order to deduce what fields they contain.
- The fields contained in every message might appear in varying order.
- Some messages of different types and with differing sets of fields might reflect the same information but in a different format (e.g. trade events from different markets or news headlines from different sources).
- The set of fields contained in messages might differ even if the messages are all of the same type.
- The messages might not be of an obvious type, and their nature (e.g. a trade event or a news headline) might need to be deduced from their contents.
- The set of fields might be enhanced over time to capture additional information.

- Some messages might have fields that are completely irrelevant.
- Some messages might have fields that are irrelevant for matching on but might be useful otherwise.

In order to address these conditions and allow meaningful Apama events to be created from external messages, the Semantic Mapper supports a semantically rich set of translation and transformation rules. These need to be expressed in the IAF configuration file.

The rules available are described in "Event mappings configuration" on page 29.

You can configure an adapter so that some events bypass this kind of mapping in the Semantic Mapper. Instead of mapping each field in an incoming event to a field in an Apama event or the converse, the entire event is treated as a string in a single field.

Contents of the IAF

The Integration Adapter Framework is included when you select Developer or Server during the Apama installation.

The Integration Adapter Framework contains the following components:

- Core files – these include the IAF Runtime, the management tools and the libraries they require.
- Example adapters written in C and Java – this includes the complete sources of the `FileTransport`/`JFileTransport` transport layer plug-ins and the `StringCodec`/ `JStringCodec` plug-ins, sample configuration files, a file with a set of input messages, an EPL file with a sample application, and a set of reference result messages.

  There is also a set of Market examples written in C and Java – these provide access to streaming prices for `Depth` and `Tick` and a facility to place orders, on which executions are reported. The adapter also reports its status (whether it is connected or not, or if the IAF process has been stopped). This is used by the subscription and order management services. A sample server implemented in Python is included. It requires Python 2.4 or later. Refer to the README file in the `samples\iaf_plugin\market` directory for more information and instructions on how to build and run the Market examples.
- A suite of development materials – these include the C/C++ header files and Java API sources required to develop transport and codec layer plug-ins for both languages. Also included is a skeleton transport and codec plug-in in C, the IAF configuration file XML Document Type Definition (DTD), a makefile for use with GNU Make on UNIX, and a 'workspace' file for use with Microsoft's Visual Studio.NET on Microsoft Windows.

Chapter 2: Using the IAF

This section describes how to start and manage the Integration Adapter Framework and how to specify an adapter's configuration file. It contains the following topics:

The IAF runtime

Once installed, running the IAF is straightforward. As already stated, the IAF is not a development library but a generic adapter framework whose functionality can be tailored according to a user's requirements through loading of the appropriate plug-ins.

In order to create an adapter with the IAF, one must supply a configuration file. This file – described in "The IAF configuration file" on page 26 – specifies which plug-ins to load and what parameters to configure them with, defines the translation and transformation rules of the Semantic Mapper, and configures communication with Apama.

The adapter can then be started as follows:

```
> iaf configuration.xml
```

IAF library paths

In order for the IAF to successfully locate and load C/C++ transport layer and codec plug-ins, the location(s) of these must be added to the environment variable `LD_LIBRARY_PATH` on UNIX, or `PATH` on Windows.

A transport or codec plug-in library may depend on other dynamic libraries, whose locations should also be added to the `LD_LIBRARY_PATH` or `PATH` environment variable as appropriate for the platform.

The documentation for a packaged adapter will state which paths should be used for the adapter's plug-ins. Note that on the Windows platform, the IAF may generate an error message indicating that it was unable to load a transport or codec plug-in library, when in fact it was a dependent library of the plug-in that failed to load. On UNIX platforms the IAF will correctly report exactly which library could not be loaded.

When using the IAF with a Java adapter the location of the Java Virtual Machine (JVM) library is determined in the same way. On UNIX systems the `LD_LIBRARY_PATH` environment variable will be searched for a library called `libjvm.so`, and on Windows the IAF will search for `jvm.dll`, first in the `third_party\jre\bin\server` and `third_party\jre\bin` directories of the Apama installation referenced by the `APAMA_HOME` environment variable, then in any other directories on the `PATH` environment variable. Using a JVM other than the one shipped with Apama is not supported and Technical Support will generally request that any Java-related problems with the IAF are reproduced with the supported JVM. Apama requires JDK 6.0 and ships with Oracle JRE 7.0. See "Java configuration (optional)" on page 44 for information about how the location of Java plug-in classes are determined.

IAF command line options

The complete usage information for the executable `iaf` (on UNIX) or `iaf.exe` (on Windows) is as follows. This can be displayed at any time by launching the IAF with the `--help` option.

```
Usage: iaf [ options ] [ config.xml ]
Where config.xml is the name of a configuration file using the format
described in the Integration Adapter Framework documentation.
Options include:
  -V | --version           Print program version info
  -h | --help              This message
  -p | --port <port>       Port to listen for commands on (default is 16903)
  -f | --logfile <file>    Log to named file (default is stderr)
  -l | --loglevel <level>  Set logging verbosity.  Available levels
                           are TRACE, DEBUG, INFO, WARN, ERROR, FATAL, CRIT and OFF.
                           The default logging level is INFO.
  -t | --truncat           Truncate the log file
  -N | --name <name>       Set the component name
  -e | --events            Dump event definitions to stdout then exit
  --logQueueSizePeriod <p> Send info to log every <p> seconds
  -Xconfig | --configFile <file> Use service configuration file <file>

Note that a configuration file must be provided unless the -h or -V
options are used.
```

Unless `-e` or `--events` are used, the above will generate and start a custom adapter, load and initialize the plug-ins defined in the configuration file, connect to Apama, and start processing incoming messages.

When the `-e` or `--events` command line switches are used, `iaf` generates event definitions that can be saved to a file and injected during your application's startup sequences as specified by Apama Studio, the Enterprise Management and Monitoring console (EMM), or Apama command line tools. If either of these switches is used, the IAF will load the IAF configuration file, process it, generate the event definitions and print them out onto `stdout` (standard output) and promptly exit. A valid configuration file must be supplied with either of these switches. The output definitions are grouped by package, with interleaved comments between each set. If all the event types in the configuration are in the same package, the output will be valid EPL code that can be injected directly into the correlator. Otherwise, it will have to be split into separate files for each package. The IAF can be configured to automatically inject event definitions into a connected correlator, but this is not the

default behavior. The event definitions generated by the `-e` or `--events` options are exactly what the IAF would inject into the correlator, if configured to do so.

The `-Xconfig` and `--configFile` are reserved for usage under guidance by Apama support. For more information about the service configuration file, see .

If the `--logfile` and `--loglevel` command line switches are provided, any logging settings set in the IAF configuration file (`<logging>` and `<plugin-logging>`) will be ignored.

If the IAF cannot write to the log file specified either with the `--logfile` option or in the adapter's configuration file, the IAF will fail to start.

IAF log file status messages

The IAF sends status information to its log file every 5 seconds (the default behavior) or at time intervals you specify with the `--logQueueSizePeriod` option when you start the IAF. IAF status information contains the following elements:

Status line element	Description
`ApEvRx`	Number of Apama events received since the IAF started. These events were received from the correlator that the IAF is connected to.
`ApEvTx`	Number of Apama events sent since the IAF started. These events were sent to the correlator that the IAF is connected to.
`TrEvRx`	Number of events received by all transports in the IAF since the IAF started. These events were received from user-defined sources outside the correlator.
`TrEvTx`	Number of events sent from all transports in the IAF since the IAF started. These events were sent to user-defined targets outside the correlator.
`vm`	Number of kilobytes of virtual memory being used by the IAF process.
`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.

For example:

```
Status: ApEvRx=589 ApEvTx=2056000 TrEvRx=2056008 TrEvTx=587 vm=407200 si=0.0 so=0.0
```

IAF log file rotation

Rotating the IAF log file refers to closing the IAF log file while the IAF is running and opening a new 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 IAF log rotation policy. You should consider

- How often to rotate log files
- How large an IAF log file can be
- What IAF 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 IAF 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 IAF started, as well as any other log files that were in use before and when a problem occurred.

On Windows, to automate log file rotation, you can set up scheduled tasks that run the following utilities:

- The following command instructs the IAF to close the log file it is using and start using a log file that has the name you specify. When you run this request to rotate the log file the IAF log file has a new name each time you rotate it. This is because Windows does not let you change the name of a file that is being used. Be sure to enclose the argument after `-r` in quotation marks.

  ```
  iaf_management -r "setLogFile new-log-filename"
  ```

- You can configure the IAF to log to two separate files. Each command instructs the IAF to start using the specified log file for either the IAF core processes (generic IAF information such as status messages) or the IAF plug-in processes (transports and codecs being used). Be sure to enclose the argument after `-r` in quotation marks.

  ```
  iaf_management -r "setCoreLogFile new-log-filename"
  iaf_management -r "setPluginLogFile new-log-filename"
  ```

  Consider using two IAF log files when you need to focus on diagnosing something specific to your application, for example, you need to easily spot authentication messages. If you do use separate log files you might want to rotate them at the same time so that they stay in sync with each other.

On UNIX, to automate log file rotation, you can write a `cron` job that periodically does any of the following:

- Set log file name

  ```
  iaf_management -r "setLogFile new-log-filename"
  ```

- Set core log file and plugin log file

  ```
  iaf_management -r "setCoreLogFile new-log-filename"
  iaf_management -r "setPluginLogFile new-log-filename"
  ```

- Reopen the log

  ```
  iaf_management -r "reopenLog"
  ```

  Move the IAF log file before you execute the `reopenLog` request. Since UNIX allows you to rename a file that is in use, the IAF processes will log to the renamed log file. When you then request the IAF to reopen its log file the IAF creates a new log file with the same name. For example, suppose you move `iaf_current.log` to `iaf_archive_2014_01_31.log` and then send a `reopenLog` request. The IAF creates `iaf_current.log`, opens it, and begins sending any log messages to it. Be sure to enclose the argument after `-r` in quotation marks.

If you are using two IAF log files, the `reopenLog` request applies to both of them. Consequently, you want to move both log files before you issue the `reopenLog` request.

- Send a `SIGHUP` signal

  You can write a `cron` job that sends a `SIGHUP` signal to IAF processes. The standard UNIX `SIGHUP` mechanism causes IAF processes to re-open their log files.

  The `cron` job should first rename log files. Since UNIX allows you to rename a file that is in use, the IAF processes will log to the renamed log files until the `cron` job sends a `SIGHUP` to IAF processes. The `SIGHUP` signal makes the processes re-open their log files and so they open files that have the old names and begin using them. Of course, these files are initially empty because the IAF must re-create them.

  Sending a `SIGHUP` signal does the same thing as the `reopenLog` request. Also, a `SIGHUP` signal forces the IAF configuration file to be reloaded and this reload stops and starts the transports and codecs.

If you instruct the IAF to open a named log file and the IAF cannot open that log file or cannot write to that log file, the IAF sends log messages to `stderr` but does not generate an error.

Apama does not support automatic log file rotation based on time of day or log file size.

IAF Management – Managing a running adapter I

The IAF Management tool is provided for performing generic component management operations on a running adapter. It can be used to shut down a running adapter, request the process ID of a running adapter, or check that an adapter process is running and acknowledging communications. Any output information is displayed on `stdout`.

See also "IAF Client – Managing a running adapter II" on page 24 for IAF-specific management information, as opposed to this generic component management tool.

The executable for interfacing with the IAF's management interface status is `iaf_management.exe` (on Windows) or `iaf_management` (on UNIX). When it is run with the `-h` command line option it displays the following usage information:

```
Usage: iaf_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>
```

IAF Management options

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

Table 1. IAF Management options

`-V`	Displays version information for the `iaf_management` tool.
`-h`	Displays usage information for running the `iaf_management` tool.
`-v`	Displays information in a more verbose manner.
`-n` *host*	Name of the host of the component that you want to connect to. The default is `localhost`. You cannot use non-ASCII characters in the host name.
`-p` *port*	Port on which the adapter is listening. The default is `16903`.
`-w`	Instructs the `iaf_management` tool to wait for the component to start. This option is similar to the `-W` option, except that the `-w` option instructs the `iaf_management` 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 `iaf_management` tool to wait *num* seconds for the component to start. If the component is not ready before the specified number of seconds has elapsed, the `iaf_management` tool terminates with an exit code of `1`.
`-N`	Displays the name of the component.
`-T`	Displays the type of the component that the `iaf_management` tool connects to.
`-Y`	Displays the physical ID of the component. This can be useful if you are looking at 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 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`.

`-C`	Displays the version of the component.
`-R`	Displays the product version of the component. For example, when the tool connects to a correlator, it displays the version of the Apama 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`	Disconnect sender with physical id <*id*>
`-xr`	Disconnect receiver with physical id <*id*>
`-I` *category*	This option is for use by technical support. It displays component-specific information for the specified category.
`-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 *req*	This option is for use by technical support. It sends a component-specific request.

`-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 log file with a `CRIT` flag, and then shuts down.

IAF Management exit status

The following exit values are returned:

Key	Value
`0`	All status requests were processed successfully
`1`	No connection to the adapter was possible or the connection failed
`2`	Other error(s) occurred while requesting/processing status
`3`	Deep ping failed

IAF Client – Managing a running adapter II

The IAF Client tool is provided for performing IAF-specific management operations on a running adapter. It can be used to stop a running adapter, to temporarily pause sending of events from an adapter into the event correlator, and to request an adapter to dynamically reload its configuration.

The executable for this tool is `iaf_client` (on UNIX) or `iaf_client.exe` (on Windows). The complete usage information (as displayed when it is run with the `--help` option) is as follows:

```
Usage: iaf_client [ options ]
Where options include:
    -h | --help                This message
    -n | --hostname <host>     Connect to an IAF on <host>
    -p | --port <port>         IAF is listening on <port>
    -r | --reload              Tell the IAF to reload its config
    -s | --suspend             Tell the IAF to suspend event sending
    -t | --resume              Tell the IAF to resume event sending
    -q | --quit                Tell the IAF to shut down
    -v | --verbose             Be more verbose
    -V | --version             Print program version info
    One of --reload, --quit, --suspend, -q, -r or -s must be specified
```

Note that if the adapter is listening for control connections on a non-standard port (specified with the `--port` option to the `iaf` program), you must pass the same port number to `iaf_client`.

Also, can only use ASCII characters to specify the name of the host.

Suspend and resume

The `--suspend` and `--resume` options control the sending of events to the correlator, not to the external transport.

Adapter reload

The `--reload` option is worthy of further explanation. Using `--reload` it is possible to dynamically reconfigure a running adapter from a changed configuration file without restarting the IAF.

When the IAF is started, it loads all the transport and codec plug-ins defined in its configuration file, and initializes them with any plug-in-specific properties provided.

When an adapter is reconfigured using `--reload`, the IAF will:

- Pass the current set of `<property>` names and values in the configuration file to each loaded transport and codec layer plug-in.

Note: Although plug-in authors will support dynamic reconfiguration of properties wherever possible, it is important to be aware that there may be some properties that by the nature can not be changed while the adapter is still running. These should be detailed in the documentation for the transport or codec plug-in. Some transport and codec plug-ins may not support configuration file reloading at all; this should be documented by the specific plug-ins.

- Load and initialize any new transport and codec layer plug-ins that have been listed in the `<transports>` and `<codecs>` sections of the configuration file.
- Unload any transport and codec layer plug-ins that are no longer listed in the `<transports>` and `<codecs>` sections of the configuration file.

Changing the `name` of a running plug-in and performing a reload is equivalent to unloading the plug-in and then loading it again. It is important to realize that this will result in any runtime state stored in memory by the plug-in being lost.

Note: It is not possible to dynamically change a loaded plug-in's C/C++ `library` filename or Java `className`, nor to change a C/C++ plug-in into a Java one (or vice-versa).

If an adapter is reconfigured to use a different log file and the IAF cannot write to the new log file when reloaded, the IAF uses the log file the adapter was using before reconfiguring. If the IAF cannot use the original log file, it writes to `stderr`.

IAF Watch – Monitoring running adapter status

The IAF Watch tool allows one to monitor the live status of a running adapter. This is available as `iaf_watch` (on UNIX) or `iaf_watch.exe` (on Windows).

The usage information for this tool, when run with the `--help` option, is as follows:

```
Usage: iaf_watch [ options ]
Where options include:
    -h | --help                 This message
    -n | --hostname <host>      Connect to an IAF on <host>
    -p | --port <port>          IAF 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
    -u | --utf8                 Write output in utf8
    -V | --version              Print program version info
```

By default the tool will collect status from the adapter once per second and display this in a human-readable form. Note that if the adapter is listening for control connections on a non-standard port

(specified with the `--port` option to the `iaf` program), you must pass the same port number to `iaf_watch`.

Note: You should only use ASCII characters to specify the name of a host.

The IAF configuration file

An IAF configuration file is an essential part of any adapter generated with the IAF.

The configuration file must be formatted in XML, and the Document Type Definition (DTD) for it is `iaf_4_0.dtd`, which is located in the `etc` directory of your installation. You may wish to use this DTD in conjunction with your XML editor to assist you in writing a correctly formatted configuration file.

The configuration file is loaded and processed when the adapter process starts. A running adapter can by signaled to reload and reprocess the configuration file at any time, by running the `iaf_client` tool with the `-r` or `--reload` option.

On UNIX platforms a `SIGHUP` signal sent to the IAF process re-opens the log file.

The root element in the configuration file is `<adapter-config>`. This must always be defined. Within it a single instance of the following elements must exist:

- `<transports>` - This element defines the transport layer plug-in(s) to be loaded.
- `<codecs>` - Defines the codec layer plug-in(s) to be loaded.
- `<mapping>` - Defines the mapping rules for the Semantic Mapper layer, which are used in the conversion between codec layer normalized messages and correlator events.

Following those elements, there must be at least one of the following elements. It is also possible to specify one of each of these elements:

- `<apama>` - Defines how the IAF connects to the Apama event correlator(s).
- `<universal-messaging>` - Defines how the IAF connects to Software AG's Universal Messaging message bus.

There are also three optional elements that can appear before these required elements (in order):

- `<logging>` - Defines the log file and logging level used by the IAF.
- `<plugin-logging>` - Defines the log file and logging level used by the transport and codec layer plug-ins.
- `<java>` - Defines the environment of the embedded Java Virtual Machine (JVM) in which any Java codec and transport plug-ins will run.

Each of these elements is discussed in more detail in the following sections.

Including other files

The adapter configuration file supports the XML `XInclude` extension so you can reference other files from the configuration file. This makes it possible, for example, to keep the transport properties in one file and the mapping properties in another. For more information on XML Inclusions, see https://www.w3.org/TR/xinclude/. The standard adapters packaged with the Apama installation use this

scheme. For example, the Apama ODBC adapter specifies its transport properties in the `adapters\config\ODBC.xml.dist` file and its mapping properties in the `adapters\config\ODBC-static.xml`. For more information on the standard Apama adapters, see "Apama File Adapter" on page 164.

In order to match the DTD, the `xmlns:xi` attribute must be placed either on the `<adapter-config>` element (as the name `xmlns:xi`) or on the `<xi:include>` element. Apama strongly recommends that you use only relative filenames instead of URLs to remote servers.

For example:

```
<adapter-config xmlns:xi="http://www.w3.org/2001/XInclude">
  <transports>
    <transport name="testmarket1" library="protocol-transport">
      <property name="host" value="localhost"/>
      <property name="port" value="12000"/>
    </transport>
  </transports>
  <xi:include href="market-static.xml"
              xpointer="xpointer(/static/codecs)"/>
  <xi:include href="market-static.xml"
              xpointer="xpointer(/static/mapping)"/>
```

Transport and codec plug-in configuration

The adapter configuration file requires both a `<transports>` and a `<codecs>` element.

The `<transports>` element defines the transport layer plug-in(s) to be loaded, and contains one or more nested `<transport>` elements, one for each plug-in.

The syntax of the `<codecs>` element mirrors the `<transports>` element precisely, and contains one or more nested `<codec>` elements, each of which defines a codec layer plug-in to be loaded.

The <transport> and <codec> elements

The transport or codec layer plug-in that should be loaded is defined by the attributes of the `<transport>` or `<codec>` elements:

- To load a C or C++ plug-in, there must be a `library` attribute, whose value is the filename of the library in which the plug-in is implemented. The extension and library name prefix will be deduced automatically based on the platform the IAF is running on. For example, on Windows `library="FileTransport"` would reference a file called `FileTransport.dll`; on a UNIX system the library filename would be `libFileTransport.so`.
- To load a Java plug-in, instead provide a `className` attribute, whose value is the fully qualified name of the Java class that implements the plug-in.

If the optional `jarName` attribute is also provided, the plug-in class will be loaded from the Java archive (`.jar`) that it specifies; otherwise the IAF will use the usual classpath searching mechanism to locate the class. See "Java configuration (optional)" on page 44 for more information about setting a classpath for use with the IAF.

- All `<transport>` and `<codec>` elements must also have a `name` attribute. The name is an arbitrary string used to reference the plug-in within the IAF, and must be unique within the configuration file. Even if the same plug-in was to be loaded more than once inside the same IAF, the corresponding `<transport>` or `<codec>` elements would still need to have different names.
- The `<transport>` and `<codec>` elements can include an optional `recordTimestamps` attribute. This attribute supports the latency framework feature. The value of the attribute determines the

values of the `recordUpstream` and `recordDownstream` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) object passed to the semantic mapper. The attribute takes one of the following values:

- `none` — Do not record any timestamps
- `upstream` —Record timestamps for upstream events only
- `downstream` — Record timestamps for downstream events only
- `both` — Record timestamps for both upstream and downstream events

The default, if the `recordTimestamps` attribute is not present, is `none`.

- The `<transport>` and `<codec>` elements can include an optional `logTimestamps` attribute. This attribute supports the latency framework feature. The attribute takes a space- or comma-separated list of keywords for its value. Supported keywords are:
  - `upstream` — Log latency for upstream events
  - `downstream` — Log latency for downstream events
  - `roundtrip` — Log roundtrip latency for all events, in a plug-in-specific way
  - *logLevel* — Set the logging level for timestamp logging. Any of the standard Apama log levels are accepted for this keyword.

The value of this attribute determines the values of the `logUpstream`, `logDownstream`, `logRoundtrip` and `logLevel` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) passed to the semantic mapper. If the `logTimestamps` attribute is not present, the log level defaults to `INFO` and the other timestamp logging parameters default to `false`.

Note: Although C/C++ and Java transport and codec layer plug-ins may coexist in the same IAF, using a C/C++ codec plug-in with a Java transport plug-in or vice-versa is not permitted.

Plug-in <property> elements

`<transport>` and `<codec>` elements may also contain any number of `<property>` elements, which are the mechanism by which plug-in-specific options are configured.

Each `<property>` element has two attributes: `name` and `value`. The syntax of the `name` and `value` is entirely determined by the plug-in author. Typically the `name`, `value` pairs are not ordered, and there is no constraint on the uniqueness of the names. Most plug-ins treat the `name` attribute in a case-sensitive manner.

In most (though not all) cases, plug-in authors allow properties to be changed dynamically without restarting the IAF, by using the IAF Client tool to request a reload of properties from the IAF configuration file. See "IAF Management – Managing a running adapter I" on page 21 for more information about using the IAF Client.

Example

The transport/codec definition section of an IAF configuration file might look as follows for a C/C++ transport plug-in implemented on Windows by `FileTransport.dll` with a codec in `StringCodec.dll`:

```
...
<transports>
  <transport name="File" library="FileTransport">
    <!-- Transport-specific configuration property -->
    <property name="input" value="simple-feed.evt"/>
    <property name="output" value="output.evt"/>
```

```
  </transport>
</transports>
<codecs>
  <codec name="String" library="StringCodec">
    <!-- Codec-specific configuration property -->
    <property name="NameValueSeparator" value="="/>
    <property name="FieldSeparator" value=", "/>
    <property name="Terminator" value=";"/>
  </codec>
</codecs>
...
```

Similarly the configuration section for the equivalent Java plug-ins, both packaged inside `FileAdapter.jar`, would be:

```
...
<transports>
  <transport
   name="File"
   jarName="JFileAdapter.jar"
   className="com.apama.iaf.transport.file.JFileTransport"
  >
    <!-- Transport-specific configuration property -->
    <property name="input" value="simple-feed.evt"/>
    <property name="output" value="output.evt"/>
  </transport>
</transports>
<codecs>
  <codec
   name="String"
   jarName="JFileAdapter.jar"
   className="com.apama.iaf.codec.string.JStringCodec"
  >
    <!-- Codec-specific configuration property -->
    <property name="NameValueSeparator" value="="/>
    <property name="FieldSeparator" value=", "/>
    <property name="Terminator" value=";"/>
  </codec>
</codecs>
...
```

You are advised to peruse the `iaf_4_0.dtd` file as it represents the complete syntactic reference to the correct structure of the configuration file.

The topic "Event mappings configuration" on page 29 describes the semantic translation and transformation rules. "IAF samples" on page 45 illustrates an example configuration file.

Event mappings configuration

The adapter configuration file requires a `<mapping>` element, which configures the adapter's Semantic Mapper layer.

The `<mapping>` element may contain the following optional attributes to support the latency framework:

- An optional `recordTimestamps` attribute. The value of the attribute determines the values of the `recordUpstream` and `recordDownstream` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) object passed to the transport or codec. The attribute takes one of the following values:
  - `none` — Do not record any timestamps
  - `upstream` — Record timestamps for upstream events only

  - `downstream` — Record timestamps for downstream events only
  - `both` — Record timestamps for both upstream and downstream events

  The default, if the `recordTimestamps` attribute is not present, is `none`.

- An optional `logTimestamps` attribute. This attribute takes a space- or comma-separated list of keywords for its value. Supported keywords are:
  - `upstream` — Log latency for upstream events
  - `downstream` — Log latency for downstream events
  - `roundtrip` — Log roundtrip latency for all events
  - *log level* — Set the logging level for timestamp logging. Any of the standard Apama log levels are accepted for this keyword.

  The value of this attribute determines the values of the `logUpstream`, `logDownstream`, `logRoundtrip` and `logLevel` members of the `IAF_TimestampConfig` object (C/C++ plug-ins) or `TimestampConfig` object (Java plug-ins) passed to the transport or codec. If the `logTimestamps` attribute is not present, the log level defaults to `INFO` and the other timestamp logging parameters default to `false`.

The `<mapping>` element may contain the following (in order):

- An optional `<logUnmappedDownstream>` element, which specifies a file to which unmapped downstream messages should be logged.
- An optional `<logUnmappedUpstream>` element, which specifies a file to which unmapped upstream Apama events should be logged.
- One or more `<event>` elements, specifying the mapping between Apama correlator events and external messages. Setting up the correct `<event>` elements is the main part of configuring the Semantic Mapper.
- One or more `<unmapped>` elements, which specify events that will bypass the Semantic Mapper, using a string representation of the entire Apama event.

Note: The order in which `<event>` and `<unmapped>` elements appear can be mixed.

Each of these will be discussed in more detail below, after a brief explanation of the operation of the Semantic Mapper.

Semantic Mapper operation

The IAF Semantic Mapper takes as input a set of rules that specify when and how an Apama event can be generated from an external message, and similarly how suitable messages of the correct external format should be constructed from Apama events. These rules are termed an event mapping.

All Apama events must belong to a named event type that defines their structure. On startup, the IAF will parse each `<event>` element, derive the structure of the event being described, and optionally inject an EPL event definition for it into the event correlator. The event mappings are therefore organized by Apama event type, as `<event>` elements.

When an external message is received from a codec plug-in, the Semantic Mapper will run it past each event mapping sequentially, in the order provided in the configuration file. First it checks whether it matches a set of conditions specified within that mapping. If it does, it proceeds to transform and translate it according to the mapping rules provided. If it does not match the

conditions, the Semantic Mapper will move on to the next event mapping. If the message matches against several <event> mappings, only the first mapping is executed unless the breakDownstream attribute is set to false. When this attribute is set to false, all mappings that match are executed.

In the upstream direction, when the Semantic Mapper receives an Apama event, it will already know the type of the event, because this information is part of each event sent out by the event correlator. However, it is possible to specify multiple upstream mappings from the same Apama event type; therefore, just as with downstream mappings, the Semantic Mapper will check the incoming Apama event against the conditions defined in each of the mappings for that event type. Just as for downstream mappings, the first matching mapping will be used, unless the breakUpstream attribute is set to false. When this attribute is set to false, all mappings that match are executed.

In both directions it is possible for an incoming event not to match against any event mapping, and in which case no mapping is executed. The <logUnmappedDownstream> and <logUnmappedUpstream> elements allow such messages and events to be logged.

The <logUnmappedDownstream> and <logUnmappedUpstream> elements

These optional elements enable the logging of Apama events and codec messages that were not matched by any of the configured mapping rules, before they are discarded by the adapter. This can be useful for debugging and diagnostics.

The <logUnmappedDownstream> element turns on logging of messages from an external event source that were not mapped onto an Apama correlator event, after being received from a codec plug-in; the <logUnmappedUpstream> element enables logging of events from the correlator that did not match the conditions necessary for mapping to an external message that could be passed on to a codec plug-in.

Both elements have a single attribute called file, which is used to specify the filename that the log should be written to.

For example:

```
<mapping>
  ...
  <logUnmappedDownstream  file="unmapped_from_adapter.log"/>  <logUnmappedUpstream
   file="unmapped_from_Correlator.log"/>  ...
</mapping>
```

Note: Due to buffering of files in the operating system, the contents of the log files on disk may not be complete until the IAF is shutdown or reconfigured.

The <event> element

The <mapping> section contains one or more <event> elements, each of which specifies a mapping between an Apama correlator event type and a kind of external message. Setting up the correct <event> elements is the main part of configuring the Semantic Mapper.

Each <event> element can have the following attributes:

- name – This is the name of this Apama correlator event type, and is required.
- package – This optional attribute specifies the EPL package of the Apama event; if it is not provided, the default package is used. See *Developing Apama Applications in EPL* for more information about packages.
- direction – This optional attribute defines whether this event mapping is to be used solely for downstream mapping (from incoming external messages to Apama events), upstream mapping (from Apama events to outgoing messages) or for mapping in both directions. The allowed

values for the attribute are `upstream`, `downstream` and `both`. The default value if the attribute is undefined is `both`.

- `encoder` – Required for a mapping that can be used in the upstream direction (`direction ="upstream"` or `"both"`), but ignored when processing downstream messages. In the upstream direction the attribute specifies the codec plug-in that should be used to process the message, once the translation process is complete. The name supplied here must match the `name` provided in the `<codec>` element.
- `copyUnmappedToDictionaryPayload` – This optional Boolean attribute defines what the Semantic Mapper should do with any fields in the incoming messages that do not match with any field mapping. If `copyUnmappedToDictionaryPayload` is `false` (and `copyUnmappedToPayload`, below, is also `false` or not present), then any unmapped fields are discarded. If it is set to `true` however, they will be packaged into a special field called `__payload`, implicitly added as the last field of the Apama event type. Fields in normalised events with a value of null will be included in the dictionary with the value set to an empty string. If this attribute is undefined its value defaults to false.

  Using `copyUnmappedToDictionaryPayload` puts all the payload fields in a standard EPL dictionary that is efficient and easy to access. See "Creating a payload field" on page 135 for more information about the payload field.
- `copyUnmappedToPayload` – As of Release 4.1, this option is deprecated; use `copyUnmappedToDictionaryPayload` instead. This optional Boolean attribute defines what the Semantic Mapper should do with any fields in the incoming messages that do not match with any field mapping. If `copyUnmappedToPayload` is `false` (and `copyUnmappedToDictionaryPayload`, above, is also `false` or not present), then any unmapped fields are discarded. If it is set to `true` however, they will be packaged into a special field called `__payload`, implicitly added as the last field of the Apama event type. The default value if this attribute is undefined is `false`. See "Creating a payload field" on page 135 for more information about the payload field.
- `breakUpstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an upstream Apama event to an event mapping.

  When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next event. This is the default behavior when the `breakUpstream` attribute is not specified.

  When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same event against the other event mappings.

  This attribute only affects upstream event processing.
- `breakDownstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an incoming downstream message to an event mapping.

  When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next message. This is the default behavior, when the `breakDownstream` attribute is not specified.

  When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same message against the other event mappings.

  This attribute only affects downstream event processing.
- `inject` – Determines whether the IAF will automatically inject the event definitions that are implicitly defined by this event mapping into the correlator. The default is `false`. Injecting events from the configuration files is deprecated. Instead, you should use the `-e` or `--events` switches to `iaf` to generate the EPL code for the event definitions and then inject the events (and monitors)

during the application's start-up sequence with the tool that you use to start the correlator, such as Apama Studio, the Enterprise Management and Monitoring console (EMM) or the Apama command line tools.

- `copyTimestamps` – This is an optional attribute. If set to `true` (the default is `false`), the semantic mapper will add an additional field to the generated event definition to hold timestamp information. The new field will be called `__timestamps` and will be of type `dictionary<integer,float>` where the dictionary keys are timestamp indexes and the values are the corresponding timestamps. The timestamp field is inserted before the payload field, so it may be the last or next to last field in the event definition. If timestamp copying is enabled for an event type, all timestamps present in the `__timestamps` field of a matching upstream event will be copied into a new `AP_TimestampSet`/`TimestampSet` object and passed to the upstream codec. Likewise, any timestamps passed to the semantic mapper by the codec will be copied into the `__timstamps` field of the outgoing downstream event and thus made available to the correlator.

- `transportChannel` — optional. If present, then for upstream events (events leaving the correlator), the channel is put in the `NormalisedEvent` using the value of the `transportChannel` attribute.

  If present, then for downstream events (events going into the correlator), if the value of the `transportChannel` attribute is in the `NormalizedEvent`, then that value from the `NormalizedEvent` is used as the channel name. It is possible that a subsequent `<map>` element with an identical `transport` attribute value could override it.

- `presetChannel` - optional. If present, then for downstream events (events going into the correlator), if no channel has been set by the `transportChannel` attribute, then the value of `presetChannel` is used as the channel name.

  If `transportChannel` is set, then that value in the `NormalisedEvent` can still be used for a normal `<map>` rule, but it will not appear in the `unmappedDictionary` (if present).

  Thus, it is possible to define either a default channel name per type, or a `NormalisedEvent` field that the transport will send and receive, and this could be re-using a `NormalisedEvent` field used by a `<map>` element.

A typical bidirectional `<event>` element might look like the following. For downstream, if `"CHANNEL"` (from `transportChannel`) is in the `NormalisedEvent`, then the value of the `"CHANNEL"` entry is used as the channel name, otherwise `"channelB"` from `presetChannel` is used. For upstream, the channel name is placed in the `"CHANNEL"` entry in the `NormalisedEvent`.

```
<event name="Tick"
       direction="both"
       encoder="String"
       copyUnmappedToDictionaryPayload="true"
       inject="false"
       presetChannel="channelB"
       transportChannel="CHANNEL">
   <id-rules>
     ...
   </id-rules>
   <mapping-rules>
     ...
   </mapping-rules>
</event>
```

The `<id-rules>` and `<mapping-rules>` elements are described below.

The <event> mapping conditions

The `<id-rules>` element defines a set of conditions that must be satisfied by an incoming message for it to trigger the mapping to an Apama event, or to decide how to map an incoming Apama event

back to a normalized message. The `<id-rules>` element contains `<upstream>` and `<downstream>` sub-elements, which in turn contain the mapping conditions to be used when the Semantic Mapper is searching for a mapping to use in the upstream or downstream direction, respectively. Each condition is encoded in an `<id>` element.

Note: Conditions are only required for the directions that the mapping can operate in. For example, a mapping with `direction="downstream"` does not need any `<upstream>` id rules, while a mapping with `direction="both"` must specify both `<upstream>` and `<downstream>` id rules. The `<id-rules>`, `<upstream>` and `<downstream>` elements themselves must exist though.

`<id>` - Each `<id>` sets a condition on a set of fields contained in the normalized message or Apama event. This element takes up to three attributes; `fields`, which defines the fields that the condition must apply to; `test`, which specifies the condition; and `value`, which provides a value to compare the field value with. The value attribute is only required for relational tests. For example:

```
<id fields="Stock, Exchange, Price" test="exists"/>
```

specifies that the `Stock`, `Exchange` and `Price` fields must exist if the condition is to be satisfied and the mapping proceed. No `value` is needed to perform the test in this case.

However, the following example:

```
<id fields="Exchange" test="==" value="LSE"/>
```

specifies that the `Exchange` field must exist and have the value "LSE" for the condition to be satisfied.

Note: The value for the `fields` attribute is a list of fields, delimited by spaces or commas. This means, for example that `<id fields="Exchange EX,foo" test="==" value="LSE"/>` will successfully match a field called "Exchange", "EX" or "foo", but *not* a field called "Exchange EX,foo". You should keep this in mind when you assign field names for normalized events. While fields with spaces or commas in their names may be included in a payload dictionary in upstream or downstream directions, they cannot be referenced directly in mapping or id rules.

The following test conditions may be specified. The first four tests are unary operators that do not need a `value` attribute. These tests can be applied to multiple fields in the same `<id>` rule:

- `exists` - The fields exist, but do not necessarily have a value
- `notExists` - The fields do not exist
- `anyExists` - One or more of the fields exist, not necessarily with values
- `hasValue` - At least one of the fields exists and has a value. To test that multiple fields all exist and all have values, use multiple `hasValue` conditions, one for each field, such as

```
<id fields="symbol" test="hasValue"/>
<id fields="newLimit" test="hasValue"/>
```

The remaining tests are binary relational operators that do require a `value` attribute. Furthermore, these tests can only be applied to single fields:

- `==` - Case-sensitive string equality
- `!=` - Case-sensitive string inequality
- `~==` - Case-insensitive string equality
- `~!=` - Case-insensitive string inequality

While the equality tests will fail on fields with missing values, the inequality tests will pass.

All <id> conditions in an <id-rules> element must be satisfied for the mapping to proceed.

Note: A mapping with no <id> elements will always match. This allows a catch-all mapping to be specified. This should be the last definition.

The id rules that test transport field values function in isolation from each other, that is, as soon as a test id rule fails, the mapper stops looking at subsequent rules. This means there is no way to group together tests against the same field name with an OR condition to see if any of them match. Any type of OR value testing needs to be implemented at the codec or EPL layers, or by creating copies of the entire <event> element for each value to test.

The following is an example of a valid <id-rules> element for a bidirectional mapping. Note that the upstream rules are empty, so this mapping will match any incoming Apama event of the appropriate type:

```
<id-rules>
  <downstream>
  <id fields="Stock, Price" test="exists"/>
  <id fields="Exchange" test="==" value="LSE"/>
 </downstream>
 <upstream/>
</id-rules>
```

The <event> mapping rules

The <mapping-rules> element defines a set of mappings that describe how to create an Apama event from an incoming message. Conversely they define the mapping from an Apama event to an outgoing message. Each mapping must be defined in a <map> element, which has the following attributes:

- apama – This is the Apama event field name to copy the value into (downstream) or to take the value from (upstream). This attribute is optional. In an upstream direction, if the apama attribute is not specified or is provided empty, a field will be created within the external message and set to the value specified by default. Not specifying the apama attribute has no significance in a downstream direction — this line of the mapping will be ignored.

  Note: The IAF does not know what types are injected into the correlator, and will drop events with a Failed to parse warning if the *types* and *order* of the elements with an apama= attribute do not match the event definition that was injected into the correlator. Mapping rules that do not specify an apama= attribute are not affected by this.

- transport – This defines the external message's field name to copy the value from (downstream) or to copy the value into (upstream). For a downstream mapping it is possible to define more than one value here; in which case the first encountered is used. This attribute is optional. In a downstream direction, if the transport attribute is not specified or is provided empty, a field will be created within the Apama event and set to the value specified by default. Not specifying the transport attribute has no significance in an upstream direction — this line of the mapping will be ignored.

- type – The type of the field in the Apama event type. Any simple correlator type is valid here (string, integer, decimal, float, boolean and location); for complex correlator reference types such as sequence<...> and dictionary<...,...>, specify reference instead. If a field is of reference type, the referenceType attribute can be supplied if needed to define the type (see below). When a field is of reference type, the Semantic Checker passes its string form to and from the codec untouched, and performs no checking upon the validity of the value. Note that fields in the external message are always un-typed character strings, regardless of any type they may have had on the external transport that produced them. Furthermore, the Semantic Mapper does not

perform any type "casting" or "coercion" when converting a character string in an external event field to the appropriate Apama type, meaning that the Apama event produced might be invalid and be rejected by the correlator. Conversions in the upstream direction, from the Apama field to a string in the external event, will always succeed. Codec and transport plug-ins should be aware of these rules when working with events that will be, or have been, processed by the Semantic Mapper.

- `referenceType` – This is an optional attribute, but it must be supplied if the attribute `type="reference"` and the event of which this field is a member has the attribute `inject="true"` (which is now deprecated) or if the IAF will be run with the `-e` option in order to generate a EPL file with event definitions. This EPL file is then injected to the correlator (this is the recommended method of injecting events). The value of this attribute must be a valid correlator type. This attribute is only used for the process of constructing the event definitions that are to be injected into a correlator. Note that since this is an XML attribute value, some characters such as angle brackets or quotation marks must be correctly encoded using their XML entity name. For example, a `sequence<string>` must be written as `referenceType="sequence &lt;string&gt;"`. Note also that when nesting sequences within sequences, a space must be present between the angle brackets to prevent the correlator from parsing this as a bitwise shift.
- `default` – The default value to set the Apama field to if the external field specified in `transport` is missing. Note that the value provided must be of the type specified in `type`. For example, a valid string is "`test`" or "", a valid integer is "`0`", a valid decimal is "`0.0`" or "`0.0d`", a valid float is "`0.0`", a valid boolean is "`true`" or "`false`", and a valid location is "`(0.0, 0.0, 0.0, 0.0)`".
- `defaultIfEmpty` – Optionally, sets the default value to assign to the field in the Apama event if the external field specified in `transport` is present but has no value defined. The same type considerations apply as for the `default` attribute. If this attribute is not defined the value specified by the `default` attribute will apply for this condition as well. Bearing in mind angled brackets and quotes have to be written as XML entities (see above), for a nested event you need to write `default="InnerEvent(&quot;&quot;)"` if you want the default value to be `InnerEvent("")`.

Note that at least one of `apama` and `transport` must be specified in a mapping.

The following is an example of a valid `<mapping-rules>` element:

```
<mapping-rules>
  <map apama="stockName" transport="Stock" type="string" default=""/>
  <map apama="stockPrice" transport="Price" type="float" default="0.0"/>
  <map apama="stockVolume" transport="Volume, TradingVolume,
    CombinedVolume" type="float" default="0.0"/>
</mapping-rules>
```

In a downstream direction, this specifies that the `Stock` field must be copied over into `stockName`, `Price` must be copied into `stockPrice`, and the first encountered of `Volume`, `TradingVolume` or `CombinedVolume` must be copied into `stockVolume`. First encountered means the first such instance when the event is parsed left to right.

In an upstream direction, the Apama field values would be copied into external message fields of the names specified. A given field in an upstream message can be generated from several different sources. These are evaluated in the following order:

1. A `<map>` rule mapping from a named Apama event field to the transport field.
2. A value for the transport field in the event payload. See "The Event Payload" on page 135 for more details on using the event payload.
3. Any default value available from a `<map>` rule with no corresponding Apama event field.

You may have noticed that while an Apama event must always have its full complement of fields defined and with type-valid values, the same is not assumed of external events.

Note: If multiple upstream mappings for the same Apama type exist, they must all specify all of the fields in the type in the same order, with the same type values.

Note: Tips for writing a codec when using reference types:

- A codec is responsible for constructing the string form of any value. This means that if your event contains a `sequence<string>` then the codec must generate an entry in the normalized event whose value is of the form:

  ```
  ["string value 1", "string value 2", "Value with a \" and backslash \\"]
  ```

- If the codec generates an event that the correlator cannot parse, the correlator will drop the event and the codec will have no way of knowing. Be careful constructing the event strings.
- Similarly, events from the correlator will contain a normalized event entry whose value is the string from of the field's value, as in the example above. The codec is responsible for parsing these strings.
- When writing adapters in Java, Apama suggests you use the classes in the `com.apama.event.parser` package to parse and construct the strings to send to the Semantic Mapper. If you are writing a C/C++ adapter, the corresponding functions for parsing and constructing strings to send to the Semantic Mapper are found in the `AP_EventParser.h` and `AP_EventWriter.h` header files.

For more information on the Java classes, see "Working with normalized events" on page 106 as well as the Apama `Javadoc`. For more information on the C/C++ functions, see "Codec utilities" on page 74.

- If nesting other events in the fields of an event, caut
- ion must be exercised regarding package namespaces. Always use the fully qualified event name when referencing it in the string form. Also always ensure that the correlator has the enclosed event type defined before the enclosing event type.

The <unmapped> element

The `<mapping>` section may contain one or more `<unmapped>` elements, each of which specifies a mapping between the string representation of an Apama event type and a normalized event.

Each `<unmapped>` element can have the following attributes:

- `name` – This optional attribute specifies the name of the Apama correlator event type to match. If omitted, matches all Apama event types.
- `package` – This optional attribute specifies the EPL package of the Apama event. This attribute can be specified only if the `name` attribute is also supplied.
- `transport` – This attribute is required; it specifies the field in the `NormalisedEvent` to map to.
- `direction` – This optional attribute defines whether this event mapping is to be used solely for downstream mapping (from incoming external messages to Apama events), upstream mapping (from Apama events to outgoing messages) or for mapping in both directions. The allowed values for the attribute are `upstream`, `downstream` and `both`. The default value if the attribute is undefined is `both`.

- `encoder` – Required for a mapping that can be used in the upstream direction (`direction ="upstream"` or `"both"`), but ignored when processing downstream messages. In the upstream direction the attribute specifies the codec plug-in that should be used to process the message, once the translation process is complete. The name supplied here must match the `name` provided in the `<codec>` element.
- `breakUpstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an upstream Apama event to an event mapping.

  When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next event. This is the default behavior when the `breakUpstream` attribute is not specified.

  When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same event against the other event mappings.

  This attribute only affects upstream event processing.
- `breakDownstream` – This optional Boolean attribute defines what the Semantic Mapper should do when it matches an incoming downstream message to an event mapping.

  When set to `true`, the semantic mapper stops the evaluating process and begins executing the actions specified by that mapping and then goes on to evaluate the next message. This is the default behavior, when the `breakDownstream` attribute is not specified.

  When set to `false`, the semantic mapper executes the actions specified in the mapping and then keeps evaluating the same message against the other event mappings.

  This attribute only affects downstream event processing.
- `transportChannel` — optional. If present, then for upstream events (events leaving the correlator), the channel is put in the `NormalisedEvent` using the value of the `transportChannel` attribute.

  If present, then for downstream events (events going into the correlator), if the value of the `transportChannel` attribute is in the `NormalizedEvent`, then that value from the `NormalizedEvent` is used as the channel name. It is possible that a subsequent `<map>` element with an identical `transport` attribute value could override it.
- `presetChannel` - optional. If present, then for downstream events (events going into the correlator), if no channel has been set by the `transportChannel` attribute, then the value of `presetChannel` is used as the channel name.

  If `transportChannel` is set, then that value in the `NormalisedEvent` can still be used for a normal `<map>` rule, but it will not appear in the `unmappedDictionary` (if present).

  Thus, it is possible to define either a default channel name per type, or a `NormalisedEvent` field that the transport will send and receive, and this could be re-using a `NormalisedEvent` field used by a `<map>` element.

In the following example, for downstream, if `"CHANNEL"` (from `transportChannel`) is in the `NormalisedEvent`, then the value of the `"CHANNEL"` entry is used as the channel name, otherwise `"channelB"` from `presetChannel` is used. For upstream, the channel name is placed in the `"CHANNEL"` entry in the `NormalisedEvent`.

```
<unmapped
      name="Unmapped"
      direction="both"
      package="com.apama.sample"
      transport="Apama"
      encoder="$CODEC$"
```

```
      presetChannel="channelB"
      transportChannel="CHANNEL">
   <id-rules>
      <downstream>
         <id fields="Apama" test="exists"/>
      </downstream>
   </id-rules>
</unmapped>
```

The <unmapped> mapping conditions

An `<unmapped>` element must have an `<id-rules>` element that defines a set of conditions an incoming must satisfy in order to trigger the mapping to an Apama event. If the value of the `direction` attribute of an `<unmapped>` element is "both" or "downstream," the `<id-rules>` element must contain a `<downstream>` sub-element. The `<downstream>` sub-element contains conditions to be used by the Semantic Mapper when the message is moving downstream direction. Each condition is encoded in an `<id>` element.

Each `<id>` sets a condition on a set of fields contained in the normalized message or Apama event. This element takes up to three attributes; `fields`, which defines the fields that the condition must apply to; `test`, which specifies the condition; and `value`, which provides a value to compare the field value with. The value attribute is only required for relational tests.

The `<unmapped>` entries behave in the same way as `<event>` entries — the IAF processes `<event>` and `<unmapped>` entries in order, translating events with any that match, and ending at the first entry that has `breakUpstream` or `breakDownstream` set to true or not specified (they both default to true).

Apama event correlator configuration

The adapter configuration file requires an `<apama>` element, which configures how the IAF connects to the Apama event correlator(s). An `<apama>` element can contain the following elements in the following order:

- `<sinks>` — This element lists the Apama correlators that the IAF needs to connect with in order to inject EPL event type definitions and events. You can specify the following attribute in a `<sink>` element:

  `parallelConnectionsLimit` — optional —The default behavior is that the IAF limits itself to an internally set number of connections with each specified sink. This number scales according to the number of CPUs that the IAF detects on the host that is running the IAF. While this number is usually sufficient, there are some situations in which you might want to change it. For example, if you are trying to conserve resources you might want to limit the number of connections to 1, or if you want to prevent multiple threads from sharing a connection you might allow a higher number of connections than the default allows. See the information below about multiple connections from IAF to correlator.

  Each correlator is defined in its own `<sink>` element:

  - `<sink>` — This element defines an event correlator that the IAF must send events to. You can define more than one `<sink>` element. All sinks specified will be injected with any EPL event type definitions that are defined in `<event>` elements in the configuration file. The following attributes are allowed in `<sink>` elements:

    `host` —required — Defines the name or address of the host machine where the correlator is running.

`port` — required — Specifies the port that this correlator can be contacted on.

`sendEvents` — optional — The default behavior is that all sinks receive all events generated by the Semantic Mapper. To prevent the Semantic Mapper from sending all events to a particular correlator, add `sendEvents="false"` to the `<sink>` element that defines that correlator. No events will be sent to that correlator regardless of any channel settings.

- `<sources>` — This element lists the Apama components (usually event correlators) from which the IAF can receive events. Each component is defined in its own `<source>` element:
  - `<source>` — This element defines an Apama component that the IAF needs to register with as an event consumer. This enables the IAF to receive any alerts generated by the specified component. The following attributes are allowed in `<source>` elements:

    `host` —required — Defines the name or address of the host machine where the Apama component is running.

    `port` — required — Specifies the port that this Apama component can be contacted on.

    `channels` — required — Specifies the channels that the IAF should listen on to receive events. An empty string indicates that the IAF receives all generated events. To receive events on only particular channels, specify a comma-separated list of channel names. Do not include any spaces. For example, `channels="UK,USA,GER"`.

    `disconnectable` — Specifies whether or not the IAF can be disconnected if it is slow. If set to `yes` or `true` (case insensitive), the IAF can be disconnected. Any other setting specifies that it cannot be disconnected.

It is possible to define the IAF as having only sinks, or only sources, or both, or neither. If the IAF has been started with no sinks and no sources, you would use the `engine_connect` tool to connect it to a correlator or another IAF.

For complete information on `engine_connect`, see "Event correlator pipelining" in *Deploying and Managing Apama Applications*.

Disabling Apama messaging and using UM instead

A deployed adapter can use Software AG's Universal Messaging (UM) message bus in place of connections specified in the `<apama>` element. When you want to use UM instead of explicitly set connections do the following:

- Add a `<universal-messaging>` element in place of or after the `<apama>` element. See Configuring adapters to use UM.
- Add the `enabled` attribute to the `<apama>` element, if you have one, and set it to `false`. For example:

  ```
  <apama enabled="false">.....</apama>
  ```

  Alternatively, you can remove the `<apama>` element.

When the `enabled` attribute is set to `"false"` then the entire `<apama>` element is ignored. In other words, the deployed adapter does not use any connections specified in the `<apama>` element. Instead, the deployed adapter uses the UM configuration specified in the `<universal-messaging>` element.

The default is that the `enabled` attribute is set to `true`. If the `enabled` attribute is not specified or if it is set to `true` then the connections specified in the `<apama>` element are used.

While specifying both an `<apama>` element and a `<universal-messaging>` element in an adapter configuration file is permitted, it is not recommended.

See also "Using Universal Messaging" in *Deploying and Managing Apama Applications*.

Multiple connections from IAF to correlator

To improve performance, an IAF transport might use multiple threads to send events to the codec and thus to the Semantic Mapper. If more than one thread is sending events downstream (IAF to correlator) then for each thread, the IAF creates a new connection to each `<sink>` defined in the configuration file, up to the defined limit. Thus, multiple threads can deliver events in parallel to the same sink. In combination with the `channelTransport` attribute on events (defined in `<event>` elements), threads can deliver events to different channels to be received by different contexts. For optimal parallel event delivery, each IAF transport thread should send events on a distinct set of channels. There are no ordering guarantees when different threads deliver events to the same sink.

There is a limit on how many connections to each sink the IAF can create. The IAF logs the limit for the number of connections in the startup stanza. If events are sent on more threads than the number of allowed connections, then the IAF re-uses existing connections, which means that some threads share connections. If a thread terminates, the connection it is using is not closed since it might be in use by another thread. See the information above for the `parallelConnectionsLimit` attribute on the `<sink>` element.

Example

Following is an example of an `<apama>` element:

```
<adapter-config>
  ...
     <apama>
          <sinks>
               <sink host="localhost" port="15903" parallelConnectionsLimit="1"/>
          </sinks>
          <sources>
               <source host="localhost" port="15903" channels="MY_ADAPTER"/>
          </sources>
     </apama>
</adapter-config/>
```

Configuring adapters to use UM

If you are configuring your Apama application to use Software AG's Universal Messaging, you can configure an adapter to use the UM message bus to send and receive events. To do this, add a `<universal-messaging>` element to your adapter configuration file. A `<universal-messaging>` element can replace or follow the `<apama>` element.

A `<universal-messaging>` element contains:

- Required specification of the `realms` attribute OR the `um-properties` attribute.
- Optional specification of the `defaultChannel` attribute.
- Optional specification of the `enabled` attribute, which indicates whether the deployed adapter uses the configuration specified in this `<universal-messaging>` element.
- Optional specification of a `<subscriber>` element.

Specification of realms or um-properties attribute

Specification of the `realms` attribute OR the `um-properties` attribute is required.

The `realms` attribute can be set to a list of `RNAME`s (UM realm names) to connect to. You can use commas or semicolons as separators.

Commas indicate that you want the adapter to try to connect to the UM realms in the order in which you specify them here. Semicolons indicate that the adapter can try to connect to the specified UM realms in any order. See for more information.

If you specify more than one `RNAME`, each UM realm you specify must belong to the same UM cluster. Specification of more than one UM realm lets you benefit from failover features. See "Communication Protocols and RNAMEs" in the Universal Messaging documentation.

The `um-properties` attribute can be set to the name or path of a properties file that contains UM configuration settings. See .

Specification of defaultChannel attribute

Specification of the `defaultChannel` attribute is optional. If specified, set the `defaultChannel` attribute to the name of a UM channel. You cannot specify an empty string. In other words, the value of the `defaultChannel` attribute cannot be the default Apama channel, which is the empty string.

An adapter that uses UM must send each event to a named channel. An adapter that is configured to use UM identifies the named channel to use as follows:

1. If the `transportChannel` attribute is set for an event type (in an `<event>` or `<unmapped>` element) then this is the channel the adapter uses for that event type.
2. If the `transportChannel` attribute is not set for an event type but the `presetChannel` attribute is set then this is the channel the adapter uses for that event type.
3. If neither `transportChannel` nor `presetChannel` is set for an event type then the adapter uses the channel set by the `defaultChannel` attribute in the `<universal-messaging>` element.
4. If neither `transportChannel` nor `presetChannel` is set and you did not explicitly set `defaultChannel` and you used Apama Studio to create the adapter configuration file then the `defaultChannel` attribute is set to `"`*`adapter_nameadapter_instance_id`*. For example: `"File Adapter instance 3"`.
5. If none of `transportChannel`, `presetChannel`, or `defaultChannel` are set and if you did not use Apama Studio to create the adapter configuration file then the adapter fails if it tries to use UM.

All events sent by the adapter on channels that are UM channels are delivered to those channels.

Specification of a value for the `defaultChannel` attribute affects events that are sent from this adapter to Apama engine clients, and from this adapter to correlators when the adapter connects to that correlator by means of the `engine_connect` correlator utility.

Specification of the enabled attribute

Optional specification of the `enabled` attribute, which indicates whether the deployed adapter uses the configuration specified in this `<universal-messaging>` element. The default is that the `enabled` attribute is set to `"true"`. If the `enabled` attribute is not specified or if it is set to `"true"` then the configuration specified in the `<universal-messaging>` element is used.

If `enabled` is set to `"false"` then the deployed adapter ignores the `<universal-messaging>` element and does not use UM. The deployed adapter uses only its explicitly set connections.

Specification of subscriber element

Optional specification of a `<subscriber>` element. If specified, the `<subscriber>` element must specify the `channels` attribute. Set the `channels` attribute to a string that specifies the names of the UM channels this adapter receives events from. Use a comma to separate multiple channel names.

Subscribing to receive events from an adapter that is using UM

In each context, in any correlator, that is listening for events from an adapter that is using UM, at least one monitor instance must subscribe to the channel or channels on which events are sent from the adapter. For example, if you are using an ADBC adapter, you must include a `monitor.subscribe(`*`channelName`*`)` command for the corresponding instance of the ADBC adapter. Note that not all adapter service monitors support access from multiple correlators. If this is the case, then only one correlator should run the service monitors for that adapter.

Adapter configuration examples

Following are some examples of `<universal-messaging>` elements:

```
<universal-messaging
      realms="nsp://localhost:5629"
      defaultChannel="orders"
      enabled="true">
   <subscriber channels="UK, US, GER"/>
</universal-messaging>
<universal-messaging um-properties="UM-config.properties">
   <subscriber channels="signal,forward"/>
</universal-messaging>
```

Logging configuration (optional)

The `<logging>` and `<plugin-logging>` optional elements define the logging configuration used by the adapter. If present, they must appear as the first elements nested in the `<adapter-config>` element.

The `<logging>` element configures the logging for the IAF itself, whereas the `<plugin-logging>` element configures logging for the transport and codec layer plug-ins in the adapter.

Both elements have two attributes:

- The `level` attribute sets the logging verbosity level; it must be one of the strings `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `CRIT`, or `OFF` (with the same case).
- The `file` attribute determines the file that logging messages will be written to. This should be a file path relative to the directory that the adapter was started from, or one of the special values `stdout` or `stderr` which to log to standard output or standard error, respectively.

If the IAF cannot write to the specified log file, the adapter will fail to start.

Note: If the `--logfile` and `--loglevel` command line switches are passed to the IAF executable when this is run, the `<logging>` and `<plugin-logging>` elements are ignored.

An example of these elements is provided below:

```
<adapter-config>
  <logging        level="INFO"  file="iaf.log" />  <plugin-logging
level="DEBUG" file="plugins.log" />  ...
</adapter-config>
```

If logging is not configured explicitly in the configuration file or with command line options, logging defaults to the INFO level on the standard error stream.

The <logging> element accepts two optional sub-elements, <upstream-events> and <downstream-events> that can be configured to log details of all events sent to and received from a connected correlator, without needing to set the entire IAF to DEBUG level logging. These sub-elements each take a single attribute (level) whose values specifies the logging level to be used to log upstream and downstream events, respectively. If this log level is equal to or greater than the IAF logging level, details of the events will appear in the IAF log file. For example:

```
<logging level="WARN">
  <upstream-events level="ERROR"/>
  <downstream-events level="INFO"/>
</logging>
```

In this configuration, upstream events will be logged (because ERROR is greater than WARN) but downstream events will not (because INFO is less than WARN). If either of the upstream or downstream event logging levels is not explicitly set, it will default to DEBUG (so events will not be logged by default, unless the IAF is explicitly configured for DEBUG logging)

Java configuration (optional)

Transport and codec plug-ins written in Java are executed by the IAF inside an embedded Java Virtual Machine (JVM). The optional <java> element allows the environment of this JVM to be configured.

The <java> element may contain zero or more of the following nested elements:

- <classpath> – This element adds a single entry onto the JVM classpath, which is the list of paths used by Java to locate classes. Each <classpath> element has a single path attribute that specifies a directory or Java Archive file (.jar) to add to the classpath.

  The full classpath used by the IAF's JVM is made up by concatenating (in order):

  1. the contents of the APAMA_IAF_CLASSPATH environment variable if one is defined,
  2. each of the path entries specified by <classpath> elements, in the order they appear in the configuration file, OR if there are none, the contents of the CLASSPATH environment variable,
  3. the path of the lib/JPlugin_internal.jar file used internally by the IAF.

  Additionally, if a jarName attribute is used in the <codec> or <transport> element that defines a plug-in (as in "Transport and codec plug-in configuration" on page 27), the plug-in will be loaded using a new classloader with access to the specified Java Archive in addition to the JVM classpath.

  You should make sure that all shared classes are in a separate jar that is specified by a <classpath> element. The shared classes are then loaded by the parent classloader. This ensures that when a codec or transport references a shared class, they both agree it is the same class.

- <jvm-option> - This element allows arbitrary JVM command line options to be specified. The JVM option should be placed between the start and end jvm-option tags, e.g.

```
<jvm-option>-Xmx256m</jvm-option>
```

See the usage screen of the JVM's Java executable for a full list of supported options.

- `<property>` - This element specifies a Java system property that should be passed to the JVM. It has `name` and `value` attributes, such that using:

```
<property name="propName" value="propValue"/>
```

is a shorthand equivalent to:

```
<jvm-option>-DpropName=propValue</jvm-option>
```

See "The IAF runtime" on page 17 for a description of how the IAF selects the JVM library to use.

The properties specified in the `<java>` element cannot be changed once the JVM has been loaded by the IAF. This will occur when the IAF reads a configuration file that specifies a Java transport or codec plug-in. If the same IAF process is later reconfigured to use only C/C++ transports, the JVM will *not* be unloaded. The IAF will log a warning message if a reconfiguration of the IAF process attempts to change the previously configured JVM properties.

IAF samples

Your distribution contains two complete examples that demonstrate how the IAF can be used in practice – C and Java implementations of a text file adapter, including build scripts and complete source code.

See "Standard Plug-ins" on page 137 for information about how the sample plug-ins could be used in practice.

The C example

The C example is available in `samples\iaf_plugin\c\simple` and contains the following:

- The complete source code of the `FileTransport` transport layer plug-in and the `StringCodec` codec plug-in, in the `FileTransport.c` and `StringCodec.c` files.
  - The `FileTransport` transport layer plug-in can read and write messages from and to a text file. This makes it a useful tool in testing the IAF and the event correlator with files of sample messages.
  - The `StringCodec` codec plug-in can decode messages represented as strings containing a list of field names and values. The configuration properties for the plug-in allow customization of the syntactic characters used as field, name, and message separators (e.g. “,”, “=”, “;”).
- A `Makefile` for compiling the plug-in sources with GNU Make on UNIX. This builds `libFileTransport.so` and `libStringCodec.so`, the plug-in binaries.
- A ‘workspace’ file and `dsp` folder for compiling the plug-in sources with Microsoft’s Visual Studio .NET on Microsoft Windows. The `make.bat` batch file can be used to build the Windows plug-in binaries, `FileTransport.dll` and `StringCodec.dll`.
- A sample configuration file, `config.xml`. This is an example of an IAF configuration that loads C plug-ins, configures them with plug-in properties, injects a specific EPL file into the event correlator, provides a simple event mapping, and configures the IAF for sending and receiving events to and from the event correlator.
- A simple EPL file, `simple.mon`. This defines a monitor that examines the incoming events and selectively emits some back out to the IAF.

- A text file, `simple-feed.evt`, with some test input messages that can be loaded by the File Transport plug-in, parsed by the String Codec plug-in, translated into Apama events by the Semantic Mapper, and then injected into the event correlator.
- A reference file, `simple-ref.evt`, which shows the expected output file generated when the adapter is run.

In order to run the example please follow the steps outlined in the `README.txt` file provided in the `samples\iaf_plugin\c\simple` folder.

Note: Plug-ins need to be placed in a location where they can be picked up by the event correlator:

- On Windows you either need to copy the `.dll` into the `bin` folder, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.
- On UNIX you either need to copy the `.so` into the `lib` directory, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

The Java example

The Java example is in the `samples\iaf_plugin\java\simple` directory, which contains the files:

- The complete source code of the `JFileTransport` transport layer plug-in and the `JStringCodec` codec plug-in, in the `src` directory.
  - The `JFileTransport` transport layer plug-in can read and write messages from and to a text file. This makes it a useful tool in testing the IAF and the event correlator with files of sample messages.
  - In normal operation, `JFileTransport` sends `String` objects on to the codec for decoding; however by setting the `upstreamNormalised` plug-in property it is possible to use the transport plug-in in a different mode in which it also performs the functionality that the codec usually performs (in this case by calling the `JStringCodec` class directly). In this mode the transport passes IAF normalized event messages on to the codec plug-in, demonstrating the use of the pass-through `JNullCodec` plug-in provided with the Apama distribution.
  - The `JStringCodec` codec plug-in can convert between normalized events and messages represented as strings containing a list of field names and values. The configuration properties for the plug-in allow customization of the syntactic characters used as field, name, and message separators (e.g. ",", "=", ";").
- An Apache Ant `build.xml` file is included, for compiling the `JFileAdapter.jar` binary that contains both plug-ins (and works on all platforms).
- A sample configuration file, `config.xml`. This is an example of an IAF configuration that loads Java transport and codec plug-ins, configures them with plug-in properties, provides an event mapping, and configures the IAF for sending and receiving events to and from the event correlator.
- This configuration file also includes several optional configuration options for logging, custom JVM options, logging of unmapped events/messages, and use of non-standard correlator event batching.
- A second configuration file, `config-no-codec.xml` that demonstrates how the standard `JNullCodec` plug-in can be used with a transport plug-in that incorporates codec functionality itself and produces normalized events directly.

- A simple EPL file, `simple.mon`. This is identical to the file included with the C sample, and defines a monitor that examines the incoming events and selectively emits some back out to the IAF.
- A text file, `simple-feed.evt`. This is identical to the file included with the C sample, and contains some test input messages that can be loaded by the File Transport plug-in, parsed by the String Codec plug-in, translated into Apama events by the Semantic Mapper, and then injected into the event correlator.
- A reference file, `simple-ref.evt`, which shows the expected output file generated when the adapter is run. Note that this file is (only trivially) different to the reference file for the C plug-ins.

In order to run the example please follow the steps outlined in the `README.txt` file provided in the `samples\iaf_plugin\java\simple` folder.

Chapter 3: C/C++ Transport Plug-in Development

The *transport layer* is the front-end of the IAF. The transport layer's purpose is to abstract away the differences between the programming interfaces exposed by different middleware message sources and sinks. It consists of one or more custom plug-in libraries that extract *downstream* messages from external message sources ready for delivery to the codec layer, and send Apama events already encoded by the codec layer *upstream* to the external message sink. See "The Integration Adapter Framework" on page 11 for a full introduction to transport plug-ins and the IAF's architecture.

An adapter should send events to the correlator only after its `start` function is called and before the `stop` function returns.

This topic includes the C/C++ Transport Plug-in Development Specification and additional information for developers of event transports using C/C++. "Transport Plug-in Development in Java" on page 89 provides information about developing transport plug-ins in Java.

To configure the build for a transport plug-in:

- On Linux, copying and customizing an Apama makefile from a sample application is the easiest method.
- On Windows, you might find it easiest to copy an Apama sample project. If you prefer to use a project you already have, be sure to add `$(APAMA_HOME)\include` as an include directory. To do this in Visual Studio, select your project and then select Project Properties > C/C++ > General > Additional Include Directories.

  Also, link against `iafcore$(APAMA_LIBRARY_VERSION).lib`. To do this in Visual Studio, select your project and then select Project Properties > Linker > Input > Additional Dependencies and add:

  ```
  iafcore$(APAMA_LIBRARY_VERSION).lib;apcommon$(APAMA_LIBRARY_VERSION).lib
  ```

  Finally, select Project Properties > Linker > General > Additional Library Directories, and add `$(APAMA_HOME)\lib`.

The C/C++ transport plug-in development specification

A C/C++ transport layer plug-in is implemented as a dynamic shared library. In order for the IAF to be able to load and use it, it must comply with Apama's Transport Plug-in Development Specification. This Specification describes the structure of a transport layer plug-in, and the C/C++ functions it needs to implement so that it can be used with the IAF. The Specification also provides a mechanism for startup and configuration parameters to be passed to the plug-in from the IAF's configuration file.

Property names and values used by transport plug-ins must be in UTF-8 format.

A transport layer plug-in implementation must include the C header file `EventTransport.h`. It also needs to include `EventCodec.h`, to allow the event transport to pass messages to codecs within the IAF codec layer.

C/C++ Transport Plug-in Development

Transport functions to implement

`EventTransport.h` provides the definition for a number of functions whose implementation needs to be provided by the event transport author.

These functions are as follows:

updateProperties

```
/**
  * Update the configuration of the transport.  The transport may assume
  * that stop(), flushUpstream() and flushDownstream() have all been called
  * before this function is invoked.  The recommended procedure for
  * updating properties is to first compare the new property set with the
  * existing stored properties -- if there are no changes, no action should
  * be taken.  Any pointer to the old property set becomes invalid as soon
  * as this function returns; any such pointers should therefore be
  * discarded in favour of the supplied new properties.
  *
  * @param transport The event transport instance
  * @param properties The new transport property set derived from the IAF
  * configuration file
  * @param timestampConfig Timestamp recording/logging settings
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*updateProperties)(
     struct AP_EventTransport* transport,
     AP_EventTransportProperties* properties,
     IAF_TimestampConfig* timestampConfig);
```

sendTransportEvent

```
/**
  * Called by an event encoder to send a message to the external transport.
  * Ownership of the message is transferred to the transport when this
  * function is called.  It is assumed that the encoder and transport share
  * the same definition of the content of the event, so that the transport
  * can effectively interpret the event and free any dynamically-allocated
  * memory.
  *
  * @param transport The event transport instance
  * @param event The event to be sent on the external transport
  * @param timeStamp Timestamps associated with this event
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*sendTransportEvent)(
     struct AP_EventTransport* transport,
     AP_TransportEvent event,
     AP_TimestampSet* timeStamp);
```

addEventDecoder

```
/**
  * Add a named event decoder to the set of decoders known to the
```

```
  * transport.  If the named decoder already exists, it should be
  * replaced.
  *
  * @param transport The event transport instance
  * @param name The name of the decoder to be added
  * @param decoder The decoder object itself
  */
  void (*addEventDecoder)(struct AP_EventTransport* transport,
     const AP_char8* name,
     struct AP_EventDecoder* decoder);
```

removeEventDecoder

```
/**
  * Remove a named event decoder from the set of decoders known to the
  * transport.  If the named decoder does not exist, the function should do
  * nothing.
  *
  * @param transport The event transport instance
  * @param name The decoder to be removed
  */
  void (*removeEventDecoder)(struct AP_EventTransport* transport,
     const AP_char8* name);
```

flushUpstream

```
/**
  * Flush any pending normalized events onto the external transport. The
  * transport may assume that the stop() function has been called before
  * this function, so in many cases no action will be required to complete
  * the flushing operation.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*flushUpstream)(
     struct AP_EventTransport* transport);
```

flushDownstream

```
/**
  * Flush any pending transport events into the decoder.  The transport may
  * assume that the stop() function has been called before this function,
  * so in many cases no action will be required to complete the flushing
  * operation. Under no circumstances should any events be sent to the
  * Correlator after flushDownstream() has returned.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*flushDownstream)(
     struct AP_EventTransport* transport);
```

start

```
/**
  * Establish a connection and start processing incoming data from the
  * external transport.
  *
  * An adapter should send events to the correlator only after its start()
  * method is called and before the stop() method returns. Therefore we
  * strongly recommend that a transport should not change to a state where
  * it is possible to receive events from any external transport until the
  * start() method has been called. In many cases, adapters will also need
  * to communicate with service monitors in the correlator to ensure that
  * the required monitors and event definitions are injected before they
```

```
  * begin to process messages from the external system. This is necessary in
  * order to avoid events from the adapter being lost if the correlator is
  * not yet ready to parse and process them.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*start)(struct AP_EventTransport* transport);
```

When the `start` function is invoked the event transport is effectively signaled to start accepting incoming messages and pass them onto a codec. Events should not be sent to the correlator until the `start` function is called.

It is up to the event transport to determine which codec to communicate with from the list of codecs made available to it through `addEventDecoder` and `removeEventDecoder`. Typically a configuration property would be used to specify the codec to be used. If a handle to the desired codec had been stored in a variable called `decoder` (of type `AP_EventDecoder*`) when `addEventDecoder` was called, an event could be passed on to the codec using:

```
decoder->functions->sendTransportEvent(decoder, event);
```

This codec function is described in "C/C++ Codec Plug-in Development" on page 62.

stop

```
/**
  * Stop processing incoming data from the external transport, typically
  * by pausing or closing down connections.
  *
  * Adapter authors must ensure that no events are sent to the Correlator
  * after stop() has returned (the only exception being rare cases where the
  * transport sends buffered events in the Correlator in the
  * flushDownstream() method, which is called by the IAF after stop()).
  * If necessary any messages that are unavoidably received from the
  * transport after stop() has returned should be blocked, queued or simply
  * dropped.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*stop)(struct AP_EventTransport* transport);
```

Events should not be sent to the correlator after the `stop` function has returned. The `stop` method must wait for any other threads sending events to complete before the `stop` method returns.

getLastError

```
/**
  * getLastError
  *
  * Return the transport's stored error message, if any.  The message
  * string is owned by the transport so should not be modified or freed by
  * the caller.
  *
  * @param transport The event transport instance
  * @return The last error message generated by the transport
  */
  const AP_char8* (*getLastError)(struct AP_EventTransport* transport);
getStatus/**
  * getStatus
  *
  * Fill in the supplied AP_EventTransportStatus structure with up-to-date
  * status information for the transport.  Note that any data pointed to by
  * the returned structure (such as strings) remains owned by the
  * transport.  The caller must copy this data if it wishes to modify it.
```

```
  *
  * @param codec The event transport instance
  * @param status The status structure to be filled in
  */
  void (*getStatus)(struct AP_EventTransport* transport,
     AP_EventTransportStatus* status);
```

The `AP_EventTransportStatus` structure contains four fields. The first field is a free-form text string that the transport can use to report any custom status information it might have. The `iaf_watch` tool will display the contents of this string. Note that the length of the status string is limited, currently to 1024 characters. Longer strings will be silently truncated. The next two fields report the total number of events received and sent by the transport. The last field, a pointer to an `AP_NormalisedEvent`, can contain custom information such as the state of the adapter.

The C/C++ transport plug-in development specification

Defining the transport function table

The `EventTransport.h` header file provides a definition for an `AP_EventTransport_Functions` structure. This defines a function table whose elements must be set to point to the implementations of the above functions. Its definition is as follows:

```
/**
 * AP_EventTransport_Functions
 *
 * Table of client visible functions exported by a transport library
 * instance.  These functions declare the only operations that may be
 * performed by users of a transport.
 *
 * Note that all of these functions take an initial AP_EventTransport*
 * argument; this is analogous to the (hidden) 'this' pointer passed to
 * a C++ object when a member function is invoked on it.
 */
struct AP_EventTransport_Functions {
/**
  * updateProperties
  *
  * Update the configuration of the transport.  The transport may assume
  * that stop(), flushUpstream() and flushDownstream() have all been called
  * before this function is invoked.  The recommended procedure for
  * updating properties is to first compare the new property set with the
  * existing stored properties -- if there are no changes, no action should
  * be taken.  Any pointer to the old property set becomes invalid as soon
  * as this function returns; any such pointers should therefore be
  * discarded in favour of the supplied new properties.
  *
  * @param transport The event transport instance
  * @param properties The new transport property set derived from the IAF
  * configuration file
  * @param timestampConfig Timestamp recording/logging settings
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*updateProperties)(
     struct AP_EventTransport* transport,
     AP_EventTransportProperties* properties,
     IAF_TimestampConfig* timestampConfig);
/**
  * sendTransportEvent
  *
  * Called by an event encoder to send a message to the external transport.
  * Ownership of the message is transferred to the transport when this
  * function is called.  It is assumed that the encoder and transport share
```

```
  * the same definition of the content of the event, so that the transport
  * can effectively interpret the event and free any dynamically-allocated
  * memory.
  *
  * @param transport The event transport instance
  * @param event The event to be sent on the external transport
  * @param timeStamp Timestamps associated with this event
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*sendTransportEvent)(
     struct AP_EventTransport* transport,
     AP_TransportEvent event,
     AP_TimestampSet* timeStamp);
/**
  * addEventDecoder
  *
  * Add a named event decoder to the set of decoders known to the
  * transport.  If the named decoder already exists, it should be
  * replaced.
  *
  * @param transport The event transport instance
  * @param name The name of the decoder to be added
  * @param decoder The decoder object itself
  */
  void (*addEventDecoder)(struct AP_EventTransport* transport,
     const AP_char8* name, struct AP_EventDecoder* decoder);
/**
  * removeEventDecoder
  *
  * Remove a named event decoder from the set of decoders known to the
  * transport.  If the named decoder does not exist, the function should do
  * nothing.
  *
  * @param transport The event transport instance
  * @param name The decoder to be removed
  */
  void (*removeEventDecoder)(struct AP_EventTransport* transport,
     const AP_char8* name);
/**
  * flushUpstream
  *
  * Flush any pending normalized events onto the external transport. The
  * transport may assume that the stop() function has been called before
  * this function, so in many cases no action will be required to complete
  * the flushing operation.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*flushUpstream)(
     struct AP_EventTransport* transport);
/**
  * flushDownstream
  *
  * Flush any pending transport events into the decoder.  The transport may
  * assume that the stop() function has been called before this function,
  * so in many cases no action will be required to complete the flushing
  * operation. Under no circumstances should any events be sent to the
  * Correlator after flushDownstream() has returned.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong
  */
  AP_EventTransportError (*flushDownstream)(
```

```
     struct AP_EventTransport* transport);
/**
  * start
  *
  * Establish a connection and start processing incoming data from the
  * external transport.
  *
  * An adapter should send events to the correlator only after its start()
  * method is called and before the stop() method returns. Therefore we
  * strongly recommend that a transport should not change to a state where
  * it is possible to receive events from any external transport until the
  * start() method has been called. In many cases, adapters will also need
  * to communicate with service monitors in the correlator to ensure that
  * the required monitors and event definitions are injected before they
  * begin to process messages from the external system. This is necessary in
  * order to avoid events from the adapter being lost if the correlator is
  * not yet ready to parse and process them.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*start)(struct AP_EventTransport* transport);
  /**
  * stop
  *
  * Stop processing incoming data from the external transport, typically
  * by pausing or closing down connections.
  *
  * Adapter authors must ensure that no events are sent to the Correlator
  * after stop() has returned (the only exception being rare cases where the
  * transport sends buffered events in the Correlator in the
  * flushDownstream() method, which is called by the IAF after stop()).
  * If necessary any messages that are unavoidably received from the
  * transport after stop() has returned should be blocked, queued or simply
  * dropped.
  *
  * @param transport The event transport instance
  * @return Event transport error code.  If this is not
  * AP_EventTransport_OK, the getLastError() function should be called to
  * get a more detailed description of what went wrong.
  */
  AP_EventTransportError (*stop)(struct AP_EventTransport* transport);
/**
  * getLastError
  *
  * Return the transport's stored error message, if any.  The message
  * string is owned by the transport so should not be modified or freed by
  * the caller.
  *
  * @param transport The event transport instance
  * @return The last error message generated by the transport
  */
  const AP_char8* (*getLastError)(struct AP_EventTransport* transport);
/**
  * getStatus
  *
  * Fill in the supplied AP_EventTransportStatus structure with up-to-date
  * status information for the transport.  Note that any data pointed to by
  * the returned structure (such as strings) remains owned by the
  * transport.  The caller must copy this data if it wishes to modify it.
  *
  * @param codec The event transport instance
  * @param status The status structure to be filled in
  */
  void (*getStatus)(struct AP_EventTransport* transport,
     AP_EventTransportStatus* status);
};
```

Please note that the order of the function pointers within the function table is critical to the reliable operation of the IAF. However, the order that the function definitions appear within the plug-in source code, and indeed the names of the functions, are not important. Apama recommends that the functions be declared `static`, so that they are not globally visible and can only be accessed via the function table.

It is therefore not obligatory to implement the functions documented above with the same names as per the definitions, as long as the mapping is performed correctly in an instantiation of `AP_EventTransport_Functions`. A definition in an event transport implementation would look as follows:

```
/**
 * Function table for the AP_EventTransport interface. The address of this
 * structure must be placed in the 'functions' field of the AP_EventTransport
 * object.
 */
static struct AP_EventTransport_Functions EventTransport_Functions
  = {
  updateProperties,
  sendTransportEvent,
  addEventDecoder,
  removeEventDecoder,
  flushUpstream,
  flushDownstream,
  start,
  stop,
  getLastError,
  getStatus
};
```

The function table created above needs to be placed in an `AP_EventTransport` object. The definition of this structure is as follows:

```
/**
 * External (client-visible) interface to an IAF TIL plugin library. The
 * AP_EventTransport struct contains a table of function pointers, declared in
 * the AP_EventTransport_Functions struct above. The implementation of these
 * functions is private. Users of the transport libary should invoke
 * functions on it as in the following example (transport is of type
 * AP_EventTransport*):
 *
 * i = transport->functions->sendTransportEvent(mapper, event, timestamp);
 */
struct AP_EventTransport {
  void* reserved;
  struct AP_EventTransport_Functions* functions;
};
```

and one such object needs to be created for every plug-in within its constructor function. Its first element, `reserved`, is a placeholder for any private data that the transport layer requires.

The C/C++ transport plug-in development specification

The transport constructor, destructor and info functions

Every event transport needs to implement a constructor function, a destructor function and an ‘info’ function. These methods are called by the IAF to (respectively) instantiate the event transport, to clean it up during unloading, and to provide information about the plug-in’s capabilities.

`EventTransport.h` provides the following definition for a pointer to the constructor function:

`AP_EventTransportCtorPtr`

```
/**
 * Pointer to the constructor function for the transport library.  Each TIL
 * plugin library must export a function with this signature, named using the
 * AP_EVENTTRANSPORT_CTOR_FUNCTION_NAME macro.
 *
 * Constructs a new instance of the event transport.  This function will be
 * called by the adapter main program when the adapter starts up.  The
 * transport should be created in a 'stopped' state, such that it is not
 * actively processing data from the external transport.  A call to the
 * start() function is required to begin processing.
 *
 * Note that the input parameters (name, properties, timestampConfig) are
 * owned by the caller.  The IAF framework guarantees that propertes and
 * timestampConfig will remain valid until after a subsequent call to
 * updateProperties(), so it is safe to hold a pointer to this structure.
 * You should, however, copy the name string if you wish to keep it.  The
 * contents of the output parameter errMsg belongs to the transport.
 *
 * @param name The name of this transport instance
 * @param properties Transport property set derived from the IAF config file
 * @param err 'Out' parameter for error code if constructor fails
 * @param errMsg 'Out' parameter for error message if constructor fails
 * @param timestampConfig Timestamp recording/logging settings
 * @return Pointer to a new transport instance or NULL if the constructor
 * fails for some reason.  In the case of failure, err and errMsg should be
 * filled in with an appropriate error code and message describing the
 * failure.
 */
typedef AP_EVENTTRANSPORT_API AP_EventTransport* (
     AP_EVENTTRANSPORT_CALL* AP_EventTransportCtorPtr)(AP_char8* name,
       AP_EventTransportProperties* properties, AP_EventTransportError* err,
       AP_char8** errMsg, IAF_TimestampConfig* timestampConfig);
```

Typically part of the work of this constructor would be a call to `updateProperties`, in order to set up the initial configuration of the plug-in.

The destructor function's related definition is as follows:

`AP_EventTransportDtorPtr`

```
/**
 * Pointer to the destructor function for the transport library.  Each TIL
 * plugin library must export a function with this signature, named using
 * the AP_EVENTTRANSPORT_DTOR_FUNCTION_NAME macro.
 *
 * Destroys an instance of the event transport that was previously created
 * by AP_EventTransport_ctor.  The transport may assume that it has been
 * stopped and flushed before the destructor is called.
 *
 * @param transport The event transport object to be destroyed
 */
typedef AP_EVENTTRANSPORT_API void (
     AP_EVENTTRANSPORT_CALL* AP_EventTransportDtorPtr)(
       AP_EventTransport* obj);
```

while the info function is defined as:

`AP_EventTransportInfoPtr`

```
/**
 * Pointer to the info function for the tranport library.  Each transport
 * plugin library must export a function with this signature, named using
 * the AP_EVENTTRANSPORT_INFO_FUNCTION_NAME macro.
 *
 * Returns basic information about the transports implemented by this
 * shared library, along with the transport API version it was built
 * against.
 *
 * @param version 'Out' parameter for version number of transport library.
 * This should be the value of AP_EVENTTRANSPORT_VERSION that was defined
 * in EventTransport.h when the transport was built.  The IAF will check
```

```
 * the version number to ensure that it can support all the capabilities
 * offered by the transport
 * @param capabilities 'Out' parameter for the capabilities of the
 * transports provided by this library.  Currently no such capabilities
 * are defined, so the value of this field will be ignored by the IAF.
 * However, all bits ofthe capabilities value are reserved for future use,
 * so current transports should ensure that they set this value to zero.
 */
typedef AP_EVENTTRANSPORT_API void (
     AP_EVENTTRANSPORT_CALL* AP_EventTransportInfoPtr)(AP_uint32* version,
       AP_uint32* capabilities);
```

The IAF will search for these functions by the names `AP_EventTransport_ctor`,`AP_EventTransport_dtor` and `AP_EventTransport_info` when the library is loaded, so you must use these exact names when implementing them in a transport layer plug-in.

The C/C++ transport plug-in development specification

Other transport definitions

`EventTransport.h` also provides some additional definitions that the event transport author needs to be aware of.

First of these is the set of error codes that can be returned by the transport's functions:

```
/**
 * AP_EventTransportError
 *
 * Error codes that can be returned by transport library functions.  The
 * enumeration values follow the normal UNIX convention of zero == OK and
 * non-zero == error.
 */
typedef enum {
  AP_EventTransport_OK = 0,           /* Everything is fine */
  AP_EventTransport_InternalError,    /* Some unspecified internal error occurred */
  AP_EventTransport_TransportFailure, /* Trouble reading/writing the external transport */
  AP_EventTransport_DecodingFailure,  /* Trouble sending transport event to decoder */
  AP_EventTransport_BadProperties,    /* Transport was passed an invalid property set */
  AP_EventTransport_CantStart         /* Transport could not start correctly */
} AP_EventTransportError;
```

Next is a definition for a configuration property. This corresponds to the properties that can be passed in as initialization or re-configuration parameters from the configuration file of the IAF.

```
/**
 * AP_EventTransportProperty
 *
 * A single transport property, corresponding to a <property> element
 * in the adapter configuration file.
 */
typedef struct {
  AP_char8* name;
  AP_char8* value;
} AP_EventTransportProperty;
```

Properties are passed to the event transport within an `AP_EventTransportProperties` structure:

```
/**
 * AP_EventTransportProperties
 *
 * Properties for the transport, extracted from the <transport> element
 * in the adapter configuration file.
 */
typedef struct {
  AP_char8* name;
  AP_EventTransportProperty** properties;
```

```
} AP_EventTransportProperties;
```

Finally, the status of a transport is reported in an `AP_EventTransportStatus` structure:

```
/**
 * AP_EventTransportStatus
 *
 * Transport status information structure, filled in by the getStatus call.
 */
typedef struct {
  AP_char8* status;
  AP_uint64 totalReceived;
  AP_uint64 totalSent;
  AP_NormalisedEvent* statusDictionary;
} AP_EventTransportStatus;
```

The C/C++ transport plug-in development specification

Transport utilities

The header files `AP_EventParser.h` and `AP_EventWriter.h` provide definitions for the Event Parser and Event Writer utilities. These utilities allow parsing and writing of the string form of reference types that are used by any `<map type="reference">` elements in the adapters configuration file. From the header files:

AP_EventParser

```
/**
 * Event Parser - a utility to parse events from strings
 *
 * An EventParser struct is created from a C string (NULL terminated).  The
 * type of event can then be interogated (the type member), though both
 * integers and floats are labelled as floats.  The number of elements is
 * the number of fields in an event, entries in a sequence, or key,value
 * pairs in a dictionary.
 *
 * Individual fields can then be retrieved, either as a basic type, or as a
 * reference type - basic types retrieved as reference types have a single
 * element which must be retrieved using the relevant method.  getRefValue
 * will return NULL if there are no more elements, and the basic type
 * methods will return 0/ false/ empty string.
 *
 * Dictionary entries must be obtained via the getDictPair function, which
 * sets two pointers to the key and value as reference types.
 *
 * No type checking is performed on the get methods; the behaviour when
 * there is a type mismatch is undefined.
 *
 * There are two variants of the get methods; the Next set maintain a
 * counterwhile those with an idx argument retrieve a specific element,
 * which does not affect the next counter. The next counter cannot be reset.
 *
 * Note that the implementation is lazy - it only parses stuff if requested
 * to. (though getNumberElements will parse all of that entity). Parsing is
 * cached, so calling getNumberElements repeatedly is cheap, and
 * getRefValue/ getString/ getDictPair will return the same objects if
 * called repeatedly with the same index.
 *
 * Where methods return an pointer (getString and getRefValue), that
 * pointer is valid until the top-level AP_EventParser is destroyed by
 * AP_EventParser_dtor (no need to destroy any of its sub- objects).  A
 * well-formed program shouldhave a matching number of AP_EventParser_ctor
 * and AP_EventParser_dtor calls, and only use string or refValue pointers
 * before AP_EventParser_dtor is called.
 *
 */AP_EventWriter
```

```
/**
 * Event Writer - a utility to create string forms of events.
 *
 * A utility class that can build string forms of events, sequences and
 * dictionaries.
 * An AP_EventWriter object holds a number of fields, each of which may be a
 * basic type (string, int, float, boolean), or a reference type (event,
 * sequence, dictionary).  Once the structure has been built up by adding
 * values, the toString method can be called.  This returns a string that
 * the caller is responsible for freeing.
 * Calling the destructor on an object will destroy all AP_EventWriter
 * objects it refers to in a recursive manner - thus, only the top-level
 * object should be destroyed, and it is not possible to share
 * AP_EventWriter objects between two different containing events.
 *
 */
```

The C/C++ transport plug-in development specification

Communication with the codec layer

If a transport layer plug-in is to be able to receive messages and then pass them on to the codec layer it must be able to communicate with appropriate *decoding codecs*. A *decoding* codec is one that can accept messages from the transport layer and parse them (decode them) into the normalized event format accepted by the Semantic Mapper.

When a codec is loaded into the IAF its details are passed to all transport layer plug-ins by calling their `addEventDecoder` function. This tells the transport layer plug-in the name of the decoding codec and provides a reference to its `AP_EventDecoder` structure.

The reference to `AP_EventDecoder` gives the transport layer plug-in access to the following functions:

sendTransportEvent

```
/**
  * Called by the event transport to decode an event and send it on to the
  * Semantic Mapper.  Ownership of the message is transferred to the
  * decoder when this function is called.  It is assumed that the encoder
  * and transport share the same definition of the content of the event, so
  * that the transport can effectively interpret the event and free any
  * dynamically-allocated memory.
  *
  * @param decoder The event decoder instance
  * @param event The event to be decoded
  * @param timeStamp Timestamps associated with this event
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the decoder should be called to get a more
  * detailed description of what went wrong.
  */
  AP_EventCodecError (*sendTransportEvent)(struct AP_EventDecoder* decoder,
     AP_TransportEvent event, AP_TimestampSet* timeStamp);
```

getLastError

```
/**
  * getLastError
  *
  * Return the decoder's stored error message, if any.  The message string
  * is owned by the decoder so should not be modified or freed by the
  * caller.
  *
  * @param decoder The event decoder instance
  * @return The last error message generated by the decoder
  */
  const AP_char8* (*getLastError)(struct AP_EventDecoder* decoder);
```

Assuming the reference to the `AP_EventDecoder` structure has been stored in a variable called `decoder`, the functions can be called as follows:

```
errorCode = decoder->functions->sendTransportEvent(decoder, event);
errorMessage = decoder->functions->getLastError(decoder);
```

The C/C++ transport plug-in development specification

Transport Example

As part of the IAF distribution Apama includes the `FileTransport` transport layer plug-in, implemented in the `samples\iaf_plugin\c\simple\FileTransport.c` source file.

The `FileTransport` plug-in can read and write messages from and to a text file, and it is recommended that developers examine this sample to see what a typical transport plug-in implementation looks like.

See "IAF samples" on page 45 for more information about this sample. "The File Transport plug-in" on page 140 describes how the `FileTransport` plug-in can be used in practice.

C/C++ Transport Plug-in Development

Getting started with transport layer plug-in development

In order to facilitate quick development of a transport layer plug-in your distribution includes a transport plug-in skeleton.

This file, called `skeleton-transport.c`, implements a complete transport layer plug-in that complies with the transport layer Plug-in Development Specification but where all the custom message source specific functionality is missing. The file is located in the `samples\iaf_plugin\c\skeleton` directory of your installation.

The skeleton starts a background thread to do the actual message reading. This is the only approach suitable, unless the external transport is able to call back into the transport layer plug-in.

In order to turn the skeleton into a fully operational message source specific transport layer plug-in, the plug-in author needs to fill in the gaps within the `updateProperties`, `sendTransportEvent`, `addEventDecoder`, `removeEventDecoder`, `flushUpstream`, `flushDownstream`, `start`, `stop`, `getLastError` and `getStatus` functions. These must implement their specified functionality in the context of the custom message source. The constructor, destructor and info functions are also likely to require adaptation.

The skeleton defines a structure, called `EventTransport_Internals`, to store all its private data, and this structure is placed within the `reserved` field of the `AP_EventTransport` object created within the constructor method. It is likely that this structure will need to be modified to contain additional data that the adapter might require.

Any custom initialization and communications code, such as code to connect and register with a message bus, or opening a database, etc., can either be placed in the constructor or in the primary worker thread's `run` method. Alternatively, one might need to place such code in the `updateProperties` method, which is called by the IAF at initialization time as well as whenever it is requested to reload the configuration file and thus resend the plug-in's properties.

The distribution also includes a `Makefile` (for use with GNU Make on UNIX) as well as a workspace file and `dsp` folder, for use with Microsoft's Visual Studio .NET on Microsoft Windows, for this

skeleton, which can be adapted to compile your transport layer plug-in and link it against any custom libraries required.

Once a plug-in is built, it needs to be placed in a location where it can be picked up by the IAF.

This means that on Windows you either need to copy the `.dll` into the `bin` folder, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.

On UNIX you either need to copy the `.so` into the `lib` directory, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

C/C++ Transport Plug-in Development

Chapter 4: C/C++ Codec Plug-in Development

The *codec layer* is a layer of abstraction between the transport layer and the IAF's Semantic Mapper. It consists of one or more plug-in libraries that perform message *encoding* and/or *decoding*. Decoders translate *downstream* messages retrieved by the transport layer into the standard 'normalized event' format on which the Semantic Mapper's rules run. Encoders work in the opposite direction, encoding *upstream* normalized events into an appropriate format for transport layer plug-ins to send on. See "The Integration Adapter Framework" on page 11 for a full introduction to codec plug-ins and the IAF's architecture.

This topic includes the C/C++ Codec Plug-in Development Specification and additional information for developers of C/C++ event codecs. "Java Codec Plug-in Development" on page 98 provides analogous information about developing codec plug-ins in Java.

Before developing a new codec plug-in, it is worth considering whether one of the standard Apama IAF plug-ins could be used instead. "Standard Plug-ins" on page 137 provides more information on the standard IAF codec plug-ins: `StringCodec` and `NullCodec`. The `StringCodec` plug-in codes normalized events as formatted text strings. The `NullCodec` plug-in is useful in situations where it does not make sense to decouple the codec and transport layers, and allows transport plug-ins to communicate with the Semantic Mapper directly using normalized events.

To configure the build for a codec plug-in:

- On Linux, copying and customizing an Apama makefile from a sample application is the easiest method.
- On Windows, you might find it easiest to copy an Apama sample project. If you prefer to use a project you already have, be sure to add `$(APAMA_HOME)\include` as an include directory. To do this in Visual Studio, select your project and then select Project Properties > C/C++ > General > Additional Include Directories.

  Also, link against `iafcore$(APAMA_LIBRARY_VERSION).lib`. To do this in Visual Studio, select your project and then select Project Properties > Linker > Input > Additional Dependencies and add:

  ```
  iafcore$(APAMA_LIBRARY_VERSION).lib;apcommon$(APAMA_LIBRARY_VERSION).lib
  ```

  Finally, select Project Properties > Linker > General > Additional Library Directories, and add `$(APAMA_HOME)\lib`.

The C/C++ codec Plug-in Development Specification

A codec plug-in needs to be structured as a dynamic shared library. In order for the IAF to be able to load and use it, it must comply with Apama's Codec Plug-in Development Specification. This describes the overall format of a codec plug-in and the C/C++ functions it needs to implement so that

its functionality is accessible by the IAF. The Specification also provides a mechanism for startup and configuration parameters to be passed to the plug-in from the IAF's configuration file.

Property names and values used by codec plug-ins must be in UTF-8 format.

A codec plug-in implementation must include the C header file `EventCodec.h`. This file can be found in the `include` directory. As a codec also needs to communicate both with a transport layer plug-in (or event transport) and with the Semantic Mapper, `EventTransport.h` and `SemanticMapper.h` also need to be included.

C/C++ Codec Plug-in Development

Codec functions to implement

`EventCodec.h` provides the definition for a number of functions whose implementation needs to be provided by the event transport author.

However, in contrast to the Transport Layer Plug-in Development Specification, the set of functions that need to be implemented varies depending on whether the codec is to implement only a message decoder, only a message encoder, or a bidirectional encoder/decoder.

In all cases implementations need to be provided for the following functions:

updateProperties

```
/**
  * Update the configuration of the codec.  The codec may assume that
  * flushUpstream() and flushDownstream() have been called before this
  * function is invoked.  The recommended procedure for updating properties
  * is to first compare the new property set with the existing stored
  * properties -- if there are no changes, no action should be taken.  Any
  * pointer to the old property set becomes invalid as soon as this
  * function returns; any such pointers should therefore be discarded in
  * favour of the supplied new properties.
  *
  * @param codec The event codec instance
  * @param properties The new codec property set derived from the IAF
  * configuration file
  * @param timestampConfig Timestamp recording/logging settings
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the codec should be called to get a more
  * detailed description of what went wrong.
  */
AP_EventCodecError (*updateProperties)(struct AP_EventCodec* codec,
    AP_EventCodecProperties* properties,
    IAF_TimestampConfig* timestampConfig);
```

It is recommended that `updateProperties` is invoked by the codec constructor.

getLastError

```
/**
  * Return the codec's stored error message, if any.  The message string is
  * owned by the codec so should not be modified or freed by the caller.
  *
  * @param codec The event codec instance
  * @return The last error message generated by the codec
  */
const AP_char8* (*getLastError)(struct AP_EventCodec* codec);
```

getStatus

```
/**
```

```
  * Fill in the supplied AP_EventCodecStatus structure with up-to-date
  * status information for the codec.  Note that any data pointed to by the
  * returned structure (such as strings) remains owned by the codec.  The
  * caller must copy this data if it wishes to modify it.
  *
  * @param codec The event codec instance
  * @param status The status structure to be filled in
  */
void (*getStatus)(struct AP_EventCodec* codec, AP_EventCodecStatus* status);
```

The `AP_EventCodecStatus` structure contains four fields. The first field is a free-form text string that the transport can use to report any custom status information it might have. The `iaf_watch` tool will display the contents of this string. Note that the length of the status string is limited, currently to 1024 characters. Longer strings will be silently truncated. The next two fields report the total number of events encoded and decoded by the codec. The last field, a pointer to an `AP_NormalisedEvent`, can contain custom information such as the state of the adapter.

The C/C++ codec Plug-in Development Specification

Codec encoder functions

If the codec is to implement an encoder, implementations need to be provided for the following functions:

sendNormalisedEvent

```
/**
  * Called by the Semantic Mapper to encode an event and send it on to the
  * event transport.  Ownership of the message is transferred to the
  * encoder when this function is called.
  *
  * @param encoder The event encoder instance
  * @param event The event to be encoded
  * @param timeStamp Timestamps associated with this event
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the encoder should be called to get a more
  * detailed description of what went wrong.
  */
AP_EventCodecError (*sendNormalisedEvent)(struct AP_EventEncoder* encoder,
    AP_NormalisedEvent* event, AP_TimestampSet* timeStamp);
```

flushUpstream

```
/**
  * Flush any pending normalized events into the attached event transport.
  * If event processing in the encoder is synchronous (as it usually will
  * be) this function need not do anything except return AP_EventCodec_OK.
  *
  * @param encoder The event encoder instance
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the encoder should be called to get a more
  * detailed description of what went wrong.
  */
AP_EventCodecError (*flushUpstream)(struct AP_EventEncoder* encoder);
```

getLastError

```
/**
  * Return the encoder's stored error message, if any.  The message string
  * is owned by the encoder so should not be modified or freed by the
  * caller.
  *
  * @param encoder The event encoder instance
  * @return The last error message generated by the encoder
  */
const AP_char8* (*getLastError)(struct AP_EventEncoder* encoder);
```

addEventTransport

```
/**
  * Add a named event transport to the set of transports known to the
  * encoder.
  *
  * If the named transport already exists, it should be replaced.
  *
  * @param encoder The event encoder instance
  * @param name The name of the transport to be added.
  * @param transport The transport object itself
  */
void (*addEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name, struct AP_EventTransport* transport);
```

removeEventTransport

```
/**
  * Remove a named event transport from the set of transports known to the
  * encoder.
  *
  * If the named transport does not exist, the function should do nothing.
  *
  * @param encoder The event encoder instance
  * @param name The name of the transport to be removed
  */
void (*removeEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name);
```

Codec functions to implement

Codec decoder functions

If the codec is to provide a decoder, implementations need to be provided for the following functions:

sendTransportEvent

```
/**
  * Called by the event transport to decode an event and send it on to the
  * Semantic Mapper.  Ownership of the message is transferred to the
  * decoder when this function is called.  It is assumed that the encoder
  * and transport share the same definition of the content of the event, so
  * that the transport can effectively interpret the event and free any
  * dynamically-allocated memory.
  *
  * @param decoder The event decoder instance
  * @param event The event to be decoded
  * @param timeStamp Timestamps associated with this event
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the decoder should be called to get a more
  * detailed description of what went wrong.
  */
AP_EventCodecError (*sendTransportEvent)(struct AP_EventDecoder* decoder,
    AP_TransportEvent event, AP_TimestampSet* timeStamp);
```

setSemanticMapper

```
/**
 * Set the Semantic Mapper object to be used by the decoder.  Currently
  * only a single Semantic Mapper is supported in each adapter instance.
  *
  * @param decoder The event decoder instance
  * @param mapper The Semantic Mapper to be used
  */
void (*setSemanticMapper)(struct AP_EventDecoder* decoder,
    AP_SemanticMapper* mapper);
```

flushDownstream

```
/**
  * Flush any pending transport events into the attached Semantic Mapper.
```

```
  * If event processing in the decoder is synchronous (as it usually will
  * be) this function need not do anything except return AP_EventCodec_OK.
  *
  * @param decoder The event decoder instance
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the decoder should be called to get a more
  * detailed description of what went wrong.
  */
AP_EventCodecError (*flushDownstream)(struct AP_EventDecoder* decoder);

getLastError
/**
  * Return the decoder's stored error message, if any.  The message string
  * is owned by the decoder so should not be modified or freed by the
  * caller.
  *
  * @param decoder The event decoder instance
  * @return The last error message generated by the decoder
  */
const AP_char8* (*getLastError)(struct AP_EventDecoder* decoder);
```

Codec functions to implement

Defining the codec function tables

In a transport layer plug-in, the plug-in author needs to provide a function table that tells the IAF which functions to call to invoke specific functionality.

The Codec Development Specification follows this model but depending on whether the codec being developed is an encoder, a decoder or an encoder/decoder, up to three function tables may need to be defined.

Please note that the order of the function pointers within each function table is critical to the reliable operation of the IAF. However, the order that the function definitions appear within the plug-in source code, and indeed the names of the functions, are not important. Apama recommends that the functions be declared `static`, so that they are not globally visible and can only be accessed via the function table.

The C/C++ codec Plug-in Development Specification

The codec function table

Every codec needs to define a generic codec function table. The header file provides a definition for this as an `AP_EventCodec_Functions` structure. Its definition is as follows:

```
/**
 * AP_EventCodec_Functions
 *
 * Table of client visible functions exported by a codec library instance.
 * These functions declare the only operations that may be performed by users
 * of a codec (but note that separate function tables exist for encoding-
 * and decoding-specific functions).
 *
 * Note that all of these functions take an initial AP_EventCodec*
 * argument; this is analogous to the (hidden) 'this' pointer passed to a
 * C++ object when a member function is invoked on it.
 */
struct AP_EventCodec_Functions {
  /**
  * updateProperties
  *
  * Update the configuration of the codec.  The codec may assume that
```

```
  * flushUpstream() and flushDownstream() have been called before this
  * function is invoked.  The recommended procedure for updating properties
  * is to first compare the new property set with the existing stored
  * properties -- if there are no changes, no action should be taken.  Any
  * pointer to the old property set becomes invalid as soon as this
  * function returns; any such pointers should therefore be discarded in
  * favour of the supplied new properties.
  *
  * @param codec The event codec instance
  * @param properties The new codec property set derived from the IAF
  * configuration file
  * @param timestampConfig Timestamp recording/logging settings
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the codec should be called to get a more
  * detailed description of what went wrong.
  */
  AP_EventCodecError (*updateProperties)(struct AP_EventCodec* codec,
    AP_EventCodecProperties* properties,
    IAF_TimestampConfig* timestampConfig);
  /**
  * getLastError
  *
  * Return the codec's stored error message, if any.  The message string is
  * owned by the codec so should not be modified or freed by the caller.
  *
  * @param codec The event codec instance
  * @return The last error message generated by the codec
  */
  const AP_char8* (*getLastError)(struct AP_EventCodec* codec);
  /**
  * getStatus
  *
  * Fill in the supplied AP_EventCodecStatus structure with up-to-date
  * status information for the codec.  Note that any data pointed to by the
  * returned structure (such as strings) remains owned by the codec.  The
  * caller must copy this data if it wishes to modify it.
  *
  * @param codec The event codec instance
  * @param status The status structure to be filled in
  */
  void (*getStatus)(struct AP_EventCodec* codec,
     AP_EventCodecStatus* status);
};
```

where the library functions `updateProperties`,`getLastErrorCodec` and `getStatus` are being defined as being the implementations of the Codec Development Specification's `updateProperties`,`getLastError` and `getStatus` function definitions respectively.

Defining the codec function tables

The codec encoder function table

If the codec being implemented is to act as an encoder it needs to implement the encoder functions listed previously and map them in an encoder function table. This structure is defined in `EventCodec.h` as an `AP_EventEncoder_Functions` struct:

```
/**
 * AP_EventEncoder_Functions
 *
 * Table of client visible functions exported by the encoder part of a
 * codec library instance.  These functions declare the only operations
 * that may be performed by users of an encoder.
 *
 * Note that all of these functions take an initial AP_EventEncoder*
 * argument; this is analogous to the (hidden) 'this' pointer passed to
 * a C++ object when a member function is invoked on it.
 */
struct AP_EventEncoder_Functions {
```

```
  /**
  * sendNormalisedEvent
  *
  * Called by the Semantic Mapper to encode an event and send it on to the
  * event transport.  Ownership of the message is transferred to the
  * encoder when this function is called.
  *
  * @param encoder The event encoder instance
  * @param event The event to be encoded
  * @param timeStamp Timestamps associated with this event
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the encoder should be called to get a more
  * detailed description of what went wrong.
  */
  AP_EventCodecError (*sendNormalisedEvent)(struct AP_EventEncoder* encoder,
    AP_NormalisedEvent* event, AP_TimestampSet* timeStamp);
  /**
  * flushUpstream
  *
  * Flush any pending normalized events into the attached event transport.
  * If event processing in the encoder is synchronous (as it usually will
  * be) this function need not do anything except return AP_EventCodec_OK.
  *
  * @param encoder The event encoder instance
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the encoder should be called to get a more
  * detailed description of what went wrong.
  */
  AP_EventCodecError (*flushUpstream)(struct AP_EventEncoder* encoder);
  /**
  * getLastError
  *
  * Return the encoder's stored error message, if any.  The message string
  * is owned by the encoder so should not be modified or freed by the
  * caller.
  *
  * @param encoder The event encoder instance
  * @return The last error message generated by the encoder
  */
  const AP_char8* (*getLastError)(struct AP_EventEncoder* encoder);
  /**
  * addEventTransport
  *
  * Add a named event transport to the set of transports known to the
  * encoder.
  *
  * If the named transport already exists, it should be replaced.
  *
  * @param encoder The event encoder instance
  * @param name The name of the transport to be added.
  * @param transport The transport object itself
  */
  */
  void (*addEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name, struct AP_EventTransport* transport);
  /**
  * removeEventTransport
  *
  * Remove a named event transport from the set of transports known to the
  * encoder.
  *
  * If the named transport does not exist, the function should do nothing.
  *
  * @param encoder The event encoder instance
  * @param name The name of the transport to be removed
  */
  void (*removeEventTransport)(struct AP_EventEncoder* encoder,
    const AP_char8* name);
};
```

In the implementation of an encoding codec, this function table could be implemented as follows:

```
/**
 * Function table for the AP_EventEncoder interface. The address of this
 * structure will be placed in the 'functions' field of the embedded
 * AP_EventEncoder object.
 */
static struct AP_EventEncoder_Functions EventEncoder_Functions = {
  sendNormalisedEvent,
  flushUpstream,
  getLastErrorEncoder,
  addEventTransport,
  removeEventTransport
};
```

This time, the library functions `sendNormalisedEvent`, `flushUpstream`,`getLastErrorEncoder`, `addEventTransport` and `removeEventTransport` are being defined as the implementations of the Codec Development Specification's `sendNormalisedEvent`, `flushUpstream`,`getLastError`, `addEventTransport` and `removeEventTransport` function definitions respectively.

Defining the codec function tables

The codec decoder function table

If the codec being implemented is to act as a decoder it needs to implement the decoder functions listed previously and map them in a decoder function table. This structure is defined in `EventCodec.h` as an `AP_EventDecoder_Functions` structure:

```
/**
 * AP_EventDecoder_Functions
 *
 * Table of client visible functions exported by the decoder part of a
 * codec library instance. These functions declare the only operations that
 *  may be performed by users of a decoder.
 *
 * Note that all of these functions take an initial AP_EventDecoder*
 * argument; this is analogous to the (hidden) 'this' pointer passed to
 * a C++ object when a member function is invoked on it.
 */
struct AP_EventDecoder_Functions {
  /**
   * sendTransportEvent
   *
   * Called by the event transport to decode an event and send it on to the
   * Semantic Mapper.  Ownership of the message is transferred to the
   * decoder when this function is called.  It is assumed that the encoder
   * and transport share the same definition of the content of the event, so
   * that the transport can effectively interpret the event and free any
   * dynamically-allocated memory.
   *
   * @param decoder The event decoder instance
   * @param event The event to be decoded
   * @param timeStamp Timestamps associated with this event
   * @return Event codec error code.  If this is not AP_EventCodec_OK, the
   * getLastError() function of the decoder should be called to get a more
   * detailed description of what went wrong.
   */
  AP_EventCodecError (*sendTransportEvent)(struct AP_EventDecoder* decoder,
    AP_TransportEvent event, AP_TimestampSet* timeStamp);
  /**
   * setSemanticMapper
   *
   * Set the Semantic Mapper object to be used by the decoder.  Currently
   * only a single Semantic Mapper is supported in each adapter instance.
   *
   * @param decoder The event decoder instance
   * @param mapper The Semantic Mapper to be used
```

```
  *
  */
  void (*setSemanticMapper)(struct AP_EventDecoder* decoder,
    AP_SemanticMapper* mapper);
  /**
  * flushDownstream
  *
  * Flush any pending transport events into the attached Semantic Mapper.
  * If event processing in the decoder is synchronous (as it usually will
  * be) this function need not do anything except return AP_EventCodec_OK.
  *
  * @param decoder The event decoder instance
  * @return Event codec error code.  If this is not AP_EventCodec_OK, the
  * getLastError() function of the decoder should be called to get a more
  * detailed description of what went wrong.
  */
  AP_EventCodecError (*flushDownstream)(struct AP_EventDecoder* decoder);
  /**
  * getLastError
  *
  * Return the decoder's stored error message, if any. The message string
  * is owned by the decoder so should not be modified or freed by the
  * caller.
  *
  * @param decoder The event decoder instance
  * @return The last error message generated by the decoder
  */
  const AP_char8* (*getLastError)(struct AP_EventDecoder* decoder);
};
```

In the implementation of a decoding codec, this function table could be implemented as follows:

```
/**
 * EventDecoder_Functions
 *
 * Function table for the AP_EventDecoder interface. The address of this
 * structure will be placed in the 'functions' field of the embedded
 * AP_EventDecoder object.
 */
static struct AP_EventDecoder_Functions EventDecoder_Functions = {
  sendTransportEvent,
  setSemanticMapper,
  flushDownstream,
  getLastErrorDecoder
};
```

As before, this definition defines a number of library functions as the implementations of the function definitions specified in the Codec Development Specification.

Defining the codec function tables

Registering the codec function tables

The encoding and decoding function tables created above need to be placed in the relevant object, `AP_EventEncoder` and `AP_EventDecoder`. These, together with the generic function table, need to be placed in an `AP_EventCodec` object. The definitions of these structures are as follows:

```
/**
 * AP_EventEncoder
 *
 * External (client-visible) interface to the encoder part of a codec
 * plugin library.  The AP_EventEncoder struct contains a table of
 * function pointers, declared in the AP_EventEncoder_Functions struct
 * above.  The implementation of these functions is private.  Users of
 * the encoder should invokefunctions on it as in the following example
 * (encoder is of type AP_EventEncoder*):
 *
 * i = encoder->functions->sendNormalisedEvent(encoder, event, timestamps);
 */
```

```
struct AP_EventEncoder {
  /**
   * Pointer to private internal data.
   */
  void* reserved;
  /**
   * Function table of encoder operations.  See documentation for
   * AP_EventEncoder_Functions for details.
   */
  struct AP_EventEncoder_Functions* functions;
};
/**
 * AP_EventDecoder
 *
 * External (client-visible) interface to the decoder part of a codec
 * plugin library.  The AP_EventDecoder struct contains a table of
 * function pointers, declared in the AP_EventEDecoder_Functions struct
 * above. The implementation of these functions is private.  Users of the
 * decoder should invoke functions on it as in the following example (
 * decoder is of type AP_EventDecoder*):
 *
 * i = decoder->functions->sendTransportEvent(decoder, event, timestamps);
 */
struct AP_EventDecoder {
  /**
   * Pointer to private internal data.
   */
  void* reserved;
  /**
   * Function table of decoder operations.  See documentation for
   * AP_EventDecoder_Functions for details.
   */
  struct AP_EventDecoder_Functions* functions;
};
/**
 * AP_EventCodec
 *
 * External (client-visible) interface to an IAF codec plugin library. The
 * AP_EventCodec struct contains pointers to the encoder and decoder parts
 * of the codec (either of these may be NULL if the codec operates in one
 * direction only) and a table of function pointers, declared in the
 * AP_EventCodec_Functions struct above.  The implementation of these
 * functions is private.  Users of the codec should invoke functions on it
 * as in the following example (codec is of type AP_EventCodec*):
 *
 * i = codec->functions->updateProperties(codec, properties, timestampConfig);
 */
struct AP_EventCodec {
  /**
   * Pointer to private internal data.
   */
  void* reserved;
  /**
   * Function table of codec operations.  See documentation for
   * AP_EventCodec_Functions for details.
   */
  struct AP_EventCodec_Functions* functions;
  /**
   * Pointer to embedded encoder.
   */
  AP_EventEncoder* encoder;
  /**
   * Pointer to embedded decoder.
   */
  AP_EventDecoder* decoder;
};
```

The first element of each structure, `reserved`, is a placeholder for any private data that the encoder, decoder, or the generic codec code requires.

An `AP_EventCodec` object needs to be created for every plug-in within its constructor function. As indicated in the descriptive text, the encoder and decoder fields in it may be set to `NULL` if the codec does not implement the respective functionality, although clearly it is meaningless to have both set to `NULL`.

Defining the codec function tables

The codec constructor, destructor and info functions

Every event codec needs to implement a constructor function, a destructor function and an 'info' function. These methods are called by the IAF to (respectively) to instantiate the event codec, to clean it up during unloading, and to provide information about the plug-in's capabilities.

`EventCodec.h` provides the following definition for a pointer to the constructor function:

AP_EventCodecCtorPtr

```
/**
 * Pointer to the constructor function for the codec library.  Each codec
 * plugin library must export a function with this signature, named using
 * the AP_EVENTCODEC_CTOR_FUNCTION_NAME macro.
 *
 * Constructs a new instance of the event codec.  This function will be
 * called by the adapter main program when the adapter starts up.
 *
 * Note that the input parameters (name, properties, timestampConfig) are
 * owned by the caller.  The IAF framework guarantees that properties and
 * timestampConfig will remain valid until after a subsequent call to
 * updateProperties(), so it is safe to hold a pointer to this structure.
 * You should, however, copy the name string if you wish to keep it.  The
 * contents of the output parameter errMsg belongs to the codec.
 *
 * @param name The name of this codec instance, also found in properties
 * @param properties Codec property set derived from the IAF config file
 * @param err 'Out' parameter for error code if constructor fails
 * @param errMsg 'Out' parameter for error message if constructor fails
 * @param timestampConfig Timestamp recording/logging settings
 * @return Pointer to a new codec instance or NULL if the constructor fails
 * for some reason. In the case of failure, err and errMsg should be filled
 * in with an appropriate error code and message describing the failure.
 */
typedef AP_EVENTCODEC_API AP_EventCodec* (
    AP_EVENTCODEC_CALL* AP_EventCodecCtorPtr)(
      AP_char8* name, AP_EventCodecProperties* properties,
      AP_EventCodecError* err, AP_char8** errMsg,
      IAF_TimestampConfig* timestampConfig);
```

while the destructor function definition is as follows:

AP_EventCodecDtorPtr

```
/**
 * Pointer to the destructor function for the codec library.  Each codec
 * plugin library must export a function with this signature, named using
 * the AP_EVENTCODEC_DTOR_FUNCTION_NAME macro.
 *
 * Destroys an instance of the event codec that was previously created by
 * AP_EventCodec_ctor.  The codec may assume that it has been flushed
 * before the destructor is called.
 *
 * @param codec The event codec object to be destroyed
 */
typedef AP_EVENTCODEC_API void
    (AP_EVENTCODEC_CALL* AP_EventCodecDtorPtr)(AP_EventCodec* codec);
```

The IAF will search for these functions by the names `AP_EventCodec_ctor` and `AP_EventCodec_dtor` when the library is loaded, so you must use these exact names when implementing a codec plug-in.

In addition, every codec needs to implement an 'information' function. This is called by the IAF to obtain information as to the capabilities (encoder/decoder) of the codec. The definition is:

AP_EventCodecInfoPtr

```
/**
 * Pointer to the info function for the codec library.  Each codec plugin
 * library must export a function with this signature, named using the
 * AP_EVENTCODEC_INFO_FUNCTION_NAME macro.
 *
 * Returns basic information about the codecs implemented by this shared
 * library, specifically whether they are encoders, decoders or
 * bidirectional codecs.
 *
 * @param version 'Out' parameter for version number of codec library.  This
 * should be the value of AP_EVENTCODEC_VERSION that was defined in
 * EventCodec.h when the codec was built.  The IAF will check the version
 * number to ensure that it can support all the capabilities offered by the
 * codec.
 * @param capabilities 'Out' parameter for the capabilities of the codecs
 * provided by this library.  This should be the sum of the appropriate
 * AP_EVENTCODEC_CAP_* constants found in EventCodec.h.
 */
typedef AP_EVENTCODEC_API void
    (AP_EVENTCODEC_CALL* AP_EventCodecInfoPtr)(AP_uint32* version,
       AP_uint32* capabilities);
```

The IAF will search for and call this function by the name `AP_EventCodec_info`.

The C/C++ codec Plug-in Development Specification

Other codec definitions

`EventCodec.h` also provides some additional definitions that the codec author needs to be aware of.

First of these are the codec capability bits. These are returned by the 'information' function previously illustrated to define whether the codec can decode or encode messages.

```
#define AP_EVENTCODEC_CAP_ENCODER  0x0001
#define AP_EVENTCODEC_CAP_DECODER  0x0002
```

Next is the set of error codes that can be returned by the codec's functions:

```
/**
 * AP_EventCodecError
 *
 * Error codes that can be returned by codec library functions.  The
 * enumeration values follow the normal UNIX convention of zero == OK and
 * non-zero == error.
 */
typedef enum {
 AP_EventCodec_OK = 0,           /* Everything is fine */
 AP_EventCodec_InternalError,    /* Some unspecified internal error occurred */
 AP_EventCodec_EncodingFailure,  /* Couldn't encode an incoming normalized event */
 AP_EventCodec_DecodingFailure,  /* Couldn't decode an incoming customer event */
 AP_EventCodec_TransportFailure, /* Trouble sending encoded event to transport */
 AP_EventCodec_MappingFailure,   /* Trouble sending decoded event to Semantic Mapper */
 AP_EventCodec_BadProperties     /* Codec was passed an invalid property set */
} AP_EventCodecError;
```

Next is a definition for a configuration property. This corresponds to the properties that can be passed in as initialization or re-configuration parameters from the configuration file of the IAF.

```
/**
 * AP_EventCodecProperty
 *
 * A single codec property, corresponding to a <property> element in the
 * adapter configuration file.
 */
typedef struct {
  AP_char8* name;
  AP_char8* value;
} AP_EventCodecProperty;
```

Properties are passed to the event transport within an `AP_EventCodecProperties` structure:

```
/**
 * AP_EventCodecProperties
 *
 * Properties for the codec, extracted from the <codec> element in the
 * adapter configuration file.
 */
typedef struct {
  AP_char8* name;
  AP_EventCodecProperty** properties;
} AP_EventCodecProperties;
```

Finally, the status of a codec is reported in an `AP_EventCodecStatus` structure:

```
/**
 * AP_EventCodecStatus
 *
 * Codec status information structure, filled in by the getStatus call.
 */
typedef struct {
  AP_char8* status;
  AP_uint64 totalDecoded;
  AP_uint64 totalEncoded;
  AP_NormalisedEvent* statusDictionary;
} AP_EventCodecStatus;
```

You are advised to peruse `EventCodec.h` for the complete definitions. `EventTransport.h` and `SemanticMapper.h` are also relevant as they define the functions that a codec author can invoke within the transport layer and the Semantic Mapper, respectively.

The C/C++ codec Plug-in Development Specification

Codec utilities

The header files `AP_EventParser.h` and `AP_EventWriter.h` provide definitions for the Event Parser and Event Writer utilities. These utilities allow parsing and writing of the string form of reference types that are used by any `<map type="reference">` elements in the adapters configuration file. From the header files:

`AP_EventParser`

```
/**
 * Event Parser - a utility to parse events from strings
 *
 * An EventParser struct is created from a C string (NULL terminated).  The
 * type of event can then be interogated (the type member), though both
 * integers and floats are labelled as floats.  The number of elements is
 * the number of fields in an event, entries in a sequence, or key,value
 * pairs in a dictionary.
```

```
 *
 * Individual fields can then be retrieved, either as a basic type, or as a
 * reference type - basic types retrieved as reference types have a single
 * element which must be retrieved using the relevant method.  getRefValue
 * will return NULL if there are no more elements, and the basic type
 * methods will return 0/ false/ empty string.
 *
 * Dictionary entries must be obtained via the getDictPair function, which
 * sets two pointers to the key and value as reference types.
 *
 * No type checking is performed on the get methods; the behaviour when
 * there is a type mismatch is undefined.
 *
 * There are two variants of the get methods; the Next set maintain a
 * counter while those with an idx argument retrieve a specific element,
 * which does not affect the next counter. The next counter cannot be reset.
 *
 * Note that the implementation is lazy - it only parses stuff if requested
 * to. (though getNumberElements will parse all of that entity). Parsing is
 * cached, so calling getNumberElements repeatedly is cheap, and
 * getRefValue/ getString/ getDictPair will return the same objects if
 * called repeatedly with the same index.
 *
 * Where methods return an pointer (getString and getRefValue), that
 * pointer is valid until the top-level AP_EventParser is destroyed by
 * AP_EventParser_dtor (no need to destroy any of its sub- objects).  A
 * well-formed program should have a matching number of AP_EventParser_ctor
 * and AP_EventParser_dtor calls, and only use string or refValue pointers
 * before AP_EventParser_dtor is called.
 *
 */AP_EventWriter

/**
 * Event Writer - a utility to create string forms of events.
 *
 * A utility class that can build string forms of events, sequences and
 * dictionaries.
 * An AP_EventWriter object holds a number of fields, each of which may be
 * a basic type (string, int, float, boolean), or a reference type (event,
 * sequence, dictionary).  Once the structure has been built up by adding
 * values, the toString method can be called.  This returns a string that
 * the caller is responsible for freeing.
 * Calling the destructor on an object will destroy all AP_EventWriter
 * objects it refers to in a recursive manner - thus, only the top-level
 * object should be destroyed, and it is not possible to share
 * AP_EventWriter objects between two different containing events.
 *
 */
```

The C/C++ codec Plug-in Development Specification

Communication with other layers

A decoding codec plug-in's role is to decode messages from a transport layer plug-in into a normalized format that can be processed by the Semantic Mapper. To achieve this it needs to be able to communicate with the Semantic Mapper. The accessible Semantic Mapper functionality is presented in `SemanticMapper.h`.

When a decoding codec starts it is passed a handle to an `AP_SemanticMapper` object through its `setSemanticMapper` function. This object is defined in `SemanticMapper.h` as follows:

```
**
 * AP_SemanticMapper
 *
 * External (client-visible) interface to the Semantic Mapper component of
```

```
 * an IAF instance.  The AP_SemanticMapper struct contains contains a table
 * of function pointers, declared in the AP_SemanticMapper_Functions struct
 * above.  The implementation of these functions is private.  Users of the
 * Semantic Mapper should invoke functions on it as in the following
 * example (mapper is of type AP_SemanticMapper*):
 *
 * i = mapper->functions->sendNormalisedEvent(mapper, event);
 */
struct AP_SemanticMapper{
  /**
   * Pointer to private internal data.
   */
  void* reserved;
  /**
   * Function table of Semantic Mapper operations.  See documentation for
   * AP_SemanticMapper_Functions for details.
   */
  struct AP_SemanticMapper_Functions* functions;
};
typedef struct AP_SemanticMapper AP_SemanticMapper;
```

where `functions`, (of type `AP_SemanticMapper_Functions*`) points to the definitions for two functions: `sendNormalisedEvent` and `getLastError`:

```
/**
 * AP_SemanticMapper_Functions
 *
 * Table of client visible functions exported by a Semantic Mapper
 * instance. These functions declare the only operations that may be
 * performed by users of a Semantic Mapper.
 *
 * Note that all of these functions take an initial AP_SemanticMapper*
 * argument; this is analogous to the (hidden) 'this' pointer passed to a
 * C++ object when a member function is invoked on it.
 */
struct AP_SemanticMapper_Functions {
  /**
   * sendNormalisedEvent
   *
   * Send a customer-specific event to the Semantic Mapper.  The event will
   * be translated into a single Apama event that will be queued for
   * injection into the Engine.
   *
   * @param mapper The Semantic Mapper to send the event to.
   * @param event The event to send, represented as a set of name-value
   * pairs.
   * @param timeStamp Timestamps associated with this event
   * @return Semantic Mapper error code.  If this is not
   * AP_SemanticMapper_OK, the getLastError() function should be called to
   * get a more detailed description of what went wrong.
   */
  AP_SemanticMapperError (*sendNormalisedEvent)(
      struct AP_SemanticMapper* mapper, AP_NormalisedEvent* event,
      AP_TimestampSet* timeStamp);
  /**
   * getLastError
   *
   * Return the Semantic Mapper's stored error message, if any.  The message
   * string is owned by the mapper so should not be modified or freed by the
   * caller.
   *
   * @param mapper The Semantic Mapper instance
   * @return The last error message generated by the mapper
   */
  const AP_char8* (*getLastError)(struct AP_SemanticMapper* mapper);
};
```

Code inside a decoding codec that calls these functions on the Semantic Mapper looks as follows. Assuming that `mapper` holds a reference to the `AP_SemanticMapper` object:

```
errorCode = mapper->functions->sendNormalisedEvent(mapper, NormalisedEvent);
```

and likewise for `getLastError`.

The error codes that may be returned by `sendNormalisedEvent` are as follows:

```
/**
 * AP_SemanticMapperError
 *
 * Error codes that can be returned by operations on the Semantic Mapper.  The
 * enumeration values follow the normal UNIX convention of zero == OK and
 * non-zero == error.
 */
typedef enum {
  AP_SemanticMapper_OK = 0,          /* Everything is fine */
  AP_SemanticMapper_InternalError,   /* Some unspecified internal error occurred */
  AP_SemanticMapper_MappingFailure,  /* Couldn't convert customer event to Apama event */
  AP_SemanticMapper_InjectionFailure /* Couldn't queue converted event for injection into Engine */
} AP_SemanticMapperError;
```

On the other hand, an encoding codec plug-in's role is to encode messages in normalized format into some specific format that can then be accepted by a transport layer plug-in for transmission to an external message sink (like a message bus). To achieve this it needs to be able to communicate with a transport layer plug-in loaded in the IAF.

When an encoding codec starts its `addEventTransport` function will be called once for each available transport. For each, it is passed a handle to an `AP_EventTransport` object. This object is defined in `EventTransport.h` and was described in detail in "C/C++ Transport Plug-in Development" on page 48. As stated in "Defining the codec function tables" on page 66, it contains a pointer to `AP_EventTransport_Functions`, which in turn references the functions available in the transport layer plug-in. Of these, only two are relevant to the author of an encoding codec:

sendTransportEvent

```
AP_EventTransportError (*sendTransportEvent)(
    struct AP_EventTransport* transport, AP_TransportEvent event,
    AP_TimestampSet* timeStamp);
```

getLastError

```
const AP_char8* (*getLastError)(struct AP_EventTransport* transport);
```

Code inside an encoding codec that calls these functions on the transport layer plug-in looks as follows. Assuming that `transport` holds a reference to the `AP_EventTransport` object:

```
errorCode = transport->functions->sendTransportEvent(transport, event);
```

and likewise for `getLastError`.

The C/C++ codec Plug-in Development Specification

Working with normalized events

The function of a decoding codec plug-in is to convert incoming messages into a standard normalized event format that can be processed by the Semantic Mapper. Events sent upstream to an encoding codec plug-in are provided to the plug-in in this same format.

Normalized events are essentially dictionaries of name-value pairs, where the names and values are both character strings. Each name-value pair nominally represents the name and content of a single field from an event, but users of the data structure are free to invent custom naming schemes

to represent more complex event structures. Names must be unique within a given event. Values may be empty or `NULL`.

Some examples of normalized event field values for different types are:

- `string  "a string"`
- `integer  "1"`
- `float  "2.0"`
- `decimal "100.0d"`
- `sequence<boolean>  "[true,false]"`
- `dictionary<float,integer> "{2.3:2,4.3:5}"`
- `SomeEvent  "SomeEvent(12)"`

Note: When assigning names to fields in normalized events, keep in mind that the `fields` and `transport` attributes for event mapping conditions and event mapping rules both use a list of fields delimited by spaces or commas. This means, for example that `<id fields="Exchange EX,foo" test="==" value="LSE"/>` will successfully match a field called "Exchange", "EX" or "foo", but *not* a field called "Exchange EX,foo". While fields with spaces or commas in their names may be included in a payload dictionary in upstream or downstream directions, they cannot be referenced directly in mapping or id rules.

To construct strings for the normalized event fields representing container types (dictionaries, sequences, or nested events), use the EventWriter utility found in the `AP_EventWriter.h` header file, which is located in the `include` directory of the Apama installation. The following examples show how to add a sequence and a dictionary to a normalized event (note the escape character " `\` " used in order to insert a quotation mark into a string).

```
#include <AP_EventWriter.h>
AP_EventWriter *map, *list;
AP_NormalisedEvent *event;
AP_EventWriterValue key, value;
list=AP_EventWriter_ctor(AP_SEQUENCE, NULL);
list->addString(list, "abc");
list->addString(list, "de\"f");
map=AP_EventWriter_ctor(AP_DICTIONARY, NULL);
      key.stringValue="key1"; value.stringValue="value";
map->addDictValue(map, AP_STRING, key, AP_STRING, value);
key.stringValue="key\"{}2";
value.stringValue="value\"{}2";
map->addDictValue(map, AP_STRING, key, AP_STRING, value);
event=AP_NormalisedEvent_ctor();
event->functions->addQuick(event, "mySequenceField",
event->functions->list->toString(list));
event->functions->event->functions->addQuick(event,
       "myDictionaryField", event->functions->map->toString(map));
AP_EventWriter_dtor(list);
AP_EventWriter_dtor(map);
```

Fields names and values of normalized events are in UTF-8 format. This means that the writer of the codec needs to ensure that downstream events are correctly formed and the codec should expect to handle UTF-8 coming upstream.

The `NormalisedEvent.h` header file defines objects and functions that make up a special programming interface for constructing and examining normalized events.

NormalisedEvent.h contains two main structures: AP_NormalisedEvent which represents a single normalized event, and AP_NormalisedEventIterator which can be used to step through the contents of a normalized event structure.

The C/C++ codec Plug-in Development Specification

The AP_NormalisedEvent structure

The AP_NormalisedEvent structure has a pointer to a table of client-visible functions exported by the object called AP_NormalisedEvent_Functions. This function table provides access to the operations that may be performed on the event object:

- size - Return the number of elements (name-value pairs) currently stored by the event.
- empty - Check whether the event is empty or not.
- add - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten. This returns an iterator into the events at the point where the new element was added.
- addQuick - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten.
- remove - Remove the named element from the normalized event.
- removeAll - Remove all elements from the normalized event. The empty function will return AP_TRUE after this function has been called.
- replace - Change the value of a named element in the normalized event. If an element with the given name already exists, its value will be replaced with a copy of the given value. Otherwise, a new element is created just as though addQuick had been called.
- exists - Check whether a given element exists in the normalized event.
- find - Search for a named element in the normalized event. Returns an iterator into the event at the point where the element was located.
- findValue - Search for a named element in the normalized event and return its value.
- findValueAndRemove - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event.
- findValueAndRemove2 - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event. Unlike findValueAndRemove it tells you whether it was able to remove the value, or whether it did not exist.
- first - Return an iterator pointing to the first element of the normalized event. Successive calls to the next function of the returned iterator will allow you to visit all the elements of the event.
- last - Return an iterator pointing to the last element of the normalized event. Successive calls to the back function of the returned iterator will allow you to visit all the elements of the event.
- toString - Return a printable string representation of the normalized event. The returned string is owned by the caller and should be freed when it is no longer required.
- char8free – Free a string returned by the toString function; should be called when the string is no longer required.

All of these functions take an initial `AP_NormalisedEvent*` argument; this is analogous to the implicit '`this`' pointer passed to a C++ object when a member function is invoked on it.

In addition, the `AP_NormalisedEvent_ctor` constructor function is provided to create a new event instance. `AP_NormalisedEvent_dtor` destroys a normalized event object, and should be when the event is no longer required to free up resources.

The reader is invited to examine the header file for the full definitions of all these functions.

Working with normalized events

The AP_NormalisedEventIterator structure

Some of the functions above refer to a normalized event 'iterator'. This object is for stepping through the contents of a normalized event, in forwards or reverse order.

The `AP_NormalisedEventIterator` structure contains a function table defined by `AP_NormalisedEventIterator_Functions`, which includes all of the functions exported by a normalized event iterator:

- `valid` - Check whether the iterator points to a valid element of the normalized event. Typically used as part of the loop condition when iterating over the contents of an event.
- `key` - Return the key (name) associated with the current event element pointed to by the iterator. The returned value is owned by the underlying normalized event and should not be modified or freed by the caller.
- `value` - Return the value associated with the current event element pointed to by the iterator. The returned value is owned by the underlying normalized event and should not be modified or freed by the caller.
- `next` - Move the iterator to the next element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `AP_TRUE`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.
- `back` - Move the iterator to the previous element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `AP_TRUE`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.

Note that all of these functions take an initial `AP_NormalisedEventIterator*` argument; this is analogous to the implicit '`this`' pointer passed to a C++ object when a member function is invoked on it.

`AP_NormalisedEventIterator_dtor` destroys a normalized event iterator object, and should be called when the iterator is no longer required to free up resources. There is no public constructor function; iterators are created and returned only by `AP_NormalisedEvent` functions.

Please see the `NormalisedEvent.h` header file for more information about the structures and functions introduced in this section.

Working with normalized events

Transport Example

As part of the IAF distribution Apama includes the `FileTransport` transport layer plug-in, implemented in the `samples\iaf_plugin\c\simple\FileTransport.c` source file.

The `FileTransport` plug-in can read and write messages from and to a text file, and it is recommended that developers examine this sample to see what a typical transport plug-in implementation looks like.

See "IAF samples" on page 45 for more information about this sample. "The File Transport plug-in" on page 140 describes how the `FileTransport` plug-in can be used in practice.

C/C++ Codec Plug-in Development

Getting Started with codec layer plug-in development

Note: Before developing a new codec plug-in, it is worth considering whether one of the standard Apama IAF plug-ins could be used instead; see "Standard Plug-ins" on page 137 for more information.

In order to facilitate quick development of new codec plug-ins your distribution includes a codec plug-in skeleton.

This file, called `skeleton-codec.c`, implements a complete codec plug-in that complies with the Codec Plug-in Development Specification but where the entire custom message format encoding/decoding functionality is missing. The file is located in the `samples\iaf_plugin\c\skeleton` directory of your installation.

In order to turn the skeleton into a fully operational message format specific codec plug-in, the plug-in author needs to fill in the gaps within the codec generic, decoding and encoding functions; `updateProperties`, `getLastErrorCodec`, `getStatus`, `sendNormalisedEvent`, `flushUpstream`, `getLastErrorEncoder`, `addEventTransport`, `removeEventTransport`, `sendTransportEvent`, `setSemanticMapper`, `flushDownstream`, and `getLastErrorDecoder`. These must implement their specified functionality in the context of the custom message format. The information, constructor and destructor functions are also likely to require adaptation.

The skeleton defines a structure, called `EventCodec_Internals`, to store all its private data, and this structure is placed within the `reserved` field of the `AP_EventCodec` object created within the constructor method. It is likely that this structure will need to be modified to contain additional data that the adapter might require.

The distribution also contains a `makefile` (for use with GNU Make on UNIX), as well as a workspace file and `dsp` folder, for use with Microsoft's Visual Studio .NET on Microsoft Windows, for this skeleton, which can be adapted to compile your codec plug-in and link it against any custom libraries required.

Once a plug-in is built, it needs to be placed in a location where it can be picked up by the IAF.

This means that on Windows you either need to copy the `.dll` into the `bin` folder, or else place it somewhere which is on your 'path', that is a location that is referenced by the `PATH` environment variable.

On UNIX you either need to copy the `.so` into the `lib` directory, or else place it somewhere which is on your 'library path', that is a directory that is referenced by the `LD_LIBRARY_PATH` environment variable.

C/C++ Codec Plug-in Development

Chapter 5: C/C++ Plug-in Support APIs

This section describes other programming interfaces provided with the Apama software that may be useful in implementing transport layer and codec plug-ins for the IAF.

Logging from plug-ins in C/C++

This API provides a mechanism for recording status and error log messages from the IAF runtime and any plug-ins loaded within it. Plug-in developers are encouraged to make use of the logging API instead of custom logging solutions so that all the information may be logged together in the same standard format and log file(s) used by other plug-ins and the IAF runtime.

The logging API also allows control of logging verbosity, so that any messages below the configured logging level will not be written to the log. The logging level and file are initially set when an adapter first starts up – see "Logging configuration (optional)" on page 43 for more information about the logging configuration.

The C/C++ interface to the logging system is declared in the header file `AP_Logger.h`, which can be found in the `include` directory of the Apama installation. All users of the logging system should include this header file. The types and functions of interest to IAF plug-in writers are:

AP_LogLevel

```
/**
 * Enumeration of logging verbosity levels.  In order of increasing verbosity,
 * the levels are: NULL < OFF < CRIT < FATAL < ERROR < WARN < INFO < DEBUG < TRACE.
 * Messages logged at level X will be sent to the log if the current
 * level is >= X.
 */
enum AP_LogLevel {
  AP_LogLevel_NULL,
  AP_LogLevel_OFF,
  AP_LogLevel_CRIT,
  AP_LogLevel_FATAL,
  AP_LogLevel_ERROR,
  AP_LogLevel_WARN,
  AP_LogLevel_INFO,
  AP_LogLevel_DEBUG,
  AP_LogLevel_TRACE
};
```

Note: `AP_LogLevel_NULL` means “no log level has been set” and should be interpreted by IAF and plug-ins as “use the default logging level.”

AP_LogTrace

```
/**
   * Log a message at TRACE level.  Note that the message will only appear
   * in the log file if the current logging level is AP_LogLevel_Trace.
   *
   * @param message The message to be logged.  This is a standard printf()
   * format string; any values required by the formatting should be passed
   * as additional arguments to the AP_LogTrace() call.
   */
```

```
AP_COMMON_API void AP_COMMON_CALL AP_LogTrace(const char* message, ...);
```

Along with the other logging functions described below, `AP_LogTrace` is based on the standard C library `printf` function. The message parameter may contain `printf` formatting characters that will be filled in from the remaining arguments.

AP_LogDebug

```
/**
   * Log a message at DEBUG level.  Note that the message will only appear
   * in the log file if the current logging level is AP_LogLevel_DEBUG or
   * greater.
   *
   * @param message The message to be logged.  This is a standard printf()
   * format string; any values required by the formatting should be passed
   * as additional arguments to the AP_LogDebug() call.
   */
AP_COMMON_API void AP_COMMON_CALL AP_LogDebug(const char* message, ...);
```

AP_LogInfo

```
/**
  * Log a message at INFO level.  Note that the message will only appear
  * in the log file if the current logging level is AP_LogLevel_INFO or
  * greater.
  *
   * @param message The message to be logged.  This is a standard printf()
   * format string; any values required by the formatting should be passed
   * as additional arguments to the AP_LogInfo() call.
   */
AP_COMMON_API void AP_COMMON_CALL AP_LogInfo(const char* message, ...);
```

AP_LogWarn

```
/**
   * Log a message at WARN level.  Note that the message will only appear
   * in the log file if the current logging level is AP_LogLevel_WARN or
   * greater.
   *
   * @param message The message to be logged.  This is a standard printf()
   * format string; any values required by the formatting should be passed
   * as additional arguments to the AP_LogWarn() call.
   */
AP_COMMON_API void AP_COMMON_CALL AP_LogWarn(const char* message, ...);
```

AP_LogError

```
/**
   * Log a message at ERROR level.  Note that the message will only appear
   * in the log file if the current logging level is AP_LogLevel_ERROR or
   * greater.
   *
   * @param message The message to be logged.  This is a standard printf()
   * format string; any values required by the formatting should be passed
   * as additional arguments to the AP_LogError() call.
   */
AP_COMMON_API void AP_COMMON_CALL AP_LogError(const char* message, ...);
```

AP_LogCrit

```
/**
   * Log a message at CRIT level.  Note that the message will only appear
   * in the log file if the current logging level is AP_LogLevel_CRIT or
   * greater.
   *
   * @param message The message to be logged.  This is a standard printf()
   * format string; any values required by the formatting should be passed
   * as additional arguments to the AP_LogCrit() call.
   */
AP_COMMON_API void AP_COMMON_CALL AP_LogCrit(const char* message, ...);
```

The logging API offers other functions to set and query the current logging level and output file. While these functions are available to plug-in code, it is recommended that plug-ins do not use them. The IAF core is responsible for updating the state of the logging system in response to adapter re-configuration requests.

C/C++ Plug-in Support APIs

Using the latency framework

The latency framework API provides a way to measure adapter latency by attaching high-resolution timing data to events as they stream into, through, and out of the adapter. Developers can then use these events to compute upstream, downstream, and round-trip latency numbers, including latency across multiple adapters.

The `sendNormalisedEvent()` and `sendTransportEvent()` functions contain an `AP_TimestampSet` parameter that carries the microsecond-accurate timestamps that can be used to compute the desired statistics.

C/C++ Plug-in Support APIs

C/C++ timestamp

A timestamp is an index-value pair. The index represents the point in the event processing chain at which the timestamp was recorded, for example “upstream entry to semantic mapper” and the value is a floating point number representing the time. The header file `AP_TimestampSet.h` defines a set of standard indexes but a custom plug-in can define additional indexes for even finer-grained measurements. When you add a custom index definition, be sure to preserve the correct order, for example, an index denoting an “entry” point should be less than an one denoting an “exit” point from that component.

Timestamps are relative measurements and are meant to be compared only to other timestamps in the same or similar processes on the same computer. Timestamps have no relationship to real-world “wall time”.

Using the latency framework

C/C++ timestamp set

A timestamp set is the collection of timestamps that are associated with an event. The latency framework API provides functions that developers can use to add, inspect, and remove timestamps from an event’s timestamp set.

Using the latency framework

C/C++ timestamp configuration object

Constructors and `updateProperties()` methods for transport and codec plug-ins take the following argument: `IAF_TimestampConfig`.

A timestamp configuration object contains a set of fields that a plug-in can use to decide whether to record and/or log timestamp information. Although timestamp configuration objects are passed to all transport and codec plug-ins, it is up to the authors of a plug-ins to write the code that makes use of them.

The fields in the object are:

- `recordUpstream` — If true, the plug-in should record timestamps for all upstream events it processes, and pass these along to the upstream component, if any.
- `recordDownstream` — If true, the plug-in should record timestamps for all downstream events it processes, and pass these along to the downstream component, if any.
- `logUpstream` — If true, the plug-in should log the latency for all upstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable upstream timestamp recording if upstream logging is enabled.
- `logDownstream` — If true, the plug-in should log the latency for all downstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable downstream timestamp recording if downstream logging is enabled.
- `logRoundtrip` — If true, the plug-in should log the "round trip" latency for all events it processes in either direction, if possible. At its simplest, the round trip latency can just be the difference between the largest and smallest timestamps passed to the plug-in, or an individual plug-in may choose to present some more plug-in-specific latency number. As with the other logging options, the logging level given by the `logLevel` member should be used.
- `logLevel` — The logging verbosity level to use if any of the timestamp logging options are enabled.

Using the latency framework

C/C++ latency framework API

The C/C++ interface for the latency framework is declared in the header file `AP_TimestampSet.h`. Plug-ins using the latency framework should include this file and also include the `IAF_TimestampConfig.h` header file, which declares the timestamp configuration object.

The functions of interest are the following.

addNow

```
/**
  * Add a new index-time pair to the timestamp.  The given index and current
  * time will be copied into the timestamp.  If an element with the same index
  * already exists, it will NOT be overwritten.
  *
  * @param timestamp The AP_TimestampSet instance
  * @param index The index of the new element
  */
  void (*addNow)(struct AP_TimestampSet* timestamp,
      AP_TimestampSetIndex index);
```

addTime

```
/**
  * Add a new index-time pair to the timestamp.  The given index and time will
  * be copied into the timestamp.  If an element with the same index already
  * exists, it will NOT be overwritten.
  *
  * @param timestamp The AP_TeimestampSet instance
  * @param index The index of the new element
  * @param time The time of the new element
  */
  void (*addTime)(struct AP_TimestampSet* timestamp,
      AP_TimestampSetIndex index, AP_TimestampSetTime time);
```

replace

```
/**
  * Change the time of an indexed element in the timestamp.  If an
  * element with the given index already exists, its time will be replaced
  * with a copy of the given time.  Otherwise, a new element is created
  * just as though addTime() had been called.
  *
  * @param timestamp The AP_TimestampSet instance
  * @param index The index of the element to be updated
  * @param time The new time for the indexed element
  */
  void (*replace)(struct AP_TimestampSet* timestamp,
        AP_TimestampSetIndex index, AP_TimestampSetTime newTime);
```

replaceWithNow

```
/**
  * Change the time of an indexed element in the timestamp.  If an
  * element with the given index already exists, its time will be replaced
  * with a copy of the given time.  Otherwise, a new element is created
  * just as though addNow() had been called.
  *
  * @param timestamp The AP_TimestampSet instance
  * @param index The index of the element to be updated
  * @param time The new time for the indexed element
  */
  void (*replaceWithNow)(struct AP_TimestampSet* timestamp,
        AP_TimestampSetIndex index);
```

findTime

```
/**
  * Search for an indexed element in the timestamp and return its
  * time.
  *
  * @param timestamp The AP_TimestampSet instance
  * @param index The element index to search for
  * @return The time associated with the given index, or NULL if no
  * matching element could be found.
  */
  AP_TimestampSetTime (*findTime)(struct AP_TimestampSet* timestamp,
      AP_TimestampSetIndex index);
```

findTimeAndRemove

```
/**
  * Search for an indexed element in the timestamp and return its
  * time.  If found, the element will also be removed from the timestamp.
  *
  * @param timestamp The AP_TimestampSet instance
  * @param index The element index to search for
  * @return The time associated with the given index, or NULL if no
  * matching element could be found.  If the element was found, it will be
  * deleted from the timestamp.  Note that this function cannot distinguish
  * between an element with a missing time and an element that does not
  * exist - NULL will be returned in either case. The returned object
  * (if any) must be explicitly deleted by the caller after use.
```

```
  */
 AP_TimestampSetTime (*findTimeAndRemove)(
      struct AP_TimestampSet* timestamp, AP_TimestampSetIndex index);
```

getSize

```
/**
  * Return the number of elements in the AP_TimestampSet
  *
  * @param timestamp The AP_TimestampSet instance
  * @return The number of elements in the timestamp set
  */
  AP_uint32 (*getSize)(struct AP_TimestampSet* timestamp);
```

toString

```
 w
/**
  * Return a printable string representation of the timestamp.  The
  * returned string is owned by the caller and should be freed when it is
  * no longer required using the char8free function.
  *
  * @param timestamp The AP_TimestampSet instance
  * @return Printable string representation of the timestamp
  */
  AP_char8* (*toString)(struct AP_TimestampSet* timestamp);
```

Using the latency framework

Chapter 6: Transport Plug-in Development in Java

The *transport layer* is the front-end of the IAF. The transport layer's purpose is to abstract away the differences between the programming interfaces exposed by different middleware message sources and sinks. It consists of one or more custom plug-in libraries that extract *downstream* messages from external message sources ready for delivery to the codec layer, and send Apama events already encoded by the codec layer *upstream* to the external message sink. See "The Integration Adapter Framework" on page 11 for a full introduction to transport plug-ins and the IAF's architecture.

An adapter should send events to the correlator only after its `start` function is called and before the `stop` function returns.

This topic includes the Transport Plug-in Development Specification for Java and additional information for developers of Java event transports. "C/C++ Transport Plug-in Development" on page 48 provides analogous information about developing transport plug-ins using C/C++.

The Transport Plug-in Development Specification for Java

A Java transport layer plug-in is implemented as a Java class extending `AbstractEventTransport`. Typically this class would be packaged up, together with any supporting classes, as a Java Archive (`.jar`) file.

To comply with Apama's Transport Plug-in Development Specification, an event transport class must satisfy two conditions:

1. It must have a constructor with the signature:

   ```
   public AbstractEventTransport(
              String name,
              EventTransportProperty[] properties,
              TimestampConfig timestampConfig)
           throws TransportException
   ```

   This will be used by the IAF to instantiate the plug-in.

2. It must extend the `com.apama.iaf.plugin.AbstractEventTransport` class, correctly implementing all of its abstract methods.

(These methods are mostly directly equivalent to the functions with the same names in the C/C++ Transport Plug-in Development Specification.)

Note that all Java plug-ins are dependent on classes in `jplugin_public5.2.jar`, so this file must always be on the classpath during plug-in development. It is located in the Apama installation's `lib` directory.

Unless otherwise stated, Java classes referred to in this topic are members of the `com.apama.iaf.plugin` package, whose classes and interfaces are contained in this `.jar`.

Transport Plug-in Development in Java

Java transport functions to implement

HTML Javadoc documentation for `AbstractEventTransport` and related classes is provided as part of the Apama documentation set. The `Javadoc` is located in the Apama installation's `doc\javadoc` directory.

This topic includes the text of the Javadoc documentation for the functions a transport plug-in author needs to implement, in addition to some pointers on the idiomatic way in which such plug-ins are usually written.

The Constructor

```
/**
 * Construct a new instance of AbstractEventTransport. All subclasses MUST
 * provide a constructor with the same signature, which will be used by the
 * IAF to create an instance of the transport class.
 *
 * The AbstractEventTransport implementation does nothing, but subclasses
 * should make use of the arguments to initialize the transport.
 *
 * @param name The transport name, as specified in the IAF config file
 * @param properties The transport property set specified in the IAF
 * configuration file
 * @param timestampConfig Timestamp recording/logging settings from the IAF
 * configuration file
 * @throws TransportException
 */
public AbstractEventTransport(String name,
         EventTransportProperty[] properties
         TimestampConfig timestampConfig)
   throws TransportException
```

A typical constructor would create a logger using the plug-in `name` provided (see "Logging from plug-ins in Java" on page 111), make a call to the `updateProperties` method to deal with the initial property set passed in, and perform any other initialization operations required for the particular transport being developed.

updateProperties

```
/**
 * Update the configuration of the transport. The transport may assume
 * that stop(), flushUpstream() and flushDownstream() have all been called
 * before this function is invoked. The recommended procedure for
 * updating properties is to first compare the new property set with the
 * existing stored properties - if there are no changes, no action should
 * be taken.
 *
 * @param properties The new transport property set specified in the IAF
 * configuration file
 * @param TimestampConfig timestampConfig
 * @throws TransportException
 */
abstract public void updateProperties(EventTransportProperty[] properties,
      TimestampConfig timestampConfig)
    throws TransportException;
```

The `properties` array contains an `EventTransportProperty` object for each plug-in property specified in the IAF configuration file (in order). The `getName` and `getValue` methods allow the plug-in to retrieve the name and value of each property as `String` objects.

See the Javadoc documentation for more information about the `EventTransportProperty` class.

addEventDecoder

```
/**
 * Add a named event decoder to the set of decoders known to the
 * transport. If the named decoder already exists, it should be
 * replaced.
 *
 * @param name The name of the decoder to be added
 * @param decoder The decoder object instance
 * @throws TransportException
 */
abstract public void addEventDecoder(String name, EventDecoder decoder)
    throws TransportException;
```

In an adapter in which multiple event codecs could be present, this function would usually be implemented by storing the <`name`, `decoder`> pair in a Java map, from which the `EventDecoder` could later be retrieved using a plug-in property (e.g. "`decoderName`") to determine which of the plug-ins in the map should be used.

Alternatively, if this transport plug-in will only ever be used in an adapter with just one codec plug-in, this method can be implemented simply by storing the provided `EventDecoder` object in an instance field.

See "Communication with the codec layer" on page 93 for more information.

removeEventDecoder

```
/**
 * Remove a named event decoder from the set of decoders known to the
 * transport. If the named decoder does not exist, the function should do
 * nothing.
 *
 * @param name The decoder to be removed
 * @throws TransportException
 */
abstract public void removeEventDecoder(String name)
    throws TransportException;
```

This method is usually implemented by removing the named codec plug-in from a map, or nulling out a field holding the previously added `EventCodec`.

flushUpstream

```
/**
 * Flush any pending transport events onto the transport. The transport may
 * assume that the stop() function has been called before this function,
 * so in many cases no action will be required to complete the flushing
 * operation.
 *
 * @throws TransportException
 */
abstract public void flushUpstream() throws TransportException;
```

Usually has a blank implementation, unless there is some kind of upstream buffering.

flushDownstream

```
/**
 * Flush any pending transport events into the decoder. The transport may
 * assume that the stop() function has been called before this function,
 * so in many cases no action will be required to complete the flushing
 * operation.
```

```
 *
 * @throws TransportException
 */
abstract public void flushDownstream() throws TransportException;
```

Usually has a blank implementation, unless there is some kind of downstream buffering.

start

```
/**
 * Start processing incoming data from the external transport.
 *
 * @throws TransportException
 */
abstract public void start() throws TransportException;
```

This is where a plug-in should establish a connection to its external message source/sink, often by starting a new thread to process or poll for new downstream messages. Events should not be sent to the correlator until the `start` method has been called.

"Communication with the codec layer" on page 93 explains how a transport plug-in can pass downstream messages it receives from an external source on to a codec plug-in.

stop

```
/**
 * Stop processing incoming data from the external transport. New events
 * arriving may be blocked, queued or simply dropped, but under no
 * circumstances should any be sent into the adapter until the start()
 * function is called.
 *
 * @throws TransportException
 */
abstract public void stop() throws TransportException;
```

Here, a plug-in may close existing connections and interrupt running threads to ensure that no more messages are passed to the event codec. Events should not be sent to the correlator after the `stop` method has returned. The `stop` method must wait for any other threads sending events to complete before the `stop` method returns.

cleanup

```
/**
 * Frees any resources allocated by the transport (useful for resources
 * external to the JVM that were allocated in the constructor). The IAF
 * guarantees to call this method exactly once.
 *
 * @throws TransportException
 */
abstract public void cleanup() throws TransportException;
```

This is where any heavy-weight threads or data structures used for interfacing with the external transport that do not get cleaned up when the plug-in is stopped should be destroyed.

getStatus

```
/**
 * Return a TransportStatus class containing up-to-date status information
 * for the transport.
 *
 * @return An immutable TransportStatus class containing status
 * information.
 */
abstract public TransportStatus getStatus();
```

This method provides the statistics and status message displayed by the IAF Watch tool. A typical plug-in will continuously keep track of the number of messages sent upstream and downstream. Then, when `getStatus` is called, these message counts can simply be packaged up in a new

`TransportStatus` object together with a `String` describing the current status of the plug-in (maximum length 1024 characters), and returned.

For example:

```
public TransportStatus getStatus()
{    String status = (started) ? "Status: Running" : "Status: Not running";
    return new TransportStatus(status, totalReceived, totalSent);
}
```

See the Javadoc documentation for more information about the `TransportStatus` class.

`getAPIVersion`

```
/**
 * Return the transport API version that the transport was built against.
 * @return Must be EventTransport.API_VERSION.
 */
public abstract int getAPIVersion();
```

Always return `EventTransport.API_VERSION`.

`sendTransportEvent` (send **`upstream`**)

```
/**
 * Called by an event encoder to send an upstream message to the external
 * transport.
 *
 * It is assumed that the encoder and transport share
 * the same definition of the content of the event, so that the transport
 * can effectively interpret the event.
 *
 * @param event An object representing the event to be sent by the
 * transport, in a format shared by the encoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 *
  * @throws TransportException Thrown by the transport if any error occurs
  * sending the message.
  */
public abstract void sendTransportEvent(Object event,
         TimestampSet timestamps)
    throws TransportException;
```

This is the method that a codec layer plug-in calls when it receives a translated upstream Apama event that needs to be sent on to an event transport for transmission to an external message sink.

Note that there are no guarantees about which threads might be used to call this method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures here.

See "Communication with the codec layer" on page 93 for more information about processing upstream events received from a codec plug-in.

The Transport Plug-in Development Specification for Java

Communication with the codec layer

This section discusses how the transport layer communicates with the codec layer in both the upstream and downstream directions.

The Transport Plug-in Development Specification for Java

Sending upstream messages received from a codec plug-in to a sink

When a codec plug-in has encoded an event ready for transmission by a transport plug-in it will pass it on calling the transport's `sendTransportEvent` method (as defined above). It is then up to the transport plug-in to process the message (which will be of some type agreed by the codec and transport plug-in authors), and send it on to the external sink it provides access to.

Note that there are no guarantees about which threads might call this method, so plug-in authors will need to consider thread synchronization issues carefully.

If there is a problem sending the event on, the transport plug-in should throw a `TransportException`.

Communication with the codec layer

Sending downstream messages received from a source on to a codec plug-in

In order that messages can be easily sent on to a codec plug-in, an event transport will usually have saved a reference to the event codec(s) it will be using before it establishes a connection to the external source.

Typically an event transport will build up a list of registered codec plug-ins from the parameters passed to the `addEventDecoder` and `removeEventDecoder` methods. If this is the case, the `start` method of the plug-in can select one of these plug-ins on the basis of a plug-in property provided in the configuration file (e.g. `<property name="decoderName" value="MyCodec"/>`), and saving it in an instance field (e.g. `currentDecoder`).

Once the plug-in has a reference to the event codec (or codecs) it will use, whenever an external message is received it should be passed on by calling the `sendTransportEvent` method on the codec plug-in (from the `EventDecoder` interface):

```
/**
 * Called by the event transport to decode a downstream event using a Java
 * Codec, which will then send it on to the Semantic Mapper.
 *
 * It is assumed that the encoder and transport share the same definition
 * of the content of the event, so that the transport can effectively
 * interpret the event.
 *
 * @param event An object representing the event to be decoded, in a format
 * shared by the decoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws CodecException Thrown by the decoder if the event provided
 * has an invalid format.
 * @throws SemanticMapperException Thrown if an error occurred during
 * processing of the message by the Semantic Mapper.
 */
public void sendTransportEvent(Object event,
      TimestampSet timestamps)
    throws CodecException, SemanticMapperException;
```

For example, part of the event processing code for a transport plug-in might be:

```
MyCustomMessageType message = myCustomMessageSource.getNextMessage();
currentDecoder.sendTransportEvent(message, timestamps);
```

If an error occurs in the codec or Semantic Mapper layers preventing the message from being converted into an Apama event, a `CodecException` or `SemanticMapperException` is thrown. Like all per-message errors, these should be logged at `Warning` level, preferably with a full stack trace logged

at `Debug` level too. If necessary, transports may also send messages downstream to the correlator to inform running monitors about the error.

When a transport sends a message to the codec via the `sendTransportEvent` method, it passes an `Object` reference and this allows custom types to be passed between the two plug-ins. However, any custom types should be loaded via the main (parent) classloader, as each plug-in specified in the IAF configuration file is loaded with its own classloader. Consider, for example, the following three classes all loaded into a single jar file, `MyAdapter.jar`, which is used in the IAF configuration file in the `jarName` attribute of the `<transport>` element.:

- `MyTransport.class`
- `MyCodec.class`
- `MyContainer.class` (the container class used in the call to `sendTransportEvent`)

When you load the transport and codec, a new classloader is used for each. This means both have their own copy of the `MyContainer` class. When the transport creates an instance of `MyContainer` and then passes it into the codec, the codec will recognize that the Object `getClass().getName()` is `MyContainer`, but will not be able to cast it to this type as its `MyContainer` class is from a different classloader.

To prevent this from happening, make sure that all shared classes are in a separate jar that is specified by a `<classpath>` element. The shared classes are then loaded by the parent classloader. This ensures that when a codec or transport references a shared class, they will both agree it is the same class.

Note that any codec plug-in called by a Java transport plug-in must also be written in Java.

Communication with the codec layer

Transport exceptions

`TransportException` is the exception class that should be thrown by a transport plug-in whenever the IAF calls one of its methods and an error prevents the method from successfully completing — for example, a message that cannot be sent on to an external sink in `sendTransportEvent`, or a serious problem that prevents the plug-in from initializing when `start` is called.

A `TransportException` object always has an associated `message`, which is a `String` explaining the problem (this may include information about another exception that caused the `TransportException` to be thrown). There is also a `code` field that specifies the kind of error that occurred; the possible codes are defined as constants in the `TransportException` class:

```
/**
 * Some unspecified internal error occurred
 */
public static final int INTERNALERROR = 1;
/**
 * Trouble reading/writing the external transport
 */
public static final int TRANSPORTFAILURE = 2;
/**
 * Trouble sending transport event to decoder
 */
public static final int DECODINGFAILURE = 3;
/**
 * Transport was passed an invalid property set
 */
```

```
public static final int BADPROPERTIES = 4;
```

`TransportException` defines a number of constructors, to make it easy to set up the exception's information quickly in different situations:

```
/**
 * Constructs a TransportException from a string message describing the
 * error, and assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error.
 */
public TransportException(String message) { ... }
/**
 * Constructs a TransportException from a string message describing the
 * error and a numeric code representing the class of error.
 *
 * @param message The cause of the error.
 * @param code One of the TransportException error codes.
 */
public TransportException(String message, int code) { ... }
/**
 * Constructs a TransportException from a string message describing the
 * error, and an exception object that is the cause of the error. It
 * assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error. This message will be suffixed
 * with the message of the 'cause' exception.
 * @param cause The exception object that caused the error.
 */
public TransportException(String message, Throwable cause) { ... }
/**
 * Constructs a TransportException from a string message describing the
 * error, a numeric code representing the class of error and an exception
 * object that is the cause of the error.
 *
 * @param message The cause of the error. This message will be suffixed
 * with the message of the 'cause' exception.
 * @param cause The exception object that caused the error.
 * @param code One of the TransportException error codes.
 */
public TransportException(String message, Throwable cause, int code) { ... }
```

The Transport Plug-in Development Specification for Java

Logging

Please see "Logging from plug-ins in Java" on page 111 for information about how transport plug-ins should log error, status and debug information.

The Transport Plug-in Development Specification for Java

Example

As part of the IAF distribution, Apama includes the `JFileTransport` transport layer plug-in, in the `samples\iaf_plugin\java\simple\src` directory.

The `JFileTransport` plug-in can read and write messages from and to a text file, and it is recommended that developers examine this sample to see what a typical transport plug-in implementation looks like.

See "IAF samples" on page 45 for more information about this sample. The section "The File Transport plug-in" on page 140 describes how the `JFileTransport` plug-in can be used in practice.

Transport Plug-in Development in Java

Getting started with Java transport layer plug-in development

Your distribution includes a complete 'skeleton' implementation of a transport layer plug-in in order to make development of new plug-ins faster.

This is located in the `samples\iaf_plugin\java\skeleton\src` directory of the installation, in a file called `SkeletonTransport.java`. The `SkeletonTransport` class complies fully with the Transport Plug-in Development Specification, but contains none of the custom message source/sink functionality that would be present in a full transport plug-in.

The skeleton starts a background thread to do the actual message reading. This is required unless the message source can asynchronously call back into the class that implements the plug-in.

The code contains `TODO:` comments indicating the main changes that need to be made to add support for a specific message source/sink. These include:

- Adding code to `sendTransportEvent` for sending an upstream event received from an event codec on to the external message sink (if supported).
- Adding code to the `run` method of the `MessageProcessingThread` for retrieving downstream messages from the external source and forwarding them on to an event codec (if supported).
- Alternatively, if the external message source works by making asynchronous calls using the listener pattern, the processing thread should usually be removed, and much of the code can be moved directly to the method called by the message source.
- Adding code to start communications with the external messaging system in the `start` method, and to ensure it is ceases in the `stop` method.
- Adding code to validate and save any new plug-in properties that are to be supported, in `updateProperties`.
- Adding code to initialize and clean up resources associated with the plug-in's operation. This would usually be done in the `start`/`stop` methods, in the background processing thread, or in the `updateProperties` and `cleanup` methods.

Depending on your requirements, it may also be necessary to make changes to the other methods – `addEventDecoder`, `removeEventDecoder`, `flushUpstream`, `flushDownstream`, `getStatus`, and the constructor.

The `skeleton` directory includes an Apache Ant build file called `build.xml` that provides a convenient way to build `.jar` files of compiled classes from plug-in source files, ready for use with the IAF.

Chapter 7: Java Codec Plug-in Development

The *codec layer* is a layer of abstraction between the transport layer and the IAF's Semantic Mapper. It consists of one or more plug-in libraries that perform message *encodinganddecoding*. Decoding involves translating *downstream* messages retrieved by the transport layer into the standard 'normalised event' format on which the Semantic Mapper's rules run; encoding works in the opposite direction, converting *upstream* normalized events into an appropriate format for transport layer plug-ins to send on. Note that unlike the situation with C/C++, in Java codec plug-ins are always both encoders and decoders. See "The Integration Adapter Framework" on page 11 for a full introduction to codec plug-ins and the IAF's architecture.

This chapter includes the Codec Plug-in Development Specification for Java and additional information for developers of Java event codecs. "C/C++ Codec Plug-in Development" on page 62 provides analogous information about developing codec plug-ins using C/C++.

Before developing a new codec plug-in, it is worth considering whether one of the standard Apama plug-ins could be used instead. "Standard Plug-ins" on page 137 provides more information on the standard IAF codec plug-ins: `JStringCodec` and `JNullCodec`. The `JStringCodec` plug-in codes normalized events as formatted text strings. The `JNullCodec` plug-in is useful in situations where it does not make sense to decouple the codec and transport layers, and allows transport plug-ins to communicate with the Semantic Mapper directly using normalized events.

The Codec Plug-in Development Specification for Java

A Java codec layer plug-in is implemented as a Java class extending `AbstractEventCodec`. Typically this class would be packaged up, together with any supporting classes, as a Java Archive (`.jar`) file.

To comply with Apama's Codec Plug-in Development Specification, an event codec class must satisfy two conditions:

1. It must have a constructor with the signature:

   ```
   public AbstractEventCodec(
         String name,
         EventCodecProperty[] properties,
         TimestampConfig timestampConfig)
      throws CodecException
   ```

   This will be used by the IAF to instantiate the plug-in.

2. It must extend the `com.apama.iaf.plugin.AbstractEventCodec` class, correctly implementing all of its abstract methods.

(These methods are mostly directly equivalent to the functions with the same names in the C/C++ Codec Plug-in Development Specification.)

Note that all Java plug-ins are dependent on classes in `jplugin_public5.2.jar`, so this file must always be on the classpath during plug-in development. It is located in the Apama installation's `lib` directory.

Unless otherwise stated, Java classes referred to in this chapter are members of the `com.apama.iaf.plugin` package, whose classes and interfaces are contained in this `.jar`.

Java Codec Plug-in Development

Java codec functions to implement

HTML Javadoc documentation for `AbstractEventCodec` and related classes is provided as part of the Apama documentation set. The `Javadoc` files are located in the Apama installation's `doc\javadoc` directory.

This topic includes the text of the Javadoc documentation for the functions that a codec plug-in author needs to implement, in addition to some pointers on how such plug-ins are usually written.

The Constructor

```
/**
  * Construct a new instance of AbstractEventCodec. All subclasses MUST
  * provide a constructor with the same signature, which will be used by the
  * IAF to create an instance of the codec class. <P><P>
  *
  * The AbstractEventCodec implementation does nothing, but subclasses
  * should make use of the arguments to initialize the codec.
  *
  * @param name The codec name, as specified in the IAF config file
  * @param properties The codec property set specified in the IAF
  * configuration file
  * @parm timestampConfig The timestamp recording/logging settings from the
  * IAF configuration file

  * @throws CodecException
  */
public AbstractEventCodec(String name, EventCodecProperty[] properties,
      TimestampConfig timestampConfig)
    throws CodecException
```

A typical constructor would create a logger using the plug-in `name` provided (see "Logging from plug-ins in Java" on page 111), make a call to the `updateProperties` method to deal with the initial property set passed in, and perform any other initialization operations required for the particular event codec being developed.

Note that unlike event transports, codec plug-ins do not have start and stop methods

updateProperties

```
/**
 * Update the configuration of the codec. The codec may assume
 * that flushUpstream() and flushDownstream() have been called
 * before this function is invoked. The recommended procedure for
 * updating properties is to first compare the new property set with the
 * existing stored properties - if there are no changes, no action should
 * be taken.
 *
 * @param properties The new codec property set specified in the IAF
 * configuration file
 * @param TimestampConfig timestampConfig
 * @throws CodecException
 */
```

```
abstract public void updateProperties(EventCodecProperty[] properties,
      TimestampConfig timestampConfig)
    throws CodecException;
```

The `properties` array contains an `EventCodecProperty` object for each plug-in property specified in the IAF configuration file (in order). The `getName` and `getValue` methods allow the plug-in to retrieve the name and value of each property as `String` objects.

See the Javadoc documentation for more information about the `EventCodecProperty` class.

addEventTransport

```
/**
 * Add a named event transport to the set of transports known to the
 * codec. If the named transport already exists, it should be
 * replaced.
 *
 * @param name The name of the transport to be added
 * @param transport The transport object instance
 * @throws CodecException
 */
abstract public void addEventTransport(String name, EventTransport transport)
  throws CodecException;
```

In an adapter in which multiple event transports could be present, this function would usually be implemented by storing the <`name`, `transport`> pair in a Java map, from which the `EventTransport` object could later be retrieved when required by the `sendNormalisedEvent` method, using a plug-in property (e.g. "`transportName`" to determine which of the plug-ins in the map should be used.

Alternatively, if this codec plug-in will only ever be used in an adapter with just one transport plug-in, this method can be implemented simply by storing the provided `EventTransport` object in an instance field.

See "Communication with other layers" on page 103 for more information.

removeEventTransport

```
/**
 * Remove a named event transport from the set of transports known to the
 * codec. If the named transport does not exist, the function should do
 * nothing.
 *
 * @param name The transport to be removed
 * @throws CodecException
 */
abstract public void removeEventTransport(String name)
    throws CodecException;
```

This method is usually implemented by removing the named transport plug-in from a map, or nulling out a field holding the previously added `EventTransport`.

setSemanticMapper

```
/**
 * Set the Semantic Mapper object to be used by the decoder.  Currently
 * only a single Semantic Mapper is supported in each adapter instance.
 *
 * @param mapper The Semantic mapper object instance
 * @throws CodecException
 */
abstract public void setSemanticMapper(SemanticMapper mapper)
  throws CodecException;
```

This method is usually implemented by storing the provided `SemanticMapper` object in an instance field, for use when sending on downstream messages.

flushUpstream

```
/**
 * Flush any pending codec events onto the codec. In many cases no action
 * will be required to complete the flushing operation.
 *
 * @throws CodecException
 */
abstract public void flushUpstream() throws CodecException;
```

Usually has a blank implementation, unless there is some kind of upstream buffering.

flushDownstream

```
/**
 * Flush any pending codec events into the decoder. In many cases no action
 * will be required to complete the flushing operation.
 *
 * @throws CodecException
 */
abstract public void flushDownstream() throws CodecException;
```

Usually has a blank implementation, unless there is some kind of downstream buffering.

cleanup

```
/**
 * Frees any resources allocated by the codec (useful for resources
 * external to the JVM that were allocated in the constructor). The IAF
 * guarantees to call this method exactly once.
 *
 * @throws CodecException
 */
abstract public void cleanup() throws CodecException;
```

This is where any external resources used by the event codec should be freed.

getStatus

```
/**
 * Return a CodecStatus class containing up-to-date status information
 * for the codec.
 *
 * @return An immutable CodecStatus class containing status
 * information.
 */
abstract public CodecStatus getStatus();
```

This method provides the statistics and status message displayed by the IAF Watch tool. A typical plug-in will continuously keep track of the number of messages sent upstream and downstream. Then, when `getStatus` is called, these message counts can simply be packaged up in a new `CodecStatus` object together with a `String` describing the current status of the plug-in (maximum length 1024 characters), and returned.

For example:

```
public CodecStatus getStatus()
{    String status = "Status: OK";
    return new TransportStatus(status, totalReceived, totalSent);
}
```

See the Javadoc documentation for more information about the `CodecStatus` class.

getAPIVersion

```
/**
 * Return the codec API version that the codec was built against.
 * @return Must be EventCodec.API_VERSION.
 */
public abstract int getAPIVersion();
```

Always return `EventCodec.API_VERSION`.

sendTransportEvent

```
/**
 * Called by the event transport to decode a downstream event using a Java
 * Codec, which will then send it on to the Semantic Mapper. It is assumed
 * that the encoder and transport share the same definition
 * of the content of the event, so that the transport can effectively
 * interpret the event.
 *
 * @param event An object representing the event to be decoded, in a format
 * shared by the decoder and transport.
 * @param timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws CodecException Thrown by the decoder if the event provided
 * has an invalid format.
 * @throws SemanticMapperException Thrown if an error occurred during
 * processing of the message by the Semantic Mapper.
 */
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws CodecException, SemanticMapperException;
```

This is the method that a transport layer plug-in calls when it receives a message that should be decoded and then sent downstream towards the Apama event correlator.

Note that there are no guarantees about which threads might be used to call this method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures here.

See "Communication with other layers" on page 103 for more information about processing downstream messages and passing them on to the Semantic Mapper.

sendNormalisedEvent (send **upstream**)

```
/**
 * Called by the Semantic Mapper to encode a normalized event and send
 * it directly through to the transport.
 *
 * @param event A NormalisedEvent representing the event to be encoded.
 * @params timestamps A TimestampSet representing the timestamps attached to
 * the event.
 *
 * @throws CodecException Thrown by the codec if the event provided
 * has an invalid format.
 * @throws TransportException Thrown if an error occurred in the Transport
 * when sending the message.
 */
public void sendNormalisedEvent(NormalisedEvent event,
      TimestampSet timestamps)
    throws CodecException, TransportException;
```

This is the method that the Semantic Mapper calls when it receives a message that should be encoded and then sent upstream to an event transport.

Note that there are no guarantees about which threads might be used to call this method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures here.

See "Communication with other layers" on page 103 for more information about processing upstream messages and passing them on to a transport plug-in. See "Working with normalized events" on page 106 for help working with `NormalisedEvent` objects.

The Codec Plug-in Development Specification for Java

Communication with other layers

This section discusses how the codec layer communicates with the transport layer and Semantic Mapper in upstream and downstream directions.

The Codec Plug-in Development Specification for Java

Sending upstream messages received from the Semantic Mapper to a transport plug-in

When the Semantic Mapper produces normalized events, it sends them on to the codec layer by calling the codec plug-ins' `sendNormalisedEvent` methods (as defined above). The event codec must then encode the normalized event for transmission by the transport layer.

In order to send messages upstream to an event transport, a codec plug-in must have a reference to the transport plug-in object. Typically, an event codec does this by building up a map of registered transport plug-ins from the parameters passed to the `addEventTransport` and `removeEventTransport` methods. It might then use a property provided in the configuration file (e.g. `<property name="transportName" value="MyTransport"/>`) to determine which event transport to use when the `sendNormalisedEvent` method is called.

Alternatively, if this transport plug-in will only ever be used in an adapter with just one codec plug-in, the `EventTransport` object could be stored in an instance field when it is provided to the `addEventTransport` method.

Once the plug-in has a reference to the event transport (or transports) it will use, it can pass on normalized events it has encoded into transport messages by calling the transport plug-in `sendTransportEvent` method:

```
/**
  * Called by an event encoder to send an upstream message to the external
  * transport.
  *
  * Ownership of the message is transferred to the transport when this
  * function is called. It is assumed that the encoder and transport share
  * the same definition of the content of the event, so that the transport
  * can effectively interpret the event.
  *
  * @param event An object representing the event to be sent by the
  * transport, in a format shared by the encoder and transport.
  * @param timestamps A TimestampSet representing the timestamps attached to
  * the event.
  *
  * @throws TransportException Thrown by the transport if any error occurs
  * sending the message.
  */
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws TransportException;
```

For example, the implementation of the event codec's `sendNormalisedEvent` could look something like this:

```
// Select EventTransport using saved plugin property value
EventTransport transport = eventTransports.get(currentTransportName);
// Encode message
MyCustomMessageType message = myEncodeMessage(event);
// Send to Transport layer plugin
transport.sendTransportEvent(message, timestamps);
```

If an error occurs in the transport layer a `TransportException` is thrown. Typically such exceptions do not need to be caught by the codec plug-in, unless the codec plug-in is able to somehow deal with the problem.

A `CodecException` should be thrown if there is an error encoding the normalized event.

Note that there are no guarantees about which threads might call the `sendNormalisedEvent` method, so plug-in authors will need to consider any thread synchronization issues arising from use of shared data structures.

Any transport plug-in called by a Java codec plug-in must also be written in Java.

Communication with other layers

Sending downstream messages received from a transport plug-in to the Semantic Mapper

When a transport plug-in configured to work with the event codec receives a messages from its external message source, it will pass it on to the codec plug-in by calling the `sendTransportEvent` method (as defined above). It is then up to the codec plug-in to decode the message from whatever custom format is agreed between the transport and codec plug-ins into a standard normalized event that can be passed on to the Semantic Mapper.

When the message has been decoded it should be sent to the Semantic Mapper using its `sendNormalisedEvent` method:

```
/**
  * Called by the event codec to send a decoded event to the Semantic Mapper.
  *
  * @param event A normalized event to be sent to the semantic mapper
  * @param timestamps A TimestampSet representing the timestamps attached to
  * the event.
  *
  *  @throws SemanticMapperException Thrown by the Semantic Mapper if there
  * is a problem mapping the event
  */
public void sendNormalisedEvent(NormalisedEvent event,
      TimestampSet timestamps)
    throws SemanticMapperException;
```

For example, the implementation of the event codec's `sendTranasportEvent` could look something like this:

```
// (Assume there's an instance field: SemanticMapper semanticMapper)
// Decode message
NormalisedEvent normalisedEvent = myDecodeMessage(event);
// Send to Transport layer plugin
semanticMapper.sendNormalisedEvent(normalisedEvent, timestamps);
```

If an error occurs in the Semantic Mapper, a `SemanticMapperException` is thrown. Typically such exceptions do not need to be caught by the codec plug-in, unless the codec plug-in is able to somehow deal with the problem.

A `CodecException` should be thrown if there is an error decoding the normalized event.

Communication with other layers

Java codec exceptions

`CodecException` is the exception class that should be thrown by a codec plug-in whenever the one of its methods is called and an error prevents the method from successfully completing — for example, a message that cannot be encoded or decoded because it has an invalid format.

A `CodecException` object always has an associated `message`, which is a `String` explaining the problem (this may include information about another exception that caused the `CodecException` to be thrown). There is also a `code` field that specifies the kind of error that occurred; the possible codes are defined as constants in the `CodecException` class:

```
/**
 * Some unspecified internal error occurred
 */
public static final int INTERNALERROR = 1;
/**
 * Couldn't encode an incoming normalized event
 */
public static final int ENCODINGFAILURE = 2;
/**
 * Couldn't decode an incoming customer event
 */
public static final int DECODINGFAILURE = 3;
/**
 * Trouble sending encoded event to transport
 */
public static final int TRANSPORTFAILURE = 4;
/**
 * Trouble sending decoded event to Semantic Mapper
 */
public static final int MAPPINGFAILURE = 5;
/**
 * Codec was passed an invalid property set
 */
public static final int BADPROPERTIES = 6;
```

Like the `TransportException` object, `CodecException` defines a number of constructors, to make it easy to set up the exception's information quickly in different situations:

```
/**
 * Constructs a CodecException from a string message describing the
 * error, and assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error.
 */
public CodecException(String message) { ... }
/**
 * Constructs a CodecException from a string message describing the
 * error and a numeric code representing the class of error.
 *
 * @param message The cause of the error.
 * @param code One of the CodecException error codes.
 */
public CodecException(String message, int code) { ... }
/**
 * Constructs a CodecException from a string message describing the
 * error, and an exception object that is the cause of the error. It
 * assumes an error code of TRANSPORTFAILURE.
 *
 * @param message The cause of the error. This message will be suffixed
 * with the message of the 'cause' exception.
 * @param cause The exception object that caused the error.
 */
public CodecException(String message, Throwable cause) { ... }
/**
 * Constructs a CodecException from a string message describing the
 * error, a numeric code representing the class of error and an exception
 * object that is the cause of the error.
 *
 * @param message The cause of the error. This message will be suffixed
```

```
 * with the message of the 'cause' exception.
 * @param cause The exception object that caused the error.
 * @param code One of the CodecException error codes.
 */
public CodecException(String message, Throwable cause, int code) { ... }
```

The Codec Plug-in Development Specification for Java

Semantic Mapper exceptions

Codec plug-ins should never need to construct or throw `SemanticMapperException` objects, but they need to be able to catch them if they are thrown from the `SemanticMapper.sendNormalisedEvent` method when it is called by the event codec.

`SemanticMapperException` has exactly the same set of constructors as the `CodecException` class described above. The only significant different is the set of error codes, which for `SemanticMapperException` are as follows:

```
/**
 * Some unspecified internal error occurred
 */
public static final int INTERNALERROR = 1;
/**
 * Couldn't convert customer event to an Apama event
 */
public static final int MAPPINGFAILURE = 2;
/**
 * Couldn't queue converted event for injection into the Engine
 */
public static final int INJECTIONFAILURE = 3;
```

The Codec Plug-in Development Specification for Java

Logging

Please see "Logging from plug-ins in Java" on page 111 for information about how codec plug-ins should log error, status and debug information.

The Codec Plug-in Development Specification for Java

Working with normalized events

The function of a decoding codec plug-in is to convert incoming messages into a standard normalized event format that can be processed by the Semantic Mapper. Events sent upstream to an encoding codec plug-in are provided to the plug-in in this same format.

Normalized events are essentially dictionaries of name-value pairs, where the names and values are both character strings. Each name-value pair nominally represents the name and content of a single field from an event, but users of the data structure are free to invent custom naming schemes to represent more complex event structures. Names must be unique within a given event. Values may be empty or `null`.

Some examples of normalized event field values for different types are:

- `string  "a string"`
- `integer  "1"`
- `float  "2.0"`
- `decimal "100.0d"`
- `sequence<boolean>  "[true,false]"`
- `dictionary<float,integer> "{2.3:2,4.3:5}"`
- `SomeEvent  "SomeEvent(12)"`

Note: When assigning names to fields in normalized events, keep in mind that the `fields` and `transport` attributes for event mapping conditions and event mapping rules both use a list of fields delimited by spaces or commas. This means, for example that `<id fields="Exchange EX,foo" test="==" value="LSE"/>` will successfully match a field called "Exchange", "EX" or "foo", but *not* a field called "Exchange EX,foo". While fields with spaces or commas in their names may be included in a payload dictionary in upstream or downstream directions, they cannot be referenced directly in mapping or id rules.

To construct strings for the normalized event fields representing container types (dictionaries, sequences, or nested events), use the event parser/builder found in the `util5.2.jar` file, which is located in the Apama installation's `lib` directory The following examples show how to add a sequence and a dictionary to a normalized event (note the escape character " `\` " used in order to insert a quotation mark into a string).

```
List<String> list = new ArrayList<String>();
list.add("abc");
list.add("de\"f");
Map<String,String> map = new HashMap<String,String>();
map.put("key1", "value1");
map.put("key\"{}2", "value\"{}2");
final SequenceFieldType STRING_SEQUENCE_FIELD_TYPE =
    new SequenceFieldType(StringFieldType.TYPE);
final DictionaryFieldType STRING_DICT_FIELD_TYPE =
    new DictionaryFieldType(StringFieldType.TYPE, StringFieldType.TYPE);
NormalisedEvent event = new NormalisedEvent();
event.add("mySequenceField",
    STRING_SEQUENCE_FIELD_TYPE.format(list));
event.add("myDictionaryField", STRING_DICT_FIELD_TYPE.format(map));
```

The programming interface for constructing and using normalized events is made up of three Java classes: `NormalisedEvent`, `NormalisedEventIterator` and `NormalisedEventException`. `NormalisedEvent` is the most important part of the interface, and encapsulates the data and operations that can be performed on a single normalized event. Some of these operations return `NormalisedEventIterator` objects, which support the process of stepping through the name-value pairs in the normalized event. Any errors encountered result in instances of `NormalisedEventException` being thrown.

This section provides an overview of the capabilities of the two main classes. Please see the Javadoc documentation for full information on the normalized event interface.

The Codec Plug-in Development Specification for Java

The NormalisedEvent class

The `NormalisedEvent` class represents a single normalized event. The following methods are provided for examining and modifying the name-value pairs making up the event:

- `size` - Return the number of elements (name-value pairs) currently stored by the event.
- `empty` - Check whether the event is empty or not.
- `add` - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten. This returns an iterator into the events at the point where the new element was added.
- `addQuick` - Add a new name-value pair to the event. The given name and value will be copied into the event. If an element with the same name already exists, it will not be overwritten.
- `remove` - Remove the named element from the normalized event.
- `removeAll` - Remove all elements from the normalized event. The `empty` function will return `true` after this function has been called.
- `replace` - Change the value of a named element in the normalized event. If an element with the given name already exists, its value will be replaced with a copy of the given value. Otherwise, a new element is created just as though `addQuick` had been called.
- `exists` - Check whether a given element exists in the normalized event.
- `find` - Search for a named element in the normalized event. Returns an iterator into the event at the point where the element was located.
- `findValue` - Search for a named element in the normalized event and return its value.
- `findValueAndRemove` - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event. Returns `null` if the specified element does not exist or has value `null`.
- `findValueAndRemove2` - Search for a named element in the normalized event and return its value. If found, the element will also be removed from the event. Unlike `findValueAndRemove` it throws an exception if the value is not found.
- `first` - Return an iterator pointing to the first element of the normalized event. Successive calls to the `next` function of the returned iterator will allow you to visit all the elements of the event.
- `last` - Return an iterator pointing to the last element of the normalized event. Successive calls to the `back` function of the returned iterator will allow you to visit all the elements of the event.
- `toString` - Return a printable string representation of the normalized event.

Threading note: Normalised events are not thread-safe. If your code will be accessing the same normalized event object (or associated iterators) from multiple threads, you must implement your own thread synchronization to prevent concurrent modification.

A public zero-argument constructor is provided for creation of new (initially empty) `NormalisedEvent` objects.

Working with normalized events

The NormalisedEventIterator class

Several of the `NormalisedEvent` methods return an instance of the `NormalisedEventIterator` class, which provides a way to step though the name-value pairs making up the normalized event, forwards or backwards.

The following public methods are provided:

- `valid` - Check whether the iterator points to a valid element of the normalized event. Typically used as part of the loop condition when iterating over the contents of an event.
- `key` - Return the key (name) associated with the current event element pointed to by the iterator.
- `value` - Return the value associated with the current event element pointed to by the iterator.
- `next` - Move the iterator to the next element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `true`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.
- `back` - Move the iterator to the previous element of the underlying normalized event instance. The iterator must be in a valid state (such that the `valid` function would return `true`) before this function is called. Note that the order in which elements is returned is not necessarily the same as the order in which they were added. The order may change as elements are added to or removed from the underlying event.

There is no public constructor; iterators are created and returned only by `NormalisedEvent` methods.

Please see the Javadoc documentationfor full information about the classes introduced in this section.

Working with normalized events

Java Codec Example

As part of the IAF distribution Apama includes the `JStringCodec` codec layer plug-in, in the `samples\iaf_plugin\java\simple\src` directory.

The `JStringCodec` plug-in converts between normalized events and a text string representation that can be customized using plug-in configuration properties.

Developers are encouraged to explore this sample to see what a typical codec plug-in implementation looks like.

See "IAF samples" on page 45 for more information about this sample. The section "The String Codec plug-in" on page 141 describes how the `JStringCodec` plug-in can be used in practice.

Java Codec Plug-in Development

Getting started with Java codec layer plug-in development

Your distribution includes a complete 'skeleton' implementation of a codec layer plug-in in order to make development of new plug-ins faster.

This is located in the `samples\iaf_plugin\java\skeleton\src` directory of the installation, in a file called `SkeletonCodec.java`. The `SkeletonCodec` class complies fully with the Codec Plug-in Development Specification, but contains none of the custom encoding and decoding functionality that would be present in a full codec plug-in.

The code contains `TODO:` comments indicating the main changes that need to be made to develop a useful plug-in. These include:

- Adding code to `sendTransportEvent` to decode a message received from the transport layer into a normalized event (if supported).
- Adding code to `sendNormalisedEvent` to encode a message received from the Semantic Mapper transport into a message that can be sent on by the transport layer (if supported).
- Adding code to validate and save any new plug-in properties that are to be supported, in `updateProperties`.
- Adding code to initialize and clean up resources associated with the plug-in's operation. This would usually be done in the `updateProperties` and `cleanup` methods.

Depending on your requirements, it may also be necessary to make changes to the other main methods – `addEventTransport`, `removeEventTransport`, `flushUpstream`, `flushDownstream`, `getStatus`, and the constructor.

The `skeleton` directory includes an Apache Ant build file called `build.xml` that provides a convenient way to build `.jar` files of compiled classes from plug-in source files, ready for use with the IAF.

Java Codec Plug-in Development

Chapter 8: Plug-in Support APIs for Java

This section describes other programming interfaces provided with the Apama software that may be useful in implementing transport layer and codec plug-ins for the IAF.

Logging from plug-ins in Java

This API provides a mechanism for recording status and error log messages from the IAF runtime and any plug-ins loaded within it. Plug-in developers are encouraged to make use of the logging API instead of custom logging solutions so that all the information may be logged together in the same standard format and log file(s) used by other plug-ins and the IAF runtime.

The logging API also allows control of logging verbosity, so that any messages below the configured logging level will not be written to the log. The logging level and file are initially set when an adapter first starts up – see "Logging configuration (optional)" on page 43 for more information about the logging configuration.

The Java logging API is based around the `Logger` class.

The recommended way of using the `Logger` class is to have a `private final com.apama.util.Logger` variable, and then create an instance in the transport or codec's constructor based on the plug-in name, such as the following:

```
private final Logger logger;
public MyTransport(String name, ...)
{   super(...);
   logger = Logger.getLogger(name);
}
```

The Logger class supports the following logging levels:

- `FORCE`
- `CRIT`
- `FATAL`
- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`

For each level, there are three main methods. For example, for logging at the DEBUG level, here are the three main methods:

- `logger.debug(String)` — Logs a message, if this log level is currently enabled.

- `logger.debug(String, Throwable)` — Logs the stack trace and message of a caught exception together with a high-level description of the problem. Apama strongly recommend logging exceptions like this to assist with debugging in the event of problems.
- `logger.debugEnabled()` — Determines whether messages at this log level are currently enabled (this depends on the current IAF log level, which may be changed dynamically). Apama strongly recommend checking this method's result (particularly for `DEBUG` messages) before logging messages where constructing the message string may be costly, for example:

```
if (logger.debugEnabled())
  logger.debug("A huge message was received, and the string
      representation of it is: "+thing.toString()+
      " and here is some other useful info: "+foo+", "+bar);
```

Note that there is no point using the *`Enabled()` methods if the log message is a simple string (or string plus exception), such as:

```
logger.debug("The operation completed with an error: ", exception);
```

To make it easier to diagnose any errors that may occur, Apama recommends one of the following methods to log the application's stack trace:

- `errorWithDebugStackTrace(java.lang.String msg, java.lang.Throwable ex)` — Logs the specified message at the `ERROR` level followed by the exception's message string, and then logs the exception's stack trace at the `DEBUG` level.
- `warnWithDebugStackTrace(java.lang.String msg, java.lang.Throwable ex)` — Logs the specified message at the `WARN` level followed by the exception's message string, and then logs the exception's stack trace at the `DEBUG` level.

See the `Javadoc` documentation for more information about the `Logger` class.

Plug-in Support APIs for Java

Using the latency framework

The latency framework API provides a way to measure adapter latency by attaching high-resolution timing data to events as they stream into, through, and out of the adapter. Developers can then use these events to compute upstream, downstream, and round-trip latency numbers, including latency across multiple adapters.

The `sendNormalisedEvent()` and `sendTransportEvent()` methods contain a `TimestampSet` parameter that carries the microsecond-accurate timestamps that can be used to compute the desired statistics.

HTML Javadoc documentation for `com.apama.util.TimestampSet` and `com.apama.util.TimestampConfig` classes is provided as part of the Apama documentation set. The `Javadoc` is located in the Apama installation's `doc\javadoc` directory.

Plug-in Support APIs for Java

Java timestamp

A timestamp is an index-value pair. The index represents the point in the event processing chain at which the timestamp was recorded, for example "upstream entry to semantic mapper" and the value is a floating point number representing the time. The `TimestampSet` class defines a set of standard

indexes but a custom plug-in can define additional indexes for even finer-grained measurements. When you add a custom index definition, be sure to preserve the correct order, for example, an index denoting an “entry” point should be less than an one denoting an “exit” point from that component.

Timestamps are relative measurements and are meant to be compared only to other timestamps in the same or similar processes on the same computer.

Using the latency framework

Java timestamp set

A timestamp set is the collection of timestamps that are associated with an event. The latency framework API provides functions that developers can use to add, inspect, and remove timestamps from an event’s timestamp set.

The timestamp set is represented as a `dictionary` of integer-float pairs, where the integer index refers to the location at which the timestamp was added and the floating-point time gives the time at which an event was there.

Using the latency framework

Java timestamp configuration object

The constructors and `updateProperties()` methods for transport and codec plugins take this additional argument: `TimestampConfig`.

A timestamp configuration object contains a set of fields that a plug-in can use to decide whether to record and/or log timestamp information. The fields in the object are:

- `recordUpstream` — If true, the plug-in should record timestamps for all upstream events it processes, and pass these along to the upstream component, if any.
- `recordDownstream` — If true, the plug-in should record timestamps for all downstream events it processes, and pass these along to the downstream component, if any.
- `logUpstream` — If true, the plug-in should log the latency for all upstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable upstream timestamp recording if upstream logging is enabled.
- `logDownstream` — If true, the plug-in should log the latency for all downstream events it processes, at the logging level given by the `logLevel` member. A plug-in may implicitly enable downstream timestamp recording if downstream logging is enabled.
- `logRoundtrip` — If true, the plug-in should log the “round trip” latency for all events it processes in either direction, if possible. At its simplest, the round trip latency can just be the difference between the largest and smallest timestamps passed to the plug-in, or an individual plug-in may choose to present some more plug-in-specific latency number. As with the other logging options, the logging level given by the `logLevel` member should be used.
- `logLevel` — The logging verbosity level to use if any of the timestamp logging options are enabled.

Using the latency framework

Java latency framework API

The Java interface for the latency framework is declared in the header file `com.apama.util.TimestampSet` class.

The functions of interest are the following.

Table 2. Java latency framework API

`void addNow(java.lang.Integer index)`	Add a new name-time pair to the event.
`void addTime(java.lang.Integer index,` `java.lang.Double theTime)`	Add a new index-time pair to the timestamp.
`void clear()`	Removes all mappings from this map.
`boolean containsKey(java.lang.Object index)`	Returns true if this map contains the specified key
`boolean containsValue(java.lang.Object time)`	Returns true if this map maps one or more names to the specified time.
`java.util.Set<java.util.Map.Entry<java.lang.Integer, java.lang.Double>> entrySet()`	
`java.lang.Double findTime(java.lang.Integer index)`	Search for a named element in the timestamp set and return its time.
`java.lang.Double findTimeAndRemove( java.lang.Integer index)`	Search for a named element in the timestamp set and return its time.
`java.lang.Double get(java.lang.Object index)`	Returns the time to which the specified name is mapped, or null if the map contains no mapping for this index.
`static double getMicroTime()`	Get the current microsecond-accurate relative timestamp.
`boolean isEmpty()`	Returns true if this map contains no index-time mappings.
`java.util.Iterator<java.util.Map.Entry`	Returns a standard Java Iterator over the contents of the

`<java.lang.Integer,java.lang.Double>> iterator()`	TimestampSet using Map.Entry objects.
`java.util.Set<java.lang.Integer> keySet()`	
`java.lang.Double put(java.lang.Integer index, java.lang.Double time)`	Adds or replaces the specified (index,time) pair in the underlying map
`void putAll(java.util.Map<? extends java.lang.Integer,? extends java.lang.Double> m)`	Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the indices currently in the specified map.
`java.lang.Double remove(java.lang.Object index)`	Removes the mapping for this index from this map if present.
`void replace(java.lang.Integer index, java.lang.Double newTime)`	Change the time of a named element in the timestamp set.
`void replaceWithNow(java.lang.Integer index)`	Change the time of a named element in the timestamp set.
`int size()`	Get the number of elements (name-time pairs) currently stored by the event.
`java.lang.String toString()`	Return a printable string representation of the timestamp set.
`java.util.Collection<java.lang.Double> values()`	Returns a collection view of the times contained in this map.

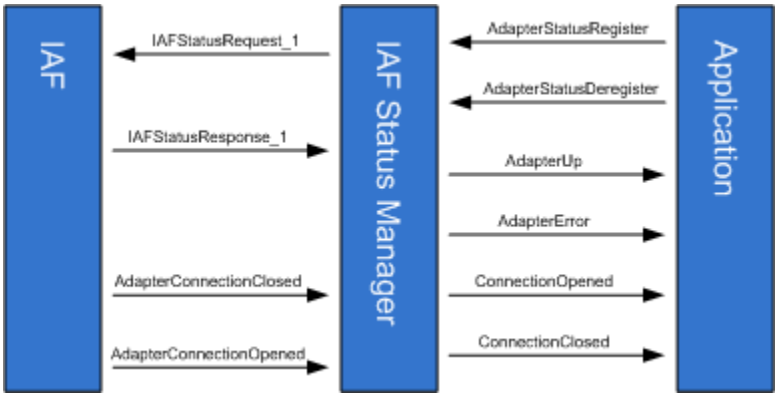
Using the latency framework

Chapter 9: Monitoring Adapter Status

Status information is available between the correlator and an adapter. When developing an IAF adapter, the adapter author can provide the ability to make use of this status information. Basic information, such as whether an adapter is up or down, is available using a standard Apama monitor. Other information, such as the number of connections an adapter has, can be provided by using the `getStatus()` method in an adapter's transport and codec. Optionally, adapter authors can also add code to the adapter's service monitors to send and receive specific status information that application developers can then use when they write Apama applications that connect to the adapters.

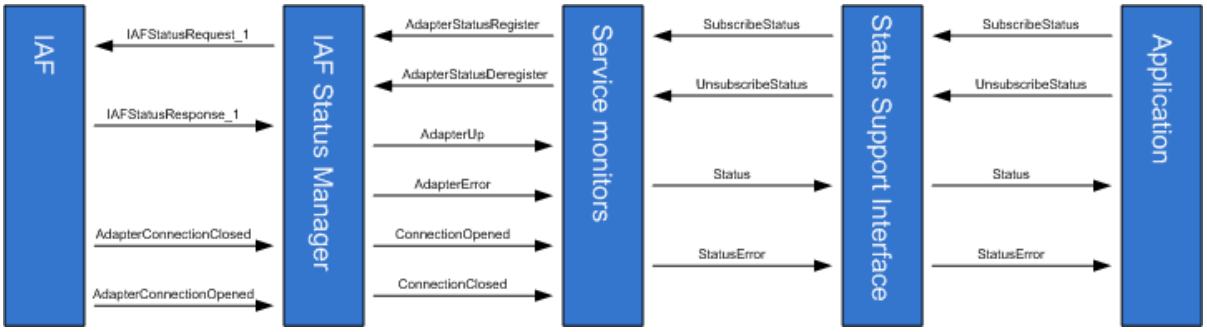
Apama provides the following two mechanisms for handling IAF Adapter status information:

- IAFStatusManager — The IAFStatusManager manages the connection status and other status information from the adapter to the correlator. In order to retrieve adapter status information, the IAFStatusManager needs to be injected into the correlator and the adapter author needs to add a small amount of code to the adapter. Application authors can then make use of status information available from the IAFStatusManager.



  For information on using the IAFStatusManager, see "IAFStatusManager" on page 117

- StatusSupport — StatusSupport is a generic interface (or contract) between an Apama application and an adapter's service monitors. This interface provides a way to provide an application with a similar view of all the status information available from multiple adapters. In order to use the StatusSupport interface, an adapter author writes code in the adapter's service monitors that send or receive specific StatusSupport events. In turn, the application author writes code to implement the desired behavior for handling the StatusSupport events.



Using the StatusSupport interface is optional. For more information on using this interface, see "StatusSupport" on page 127.

IAFStatusManager

The IAFStatusManager translates events from the adapter into simple status events for applications to consume. The monitor, `IAFStatusManager.mon` is found in the Apama installation's `adapters\monitors` directory. In order to use the monitor:

- The adapter author is required to return information about the adapter's open connections in the adapter's `getStatus` method, which is called every few seconds when the IAFStatusManager service monitor polls the IAF for status. Adapters written in Java must return an `ExtendedTransportStatus` or `ExtendedCodecStatus` object from `getStatus()`; adapters written in C++ must return `AP_EventTransportStatus` or `AP_ExtendedCodecStatus`.
- The adapter may optionally also send notifications about a connection as soon as it is opened or closed, by sending a normalized event representation of the `AdapterConnectionOpened` or `AdapterConnectionClosed`events to the correlator. This simply allows the correlator to find out about connectivity change more quickly than is the case if it needs to wait for the next status poll.

The IAFStatusManager has the following interfaces:

- An *application* interface to communicate with the consumers of the adapter status information — usually adapter service monitors.
- An IAF *adapter* interface is optional and can be used by adapter authors to issue connection notifications.

Application interface

The `IAFStatusManager.mon` file defines the event interface between it and a consumer of an adapter's status information, which is usually an adapter's service monitor. The application interface can be used to communicate status information to both the adapter's service monitors as well as Apama applications. The application interface events are either input events or output events. Input events are sent from a consumer of adapter status information to the IAFStatusManager. Output events are sent from the IAFStatusManager to a consumer of adapter status information.

Input events

The IAFStatusManager is a subscription based interface. This means that a consumer of adapter status information, such as an application service monitor, needs to send the input events `AdapterStatusRegister` and `AdapterStatusDeregister` events to register or deregister as a consumer for adapter status information. Once a subscription is made to the IAFStatusManager, the IAFStatusManager periodically receives information from the adapter and begins sending status information to the registered consumer in the form of output events — see "Output events" on page 119.

The IAFStatusManager defines the following input events:

- `AdapterStatusRegister` — An event sent by a client that is interested in receiving status events from the specified codec and transports. The fields of this event uniquely identify a subscription.

  ```
  event AdapterStatusRegister {
    string adapterName;
    string codec;
    string transport;
    string codecVersion;
    string transportVersion;
    string configVersion;
    string channel;
  }
  ```

  The fields of the `AdapterStatusRegister` event are:

  - `adapterName` — An identifier that will be used to refer to this transport and codec pair when registering, deregistering or monitoring adapter status.
  - `codec` — The name of the adapter's codec as it is specified in the adapter configuration file.
  - `transport` — The name of the adapter's transport as it is specified in the adapter configuration file.
  - `codecVersion` — The codec version that the client depends on, or an empty string ("") if the client can use any version of the codec.
  - `transportVersion` — The transport version that the client depends on, or an empty string ("") if the client can use any version of the transport.
  - `configVersion` — The value of this field can be empty, but if a value is specified it must agree with the `CONFIG_VERSION` key returned in the `ExtendedTransportStatus` or `ExtendedCodecStatus` object (for Java) or in the `statusDictionary` of an `AP_EventTransportStatus` or `AP_EventCodecStatus` (for C++).
  - `channel` — The name of the channel the IAF adapter is receiving events from.

- `AdapterStatusDeregister` — An event sent by a client to unregister its subsciption for status events.

  ```
  event AdapterStatusDeregister {
          string adapterName;
        }
  ```

  The `adapterName` field is the identifier used to refer to this transport and codec pair when deregistering

  .

Output events

Once a consumer of status information (such as an application service monitor) is registered with the IAFStatusManager, it begins to receive status information in the form of IAFStatusManager output events. Output events include connection information, adapter availability, and any custom information put into the dictionary by the transport or codec. For more information about adding custom information, see "Connections and other custom properties" on page 123.

The IAFStatusManager defines the following output events:

- `AdapterUp` — An event routed from the IAF Signaling Service to notify that the adapter is up.

  ```
  event AdapterUp {
    string adapterName;
    float latency;
    dictionary<string, string> codecStatus;
    dictionary<string, string> transportStatus;
  }
  ```

  The fields of the `AdapterUp` event are:

  - `adapterName` — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
  - `latency` — This is the difference (in seconds) between the time a request for status is sent from the IAFStatusManager to the IAF and the time the correlator receives the response.
  - `codecStatus` — Contains information about the adatper's codec. Standard keys are `VERSION`, `CONFIG_VERSION`, and `CONNECTION` (or in the case of multiple connections, keys of the form `CONNECTION_`*`connectionName`*).
  - `transportStatus` — Contains information about the adatper's transport. Standard keys are `VERSION`, `CONFIG_VERSION`, and `CONNECTION` (or in the case of multiple connections, keys of the form `CONNECTION_`*`connectionName`*).

- `AdapterError` — An event routed from the IAF Signaling Service to tell that there was an error getting the status of the adapter.

  ```
  event AdapterError {
         string adapterName;
         string description;
       }
  ```

  The fields of the AdapterError event are:

  - `adapterName` — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
  - `description` — A free form string describing the problem.

- `ConnectionClosed` — An event routed when the IAF Signaling Service discovers that the adapter's connection to an external service is closed.

  ```
  event ConnectionClosed {
           string adapterName;
           string connectionName;
           string connectionGeneration;
         }
  ```

  The fields of the `ConnectionClosed` event are:

- `adapterName` — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
- `connectionName` — This is a unique identifier for the connection. If the adapter manages more than one connection, this will be the connection name returned by the adapter as a `CONNECTION_`*`connectionName`*`getStatus` key (but without the `CONNECTION_` prefix), or "" if the adapter only manages one connection. The *`connectionName`* is often a number but could be a string. One event will be sent for each connection the adapter manages.
- `connectionGeneration` — This identifies a successful connection attempt. If the connection fails, and then is successfully connected again, this will change. The `connectionGeneration` is often a number that is initialized with a timestamp when the adapter is created, then incremented every time it reconnects.

- `ConnectionOpened` — An event routed when the IAF Signaling Service discovers that the adapter's connection to an external service is established.

```
event ConnectionOpened {
              string adapterName;
              string connectionName;
              string connectionGeneration;
            }
```

The fields of the `ConnectionOpened` event are:

- `adapterName` — This is the identifier used to refer to this transport and codec pair when monitoring adapter status.
- `connectionName` — This is a unique identifier for the connection. If the adapter manages more than one connection, this will be the connection name returned by the adapter as a `CONNECTION_`*`connectionName`*`getStatus` key (but without the `CONNECTION_` prefix), or "" if the adapter only manages one connection. The *`connectionName`* is often a number but could be a string. One event will be sent for each connection the adapter manages.
- `connectionGeneration` — This identifies a successful connection attempt. If the connection fails, and then is successfully connected again, this will change. The connectionGeneration is often a number that is initialized with a timestamp when the adapter is created, then incremented every time it reconnects.

Returning information from the getStatus method

The adapter's transport and codec `getStatus` methods periodically update status information. The transport and codec send this information to the IAFStatusManager. To take advantage of the IAFStatusManager for an adapter written in Java, the adapter author should implement the `getStatus` method so that it returns an `ExtendedTransportStatus` or `ExtendedCodecStatus` object. These objects include a `Properties` parameter, statusInfo, which contains custom information about the adapter.

For adapters written in C or C++, the adapter author should implement the `getStatus` function to include the `statusDictionary` in an `AP_EventTransportStatus` or `AP_EventCodecStatus` structure.

The IAFStatusManager then forwards the information to registered consumers of that transport or codec's status information in the form of a dictionary added to the `AdapterUp` event.

ExtendedTransportStatus

The `ExtendedTransportStatus` object is defined as follows:

```
public ExtendedTransportStatus(java.lang.String status,
                   long totalReceived,
                   long totalSent),
                   java.util.Properties statusInfo)
```

The object's parameters are:

- `status` - A string containing a transport-specific status message. Strings longer than 1024 characters will be truncated.
- `totalReceived` - The total number of downstream events received since the IAF was run.
- `totalSent` - The total number of upstream events sent since the IAF was run.
- `statusInfo` - Any additional status information about this transport. The standard `statusInfo` keys are:
  - `VERSION=transport_version_string`
  - `CONFIG_VERSION=config_version_string`
  - `CONNECTION=connectionGeneration` (if the adapter manages only one connection)

    or
  - `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

    For more information on specifying the `CONNECTION` or `CONNECTION_connectionId` key, see "Asynchronously notifying IAFStatusManager of connection changes" on page 124.

ExtendedCodecStatus

The `ExtendedCodecStatus` object is defined as follows:

```
public CodecStatus(java.lang.String status,
                   long totalDecoded,
                   long totalEncoded),
                   java.util.Properties statusInfo
```

The object's parameters are:

- `status` - A string containing a codec-specific status message. Strings longer than 1024 characters will be truncated.
- `totalDecoded` - The number of events decoded
- `totalEncoded` - The number of events encoded
- `statusInfo` - Any additional status information bout this codec. Standard statusInfo keys are:
  - `VERSION=codec_version_string`
  - `CONFIG_VERSION=config_version_string`
  - `CONNECTION=connectionGeneration` (if the adapter manages only one connection),

    or
  - `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections

AP_EventTransportStatus

The `AP_EventTransportStatus` object is defined as:

```
typedef struct {
  AP_char8* status;
```

```
  AP_uint64 totalReceived;
  AP_uint64 totalSent;
  AP_NormalisedEvent* statusDictionary;
} AP_EventTransportStatus;
```

The object's parameters are:

- `status` - A free-form text string containing a transport-specific status message. Strings longer than 1024 characters will be truncated.
- `totalReceived` - The total number of downstream events received since the IAF was run.
- `totalSent` - The total number of upstream events sent since the IAF was run.
- `statusDictionary` - Any additional status information about this transport. The standard `statusDictionary` keys are:
  - `VERSION=transport_version_string`
  - `CONFIG_VERSION=config_version_string`
  - `CONNECTION=connectionGeneration` (if the adapter manages only one connection)

    or
  - `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

AP_EventCodecStatus

The `AP_EventCodecStatus` object s defined as:

```
typedef struct {
  AP_char8* status;
  AP_uint64 totalDecoded;
  AP_uint64 totalEncoded;
  AP_NormalisedEvent* statusDictionary;
} AP_EventCodecStatus;
```

The object's parameters are:

- `status` - A free-form text string containing a codec-specific status message. Strings longer than 1024 characters will be truncated.
- `totalDecoded` - The number of events decoded.
- `totalEncoded` - The number of events encoded.
- `statusDictionary` - Any additional status information about this codec. Standard `statusDictionary` keys are:
  - `VERSION=codec_version_string`
  - `CONFIG_VERSION=config_version_string`
  - `CONNECTION=connectionGeneration` (if the adapter manages only one connection),

    or
  - `CONNECTION_connectionId=connectionGeneration` (if the adapter manages multiple connections)

Example

In the following example, the custom status information for `VERSION` and `CONNECTION` is included in the information returned by the `getStatus` method:

```
public static final String TRANSPORT_VERSION="1";
```

```
protected long connGeneration;
...
public TransportStatus getStatus()
{
   Properties properties=new Properties();
   properties.setProperty("VERSION", TRANSPORT_VERSION);
   if(market!=null)
   {
     properties.setProperty("CONNECTION",
     String.valueOf(connGeneration));
   }
   return new ExtendedTransportStatus("OK", numReceived, numSent, properties);
}
```

For more information on specifying the `CONNECTION` property, see "Asynchronously notifying IAFStatusManager of connection changes" on page 124.

Monitoring Adapter Status

Connections and other custom properties

An adapter may deal with no connections, a single connection, or an arbitrary number of connections (for example, if it is a server socket that accepts clients connecting to it); an adapter may also deal with a set number of connections. In any case, an identifier needs to be assigned to each connection. A connection may be broken and then reconnected, with either the same or different identifier. It is useful to be able to detect a connection that has been dropped and then reconnected even if it has the same identifier. To facilitate this, a “generation” identifier can be associated with each connection identifier. While typically this generation identifier will be a number that is incremented, extra information may be contained in it.

Monitors can therefore detect when a connection has been reconnected; at this point any logon procedure needs to be repeated as the generation identifier has changed.

The state of all connections should be supplied in the `statusDictionary` field of the status struct in C/C ++, or the `statusInfo` field of the `ExtendedCodecStatus` or `ExtendedTransportStatus` in Java.

Along with any other custom information, the adapter author can include connection information here. This will be passed to the correlator in event form and the IAFStatusManager will automatically attempt to pull out connection information from this data structure. If there is a single connection, a key should be supplied called `CONNECTION`. The value will be the generation identifier, typically a number. If the generation identifier changes, the IAFStatusManager will assume the connection has been dropped and reestablished, and will send appropriate events to the consumer of the status events.

If there are multiple connections, a key for each one should be supplied in the form `CONNECTION_`<id> to distinguish the different connections. Each one will also have a generation identifier associated with it. The same rules apply with the generation identifier as with a single connection.

In either case, if the connection is up, the property should be included, and if the connection is down, the property should not be included. This allows monitors to recover the state of what connections are made after losing connection to the IAF, and to determine when connections are opened or closed by polling.

The following Java example shows a simple adapter that reports the status of a single connection.

```
private long connectionGeneration = System.currentTimeMillis();
public TransportStatus getStatus()
{  Properties properties = new Properties();
  properties.setProperty("VERSION", "MyTransport_v1.0");
```

```
  properties.put("CONFIG_VERSION", "1");

  if (connected)
  {
    properties.setProperty("CONNECTION",
      String.valueOf(connectionGeneration));
  }
  return new ExtendedTransportStatus("OK", totalReceived,
      totalSent, properties);
}
```

The following Java example demonstrates usage with multiple connections, iterating through a collection of `MyConnection` objects.

```
public TransportStatus getStatus()
{  Properties properties = new Properties();
  properties.put("VERSION", "MyTransport_v1.0");
  properties.put("CONFIG_VERSION", "1");
  for (MyConnection con : connections.values())
  {
    if (!con.isClosed())
    {
      properties.put("CONNECTION_" + con.getId(), con.getGeneration());
    }
  }
  return new ExtendedTransportStatus(statusMessage, totalReceived,
      totalSent, properties);
}
```

Asynchronously notifying IAFStatusManager of connection changes

In addition to returning status information in response to a poll from the IAFStatusManager, an adapter may also send out events asynchronously when a connection is opened or closed.

This is done by creating and sending a `NormalisedEvent` object from the transport or codec to the semantic mapper. The `NormalisedEvent` object has special fields that allow for automatic mapping to an Apama event type— either `AdapterConnectionOpened` or `AdapterConnectionClosed`. The `AdapterConnectionOpened` and `AdapterConnectionClosed` events are then sent through the correlator to the IAFStatusManager.

The `NormalisedEvent` must have the following fields:

Table 3. NormalizedEvent fields

Field name	Field value
`AdapterConnectionOpenEvent` or `AdapterConnectionClosedEvent`	No value (empty string). This will either represent a connection opened or connection closed and be translated into `AdapterConnectionOpened` or `AdapterConnectionClosed` events respectively for the `IAFStatusManager` to consume.
`codecName`	Name of codec
`transportName`	Name of transport

Field name	Field value
`connectionName`	No value (empty string) if there is only one connection. If there is more than one connection, this should contain `CONNECTION_`*<id>* and one event should be sent for every connection the adapter is concerned with.
`connectionGeneration`	Connection generation identifier. This identifies a successful connection attempt with a connectionName. If the connection fails, then is successfully connected again, this should change. This is usually a number that is incremented.

This connection information should have a direct correlation to the connection information sent in the `getStatus` implementation. Note that if the transport deals with only a single connection at a time, the `connectionName` will be `""` (the empty string) instead of `CONNECTION`, as it is in the `getStatus` implementation.

The following is an example in Java of sending a `NormalisedEvent` that provides status information.

```
protected void sendAdapterConnectionStatusChangeNotification(boolean open,
       String reason, TimestampSet tss)
{
  if(decoder==null) return;
  NormalisedEvent ne=new NormalisedEvent();
  ne.add("codecName", codecName);
  ne.add("transportName", transportName);
  if(reason==null)
  {
    reason="";
  }
  if(open)
  {
    ne.add("AdapterConnectionOpenEvent", reason);
  }
  else
  {
    ne.add("AdapterConnectionClosedEvent", reason);
  }
  ne.add("connectionGeneration", String.valueOf(connGeneration));
  ne.add("connectionName", "");
  try
  {
    decoder.sendTransportEvent(ne, tss);
  }
  catch (CodecException e)
  {
    logger.error("Could not send message due to Codec error: ", e);
  }
  catch (SemanticMapperException e)
  {
    logger.error("Could not send message due to Semantic Mapper
        error: ", e);
  }
}
```

Note: When using these events, the (J)NullCodec must be used, unless you write a codec that handles these and passes them on to the correlator. For example, the XMLCodec by default will not forward these events to the semantic mapper. If you want to use the XMLCodec, you need to use the (J)NullCodec as the codec to send these particular events.

For more information on the implicit rules that the semantic mapper uses to automatically map the objects to `AdapterConnectionOpened` and `AdapterConnectionClosed` events, see "Mapping AdapterConnectionClosed and AdapterConnectionOpened events" on page 126.

Monitoring Adapter Status

Mapping AdapterConnectionClosed and AdapterConnectionOpened events

As described in "Asynchronously notifying IAFStatusManager of connection changes" on page 124, the semantic mapper contains implicit rules to map `NormalisedEvent` objects that contain special fields to `AdapterConnectionClosed` and `AdapterConnectionOpened` events. This means you do not need to add mapping rules to your adapter's configuration file. These implicit rules are:

```
<event name="AdapterConnectionClosed"
       package="com.apama.adapters"
       direction="downstream"
       breakDownstream="false">
  <id-rules>
    <downstream>
      <id fields="codecName,
                  transportName,
                  connectionName,
                  connectionGeneration"
          test="exists"/>
      <id fields="AdapterConnectionClosedEvent"
          test="exists"/>
    </downstream>
  </id-rules>
  <mapping-rules>
    <map apama="codecName"
         transport="codecName"
         type="string" default=""/>
    <map apama="transportName"
         transport="transportName"
         type="string" default=""/>
    <map apama="connectionName"
         transport="connectionName"
         type="string" default=""/>
    <map apama="connectionGeneration"
         transport="connectionGeneration"
         type="string" default=""/>
  </mapping-rules>
</event>
<event name="AdapterConnectionOpened"
       package="com.apama.adapters"
       direction="downstream"
       breakDownstream="false">
  <id-rules>
    <downstream>
      <id fields="codecName,
                  transportName,
                  connectionName,
                  connectionGeneration"
          test="exists"/>
      <id fields="AdapterConnectionOpenEvent"
          test="exists"/>
    </downstream>
  </id-rules>
  <mapping-rules>
    <map apama="codecName"
         transport="codecName"
         type="string" default=""/>
```

```
    <map apama="transportName"
         transport="transportName"
         type="string" default=""/>
    <map apama="connectionName"
         transport="connectionName"
         type="string" default=""/>
    <map apama="connectionGeneration"
         transport="connectionGeneration"
         type="string" default=""/>
  </mapping-rules>
</event>
```

StatusSupport

Consumers of the IAFStatusManager events are typically the adapter service monitors. In some cases it may desirable for an Apama application to have a more generic view of components and their status information so that getting status information will look the same across all components in a system, regardless of component type. For example, in addition to the information provided by the IAFStatusManager such as whether the adapter is up or connected, it may be useful to provide confirmation that the adapter has successfully logged in to an external system or a message that the external system is down.

Apama provides an interface called the StatusSupport event interface to help define this. It allows applications (EPL code or scenarios and blocks) to see state from service monitors such as the adapter service monitors. In order to implement this behavior, adapter authors add code to the adapter service monitors to handle the various StatusSupport events. Developers of Apama applications can then add code to take appropriate actions for the StatusSupport events to their applications that use the adapters. In this way, an application can act as a "health monitor" and be notified when a component is down or what its status is at any given time.

The StatusSupport events are described "StatusSupport events" on page 127.

The StatusSupport event interface is a subscription based interface, so consumers of this information will need to subscribe before receiving status information. The adapter service monitors need to reference count the status subscribers, so they do not stop sending status information if there are any interested consumers left. A subscription will only be removed when the call to remove the last one is made.

StatusSupport events

The StatusSupport event interface is defined in the `StatusSupport.mon` file, which is found in the `monitors` directory of the Apama installation (note, this is not the same directory as `adapters\monitors`).

All of the StatusSupport events contain the following fields:

- serviceID — The serviceID to subscribe to, a blank in this field targets all services
- `object` — The object to request status of - this may include:
  - "Connection" - whether connected or not
  - "MarketState" - a market may be "Open", "Closed", or other states
- `subServiceID` — The subService ID to subscribe to. Some services may expose several services. The interpretation of this string is adapter-specific.

- `connection` — The connection to subscribe to.Some services may expose several services. The interpretation of this string is adapter-specific.

The StatusSupport interface defines the following events:

- `SubscribeStatus` — This event is sent to the service monitor to subscribe to status.

```
event SubscribeStatus {
  string serviceID;
  string object;
  string subServiceID;
  string connection;
}
```

- `UnsubscribeStatus` —

```
event UnsubscribeStatus {
  string serviceID;
  string object;
  string subServiceID;
  string connection;
}
```

- `Status` —

```
event Status {
  string serviceID;
  string object;
  string subServiceID;
  string connection;
  string description;
  sequence<string> summaries;
  boolean available;
  wildcard dictionary <string, string> extraParams;
}
```

The additional fields for the `Status` event type are:

- `description` — A free-form text string giving a description of the status.
- `summaries` — The status of the object requested. This will be a well recognized sequence of words - for example, a financial market's "MarketState" may be "Open", "Closed", "PreOpen", etc. A Connection may be "Connected", "Disconnected", "Disconnected LoginFailed", "Disconnected TimedOut", etc. There should be at least one entry in the sequence.
- `available` — True if the object is "available" - the exact meaning is adapter specific; for example, connected, open for general orders, etc.
- `extraParams` — Extra parameters that do not map into any of the above. Convention is that keys are in TitleCase. e.g. "Username", "CloseTime", etc.

A `Status` event does not denote a change of state, merely what the current state is — in particular, one will be sent out after every `SubscribeStatus` request.

Any adapter specific information that the application needs to supply or be supplied can be passed in the `extraParams` dictionary — these are free-form (though there are conventions on the keys, see below).

- `StatusError` —

```
event StatusError {
  string serviceID;
  string object;
  string subServiceID;
  string connection;
  string description;
  boolean failed;
```

```
}
```

The additional field for this event type is:

- `failed` — Whether the subscription has been terminated. Any subscribers will need to send a new SubscribeStatus request after this.

Note that the purpose of the `StatusError` event is to report a problem in the delivery of status information, not to report an "error" status. A `StatusError` should be sent when the service is unable to deliver status for some reason. For example, reports on the status of an adapter transport's connection to a downstream server cannot be sent if the correlator has lost its connection to the adapter — in this case the service would be justified in sending a `StatusError` event for the downstream connection status. However, in the same situation the service should continue to send normal `Status` events for the correlator-adapter connection status, as this status is known. The `available` flag in these Status events would of course be set to false to indicate that the connection is down.

If the `failed` flag in a `StatusError` event is true, this indicates that the failure in status reporting is permanent and any active status subscriptions will have been cancelled and receivers will need to re-subscribe if they wish to receive further status updates from the service. If the `failed` flag is false, the failure is temporary and receivers should assume that the flow of Status events will resume automatically at some point.

Chapter 10: Out of and Connection Notifications

When a sender and receiver component, such as a correlator, connects to or disconnects from the Integration Adapter Framework (IAF), the IAF automatically sends *out of band* notification events to adapter transports. Out of band notifications are events that are automatically sent to all public contexts in a correlator whenever any component (an IAF adapter, dashboard, another correlator, or a client built using the Apama SDKs) connects or disconnects from the correlator. These out of band events, which are defined in the `com.apama.oob` package, are:

- `ReceiverConnected`
- `SenderConnected`
- `ReceiverDisconnected`
- `SenderDisconnected`

The `ReceiverConnected` and `SenderConnected` events contain the name of the component that is connecting. When correlators and IAF adapters send a notification event, the format of the string that contains the component name is as follows:

"*name* (on port *port_number*)"

The *name* is the name that was specified when the component was started. For correlators and IAF adapters, you can specify a name with the `--name` option when you start the component. The name defaults to `correlator` or `iaf` according to the type of component. The *port_number* is the port that the connecting receiver or sender is running on.

Out of band events make it possible for a developer of an adapter to add appropriate actions for the adapter to take when it receives notice that a component has connected or disconnected. For example, an adapter can cancel outstanding orders or send a notification to an external system. In order to make use of the out of band events, adapters need to provide suitable mapping in the adapter configuration file. Adapters are also free to ignore these events.

For general information about using out of band notifications, see "Out of band connection notifications" in *Developing Apama Applications in EPL*.

Mapping example

Out of band events will only be received by codecs and transports if the semantic mapper is configured to allow them through. The semantic mapper should be configured as for any other set of events which it may wish to pass down. For more information on creating semantic mapping rules, see "The <event> mapping rules" on page 35.

For example:

```
<event package="com.apama.oob" name="ReceiverDisconnected"
       direction="upstream" encoder="$CODEC$" inject="false">
  <id-rules>
```

```
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandReceiverDisconnected" transport="_name" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
<event package="com.apama.oob" name="ReceiverConnected"
       direction="upstream" encoder="$CODEC$" inject="false">
  <id-rules>
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandReceiverConnected" transport="_name" />
    <map apama="name" transport="appname" default="" type="string" />
    <map apama="host" transport="address" default="" type="string" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
<event package="com.apama.oob" name="SenderDisconnected"
       direction="upstream" encoder="$CODEC$" inject="false">
  <id-rules>
    <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandSenderDisconnected" transport="_name" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
<event package="com.apama.oob" name="SenderConnected" direction="upstream"
       encoder="$CODEC$" inject="false">
  <id-rules>
  <upstream />
  </id-rules>
  <mapping-rules>
    <map type="string" default="OutOfBandSenderConnected" transport="_name" />
    <map apama="name" transport="appname" default="" type="string" />
    <map apama="host" transport="address" default="" type="string" />
    <map apama="physicalId" transport="physicalId" default="" type="string" />
    <map apama="logicalId" transport="logicalId" default="" type="string" />
  </mapping-rules>
</event>
```

The events are transmitted to signify the following events:

- `ReceiverConnected` — an external receiver has connected; the IAF can now send events to it.
- `ReceiverDisconnected` — an external receiver has disconnected; events will not be sent to this external receiver until it reconnects.
- `SenderConnected` — an external sender has connected. This external sender may send events following this event.
- `SenderDisconnected` — an external sender has disconnected. No more events will be received from this sender until a new `SenderConnected` message event is received.

However, adapters can make use of a disconnect message to not transmit events until such time as a connect occurs. For example, an adapter can coalesce events or tell external system to stop sending. Note that if multiple senders and receivers are connected and disconnected, the adapter will need to keep track of which one is connected.

Out of and Connection Notifications

Ordering of out of band notifications

The following guidelines describe when out of band connection and disconnection messages are received, and how this interacts with the framework provided to IAF adapters:

Transports and codecs will not be sent events until after their start function has been called and completed. Transports should not start generating events until their start function has been called. The first event that is delivered after the start function is called will be a `SenderConnected` or `ReceiverConnected` event, if the semantic mapper is configured to pass them through. An adapter will always receive the `SenderConnected` before it begins to receive any other events, but the ordering of the `ReceiverConnected` and `SenderConnected` events is not guaranteed.

If a correlator (or other component) disconnects or terminates while the adapter is running, the adapter will receive both `ReceiverDisconnected` and `SenderDisconnected` events. Again, the ordering of these events is not guaranteed. Once a `SenderDisconnected` event is received, no further events from that correlator will be received until a `SenderConnected` event is received. When a `ReceiverDisconnected` event is received, no more events will be sent to that correlator until a `ReceiverConnected` event is received. Note that in this situation, some previously sent events may not yet have reached that correlator. The events will be discarded (or sent to other receivers, if other receivers are connected).

On a reload of an adapter, the adapter will be stopped, new configuration loaded, and the adapter restarted. During this period, the IAF will not drop its connection unless the configuration of which components to connect to has changed. As such, if prior to stopping for a reload the correlator was connected, it is safe to assume that it remains connected unless, on reload, the adapters receive `SenderDisconnected` or `ReceiverDisconnected` events.

During a reload, the IAF can also load new adapters. In this event, as the IAF may already have a connection open, no `ReceiverConnected` or `ReceiverDisconnected` event may be received by the new adapters. It is thus recommended to not change transports and codecs when reconfiguring the IAF if the adapters depend on receiving the out of band events. In practice, it is unusual to change the loaded transports or codecs.

Once an adapter has entered a stopped state, it will not receive any further events (unless it later re-enters a started state). Since the shutdown order of the IAF is to move all adapters to their 'stopped' state, then disconnect from downstream processes, adapters will not receive a final 'disconnected' event. Therefore, the adapter may need to notify external systems on the `stop` function being called, as well as on disconnected events.

The following topics describe the ordering the transport will see of calls to `start`, `stop` and the transport receiving out of band and normal events.

When starting the IAF

- IAF Begins Initialization
- Adapters Initialize
- IAF connects to Correlator
- [IAF Receives `SenderConnected` and `ReceiverConnected` - these are queued]
- Adapter changes state to Started

- Prior to receiving any other events, the semantic mapper (and then codec and adapter) receive the now unqueued out of band `SenderConnected` and `ReceiverConnected` events.
- The `SenderConnected` event will arrive before any other events from said sender are delivered

IAF shutdown requested

- Adapters state changes from started to stopped
- IAF disconnects from correlator
- Because transport is in state 'stopped', no events are received
- IAF terminates

IAF Configuration Reload

- Transport is in state 'started'
- IAF transitions transport to state 'stopped'
- IAF keeps its connection to the correlator up
- IAF transitions transport to state started
- Transport checks state, notices that it believes a connection is up, and continues to work without any changes

IAF Configuration reload changes correlator connection

- Transport is in state 'started'
- IAF transitions transport to state 'stopped'
- IAF breaks its connection to the correlator
- IAF receives `ReceiverDisconnected` and `SenderDisconnected`
- Since the transports are stopped, these events are queued
- IAF opens a new connection to a new correlator
- IAF receives `ReceiverConnected` and `SenderConnected`
- Since the transports are stopped, these events are queued
- IAF transitions transport to state started
- Transport checks state, notices that it believes a connection is up, and continues to work without any changes
- Prior to receiving any other events, the `ReceiverDisconnected` and `SenderDisconnected` events are received
- Following these, but prior to receiving any other events, the `ReceiverConnected` and `SenderConnected` events are received
- The transport can then behave as if a new connection has been made

Correlator dies (and a new one is started) while the IAF is running

- Transport is in state 'started'

- Correlator breaks its connection to the IAF
- IAF receives `ReceiverDisconnected` and `SenderDisconnected`
- Transport receives `ReceiverDisconnected` and `SenderDisconnected`
- Following `SenderDisconnected` no more events should arrive from the correlator
- Time Passes
- A new correlator makes a connection to the IAF
- IAF receives `ReceiverConnected` and `SenderConnected`
- Transport receives `ReceiverConnected` and `SenderConnected`
- The transport can now behave as if a new connection has been made

Out of and Connection Notifications

Chapter 11: The Event Payload

As already described, Apama events are rigidly structured and need to comply with a precise event type definition. This describes the structure of a particular event: in particular its name, as well as the order, name, type and number of its constituent fields.

By contrast, external events, even when they are of the same 'type' or nature (e.g. all `Trade` events or `News` headlines) might vary in format and structure, even when originating from the same source or feed.

In order to accommodate this, Apama provides an optional *payload* field in Apama events. The payload field, typically the last field in an event type definition, can embed any number of additional optional fields in addition to the always-present primary fields.

For example, consider an external message that can appear in several guises, but where each always consists of a particular subset of critical fields together with a variable number of additional optional fields.

If it is desired that these varying guises are mapped to a single Apama event type, then this needs to be defined so that its fields correspond to the subset of critical (and always present) fields, followed by a payload field into which the additional (and optional) fields are embedded.

Creating a payload field

When so configured, the Semantic Mapper will transparently create a payload field in an event.

As described in "Event mappings configuration" on page 29, one of the attributes of the event-mapping element `<event>` is `copyUnmappedToDictionaryPayload`.

Note: As of Release 4.1, the `copyUnmappedToPayload` attribute is deprecated.

The `copyUnmappedToDictionaryPayload` attribute defines what the Semantic Mapper should do with any fields in the incoming messages that do not match with any field mapping, i.e. if there are no rules that specifically copy their contents into a field within the Apama event being generated.

If both these attributes are set to `false` (or if one is `false` and the other is not present), any unmapped fields are discarded. If the value of either `copyUnmappedToDictionaryPayload` or `copyUnmappedToPayload` is set to `true`, unmapped fields will be packaged into a payload field, called `__payload`, set to be the last field of the Apama event type generated by the Semantic Mapper. If the values of both `copyUnmappedToDictionaryPayload` and `copyUnmappedToPayload` are set to true, `copyUnmappedToPayload` is ignored.

The Event Payload

Accessing the payload in the correlator

Apama recommends that you use `copyUnmappedToDictionaryPayload` instead of the deprecated `copyUnmappedToPayload`. Using `copyUnmappedToDictionaryPayload` puts all the payload fields in a standard EPL dictionary, it is more easily accessed, more efficient, and easier to use. In contrast, `copyUnmappedToPayload` uses a custom format that requires the use of the PayloadPlugin to decode and access.

The contents of the payload field used by `copyUnmappedToPayload` are structured as a list of field name and value pairs. Although this may be directly accessed as a string within EPL code, Apama provides the Payload Correlator plug-in to make extraction and manipulation of specific fields more straightforward. This plug-in is included with Apama.

The Payload Extraction plug-in is available as `libPayloadPlugin.so` in the Apama installation's `lib` directory on UNIX. On Microsoft Windows it is available as `PayloadPlugin.dll` in the Apama installation's `bin` folder.

For information on how to use the Payload Extraction plug-in, and on the suite of EPL functions it provides, please refer to "Using the payload extraction plug-in" in *Developing Apama Applications in EPL*.

The Event Payload

Chapter 12: Standard Plug-ins

Apama includes the following standard codec plug-ins:

- "The Null Codec plug-in" on page 137
- "The File Transport plug-in" on page 140
- "The String Codec plug-in" on page 141
- "The Filter Codec plug-in" on page 142
- "The XML codec plug-in" on page 145
- "The CSV codec plug-in" on page 158
- "The Fixed Width codec plug-in" on page 160

As well as being useful examples of how event transports and codecs can be written, these plug-ins are provided for your convenience as an aid to testing and the development of custom adapters that make use of them 'as is'.

The compiled binaries for all the standard plug-ins are copied to the `\bin` and `\lib` directories (for the C and Java versions respectively) automatically if you chose Developer during the Apama installation.

Information on where to find the source code and how to build those plug-ins for which source code is available can be found in "IAF samples" on page 45.

The Null Codec plug-in

The `NullCodec`/`JNullCodec` codec layer plug-ins are very useful in situations where it does not make sense to decouple the transport and codec layers. The transport layer plug-in might be best placed to perform all the necessary encoding and/or decoding of events, and to supply and receive Apama normalized events, rather than custom transport-specific messages.

The Null Codec plug-in is provided to make it easy to develop such transport plug-ins. This is a trivial codec layer plug-in that passes downstream normalized events from the transport layer to the Semantic Mapper, and upstream normalized events from the Semantic Mapper to the transport layer with no modification.

In order to load this plug-in, the `<codec>` element in the adapter's configuration file needs to load the `NullCodec` or `JNullCodec` library (this represents the filename of the library that implements the plug-in). Note that for the Java version, the full path to the plug-in's `.jar` file needs to be specified.

A configuration file for C/C++ uses this:

```
<codec name="NullCodec" library="NullCodec">
```

In a configuration file for Java:

```
<codec name="JNullCodec"
          jarName="Apama_install_dir\lib\JNullCodec.jar"          className="com.apama.iaf.codec.nullcodec.JNull
```

Note: The `NullCodec` and `JNullCodec` plug-ins can only be used with transport plug-ins that understand `NormalisedEvent` objects. The Null Codec plug-ins expect downstream `NormalisedEvent` objects from the transport and pass upstream `NormalisedEvent` objects it receives directly to the transport plug-in. Using the Null Codec plug-ins with a transport that expects any other kind of object does not work and can possibly crash the adapter.

Null codec transport-related properties

This codec plug-in supports standard Apama properties that are used to specify the name of the transport that will send upstream measages.

Transport-related properties

- **`transportName`** - This property specifies the transport that the codec should send upstream events to. The property can be used multiple times. The codec maintains a list of all transport names specified in the IAF configuration file. A `transportName` property with an empty value is ignored by the codec.

  If no transports are provided in the configuration file then the codec saves the last added `EventTransport` as the default transport. An upstream event is sent to the default transport if no transport information is provided in the normalized event or in the IAF configuration file.

- **`transportFieldName`** - This property specifies the name of the normalized event field whose value gives the name of the transport that the codec should send the upstream event to. You can also provide a transport name by specifying a value in the `__transport` field. Empty values of these fields are ignored and treated as if not present.

- **`removeTransportField`** - The value of this property specifies whether the transport related fields should be removed from the upstream event before sending it to transport. The default value is `true`. If the property is set then the field specified by the `transportFieldName` property and the field named `__transport` are removed from the upstream event if they are present. Values '`yes`', '`y`', '`true`', '`t`', '`1`' ignore cases and are treated as `true` for this property; any other value is treated as `false`.

Upstream behavior

The plug-in's behavior when an upstream event is received proceeds in this order:

1. The codec gets the name of the field that contains the transport name from the value of `transportFieldName` property. From the specified field, the codec then gets the transport name and sends the event to that transport. If the `transportFieldName` property is not specified, if the value

of the property is empty, if the field is not present in the event, or if the transport name is empty then then codec tries [2].

For example, the following configuration specifies two transports and the filter codec specifies a transport field named `TRANSPORT`:

```
<transports>
    <transport name="MARKET_DATA" library="transport-lib">
        <property name="Host" value="datahost.com" />
        <property name="Port" value="444" />
    </transport>
    <transport name="ORDER_MANAGEMENT" library="transport-lib">
        <property name="Host" value="orderhost.com" />
        <property name="Port" value="1234" />
    </transport>
</transports>
<codecs>
    <codec name="NullCodec" library="NullCodec">
        <property name="transportFieldName" value="TRANSPORT"/>
         ...
    </codec>
</codecs>
```

The IAF can now route any upstream event that defines a `TRANSPORT` field to one of these two transports. The value of the `TRANSPORT` field, either `MARKET_DATA` or `ORDER_MANAGEMENT`, determines the transport. Note: If the `removeTransportField` property is set `true` or not defined, then the `TRANSPORT` field and `__transport` will be removed (if present) from the upstream event before sending it to transport.

2. The codec gets the transport name from the `_transport` field of the normalized event and sends the event to specified transport. If the `_transport` field is not present or if the transport name specified is empty, the codec then tries [3].

   For example, in the above configuration, consider an upstream event that does not have a `TRANSPORT` field or the value of the field is empty. If this event has a value in the `__transport` field of either `MARKET_DATA` or `ORDER_MANAGEMENT`, then that value determines the transport.

3. The codec loops through all transports specified in the `transportName` property and sends the event to the transport. If no transport is specified then the codec tries [4]. Note that the codec ignores all transport names that are empty.

   If an exception occurs while sending the event to any transport, then the codec logs the exception and continues sending events to the remaining transports. If the codec was able to send the event to at least one transport, then it does not throws an exception; otherwise, it throws the last captured exception.

   For example, the following configuration specifies two transports:

```
<transports>
    <transport name="MARKET_DATA" library="transport-lib">
        <property name="Host" value="datahost.com" />
        <property name="Port" value="444" />
    </transport>
    <transport name="ORDER_MANAGEMENT" library="transport-lib">
        <property name="Host" value="orderhost.com" />
        <property name="Port" value="1234" />
    </transport>
</transports>
<codecs>
    <codec name="NullCodec" library="NullCodec">
        <property name="transportName" value="ORDER_MANAGEMENT"/>
        ...
    </codec>
</codecs>
```

In this example, the codec has not defined the `transportFieldName` property. The IAF will route any upstream event that does not contain a `__transport` field or has empty value in that field to the `ORDER_MANAGEMENT` transport.

4. If a default transport name is present, then the codec sends the event to that transport. The default transport is the last-added transport. If a default transport is also not found then, it throws an exception.

The File Transport plug-in

The `FileTransport/JFileTransport` transport layer plug-ins can read and write messages both from and to a text file. This makes it very convenient for testing string encoding and decoding, semantic mappings, and EPL code, because a text file with some sample messages can be put together quickly and then run through the IAF. Similarly in the upstream direction it allows messages to be written to a file instead of an external message sink such as a middleware message bus.

Messages (or events) are read from and written to named files. Each line of the input file is taken to be a single input event. Each output event is written to a new line of the output file.

In order to load this plug-in, the `<transport>` element in the adapter's configuration file must load the `FileTransport` library (this represents the filename of the library that implements the plug-in). Note that for the Java version, the full path to the plug-in's `.jar` file must be specified.

A configuration file for C/C++ would use this:

```
<transport name="FileTransport" library="FileTransport">
```

In a configuration file for Java:

```
<transport name="JFileTransport"
   jarName="Apama_install_dir\lib\JFileAdapter.jar"
   className="com.apama.iaf.transport.file.JFileTransport">
```

The File Transport plug-in takes the following properties:

- `input` — Specifies the name of the input file.
- `output` — Specifies the name of the output file.
- `cycle` — Specifies the number of times that the plug-in should cycle through the input file. Any value less than zero causes the plug-in to cycle endlessly, until the adapter is either shut down or re-configured. A zero value (the default if the property is missing) means 'no cycling' and results in the same behavior as if the value of this property was '`1`'.

For more information on specifying plug-ins in an adapter's configuration file, see "Transport and codec plug-in configuration" on page 27.

The plug-in automatically stops after reading the entire input file the requested number of times. If the adapter is subsequently asked to reload its configuration, the plug-in starts running again, using the current property values in the configuration file. If the adapter configuration is reloaded while the plug-in is running, the new configuration will not take effect until the plug-in reaches the end of the current input file. In this case, a second reload request is required before the plug-in will actually start reading the new file.

By default, the File Transport plug-in always communicates with the event codec using Java `String` objects. Therefore, the String Codec plug-in is a suitable companion as it provides a mechanism for converting between `String` objects and normalized events.

There are some minor differences between the C and Java implementations:

- In the **C** version, if no input filename is specified, the standard input stream is used; similarly if no output filename is specified the standard output stream is used.
- In the **Java** version, there is an extra property called `upstreamNormalised`. If this is specified and set to `true`, the File Transport communicates with its codec using `NormalisedEvent` objects rather than `String` objects. In this configuration it should be used with the `JNullCodec`, which does not perform any encoding or decoding but simply passes the unchanged `NormalisedEvent` objects between the codec and transport layers. If `upstreamNormalised` is set to `true`, the File Transport uses the functionality of the `JStringCodec` class to perform encoding/decoding, and all the properties available for use with the `JStringCodec` plug-in class can be specified as properties to the `JFileTransport`.

This is one of the sample plug-ins for which source code is available – see "IAF samples" on page 45 for more information.

The String Codec plug-in

The `StringCodec`/`JStringCodec` codec plug-ins read transport events as simple text strings and breaks them into fields, names and values, using delimiter strings supplied by configuration properties.

Events are assumed to have the following general format:

<name1><sepA><value1><sepB><name2><sepA><value2><sepB>…<namen><sepA><valuen><terminator>

where <*name*> corresponds to the field name, followed by a delimiter character or string <*sepA*>, followed by the field's value, <*value*>. The complete <*name*> and <*value*> pair is then separated from another such sequence by a <*sepB*> delimiter. This pattern is assumed to repeat itself.

Fields with empty values are permitted. Because the terminator is optional, the codec will consume names and values up to the end of the input string if no terminator is found.

In order to load this plug-in, the `<codecs>` element in the adapter's configuration file must load the `StringCodec` library (this represents the filename of the library that implements the plug-in). Note that for the Java version, the full path to the plug-in's `.jar` file must be specified.

A configuration file for C/C++ would use this:

```
<codec name="StringCodec" library="StringCodec">
```

In a configuration file for Java:

```
<codec name="JStringCodec"
       jarName="Apama_install_dir\lib\JFileAdapter.jar"
       className="com.apama.iaf.codec.string.JStringCodec">
```

The String Codec plug-in takes the following properties:

- `NameValueSeparator` — The string used to separate names and values (<*sepA*> above).
- `FieldSeparator` — The string used to separate fields (<*sepB*> above).
- `Terminator` — The string used to mark the end of the event string.

All properties must be specified in the adapter configuration file.

For more information on specifying plug-ins in an adapter's configuration file, see "Transport and codec plug-in configuration" on page 27.

This is one of the sample plug-ins for which source code is available – see the "IAF samples" on page 45 for more information.

The Filter Codec plug-in

The Apama filter codec plug-ins filter normalized event fields. You can use the filter codec to:

- Route upstream events to particular transports
- Remove particular fields from upstream and/or downstream events

To use the filter codec, the `FilterCodec` or `JFilterCodec` library must be available to the IAF at runtime. These are the filenames of the C++ and Java libraries that implements the plug-in.

In order to load this plug-in, the `<codec>` element in the adapter's configuration file needs to load either the `FilterCodec` or `JFilterCodec` library. Note that for the Java version, the full path to the plug-in's `.jar` file needs to be specified.

A configuration file for C/C++ uses this:

```
<codec name="FilterCodec" library="FilterCodec">
```

In a configuration file for Java:

```
<codec name="JFilterCodec"
   jarName="Apama_install_dir\lib\JFilterCodec.jar"
   className="com.apama.iaf.codec.filtercodec.JFilterCodec">
```

To configure the filter codec, add the following to the `<codecs>` section of the IAF configuration file:

```
<codec name="FilterCodec" library="FilterCodec">
  <property name="transportFieldName" value="transport_field_name"/>
  <property name="filter_spec_1" value="filter_condition_1"/>
  <property name="filter_spec_2" value="filter_condition_2"/>
...
  <property name="filter_spec_n" value="filter_condition_n"/>
</codec>
```

Details for replacing the variables in the above `codec` section are in the following topics:

- "Filter codec transport-related properties" on page 142
- "Specifying filters for the filter codec" on page 144
- "Examples of filter specifications" on page 145.

Filter codec transport-related properties

This codec plug-in supports standard Apama properties that are used to specify the name of the transport that will send upstream measages.

Transport-related properties

- **`transportName`** - This property specifies the transport that the codec should send upstream events to. The property can be used multiple times. The codec maintains a list of all transport names specified in the IAF configuration file. A `transportName` property with an empty value is ignored by the codec.

If no transports are provided in the configuration file then the codec saves the last added `EventTransport` as the default transport. An upstream event is sent to the default transport if no transport information is provided in the normalized event or in the IAF configuration file.

- **`transportFieldName`** - This property specifies the name of the normalized event field whose value gives the name of the transport that the codec should send the upstream event to. You can also provide a transport name by specifying a value in the `__transport` field. Empty values of these fields are ignored and treated as if not present.
- **`removeTransportField`** - The value of this property specifies whether the transport related fields should be removed from the upstream event before sending it to transport. The default value is `true`. If the property is set then the field specified by the `transportFieldName` property and the field named `__transport` are removed from the upstream event if they are present. Values '`yes`', '`y`', '`true`', '`t`', '`1`' ignore cases and are treated as `true` for this property; any other value is treated as `false`.

Upstream behavior

The plug-in's behavior when an upstream event is received proceeds in this order:

1. The codec gets the name of the field that contains the transport name from the value of `transportFieldName` property. From the specified field, the codec then gets the transport name and sends the event to that transport. If the `transportFieldName` property is not specified, if the value of the property is empty, if the field is not present in the event, or if the transport name is empty then then codec tries [2].

   For example, the following configuration specifies two transports and the filter codec specifies a transport field named `TRANSPORT`:

   ```
   <transports>
       <transport name="MARKET_DATA" library="transport-lib">
           <property name="Host" value="datahost.com" />
           <property name="Port" value="444" />
       </transport>
       <transport name="ORDER_MANAGEMENT" library="transport-lib">
           <property name="Host" value="orderhost.com" />
           <property name="Port" value="1234" />
       </transport>
   </transports>
   <codecs>
       <codec name="FilterCodec" library="FilterCodec">
           <property name="transportFieldName" value="TRANSPORT"/>
            ...
       </codec>
   </codecs>
   ```

   The IAF can now route any upstream event that defines a `TRANSPORT` field to one of these two transports. The value of the `TRANSPORT` field, either `MARKET_DATA` or `ORDER_MANAGEMENT`, determines the transport. Note: If the `removeTransportField` property is set `true` or not defined, then the `TRANSPORT` field and `__transport` will be removed (if present) from the upstream event before sending it to transport.

2. The codec gets the transport name from the `_transport` field of the normalized event and sends the event to specified transport. If the `_transport` field is not present or if the transport name specified is empty, the codec then tries [3].

   For example, in the above configuration, consider an upstream event that does not have a `TRANSPORT` field or the value of the field is empty. If this event has a value in the `__transport` field of either `MARKET_DATA` or `ORDER_MANAGEMENT`, then that value determines the transport.

3. The codec loops through all transports specified in the `transportName` property and sends the event to the transport. If no transport is specified then the codec tries [4]. Note that the codec ignores all transport names that are empty.

   If an exception occurs while sending the event to any transport, then the codec logs the exception and continues sending events to the remaining transports. If the codec was able to send the event to at least one transport, then it does not throws an exception; otherwise, it throws the last captured exception.

   For example, the following configuration specifies two transports:

```
<transports>
    <transport name="MARKET_DATA" library="transport-lib">
        <property name="Host" value="datahost.com" />
        <property name="Port" value="444" />
    </transport>
    <transport name="ORDER_MANAGEMENT" library="transport-lib">
        <property name="Host" value="orderhost.com" />
        <property name="Port" value="1234" />
    </transport>
</transports>
<codecs>
    <codec name="FilterCodec" library="FilterCodec">
        <property name="transportName" value="ORDER_MANAGEMENT"/>
        ...
    </codec>
</codecs>
```

   In this example, the codec has not defined the `transportFieldName` property. The IAF will route any upstream event that does not contain a `__transport` field or has empty value in that field to the `ORDER_MANAGEMENT` transport.

4. If a default transport name is present, then the codec sends the event to that transport. The default transport is the last-added transport. If a default transport is also not found then, it throws an exception.

Specifying filters for the filter codec

You specify each filter as a codec property. The filter codec plug-in applies each filter you specify to incoming and outgoing events as they pass through the codec. The property name identifies the field(s) that the filter applies to and the property value specifies the condition that must be true for the filter to operate.

The general syntax of a filter specification is:

`<property name="filter[.direction][.field_name]" value="condition" />`

direction	Indicates the direction of the events that the filter applies to. Specify `downstream`, `upstream`, or `both`. The default is `both`.
field_name	Identifies the field that the filter applies to. The default is that the filter applies to all fields in the event.
condition	Specifies the value that the field must have that causes it to be removed from the event.

Examples of filter specifications

The following filter removes the `price` field from upstream events when the value of the `price` field is `0.0`:

```
<property name="filter.upstream.price" value="0.0"/>
```

The following filter removes the `name` field from upstream and downstream events when the value of the `name` field is `NULL`:

```
<property name="filter.both.name" value="NULL"/>
```

In upstream events, the following filter removes each field in which the value is `55`:

```
<property name="filter.upstream" value="55"/>
```

In upstream and downstream events, the following filter removes each field in which the value is `<remove>`:

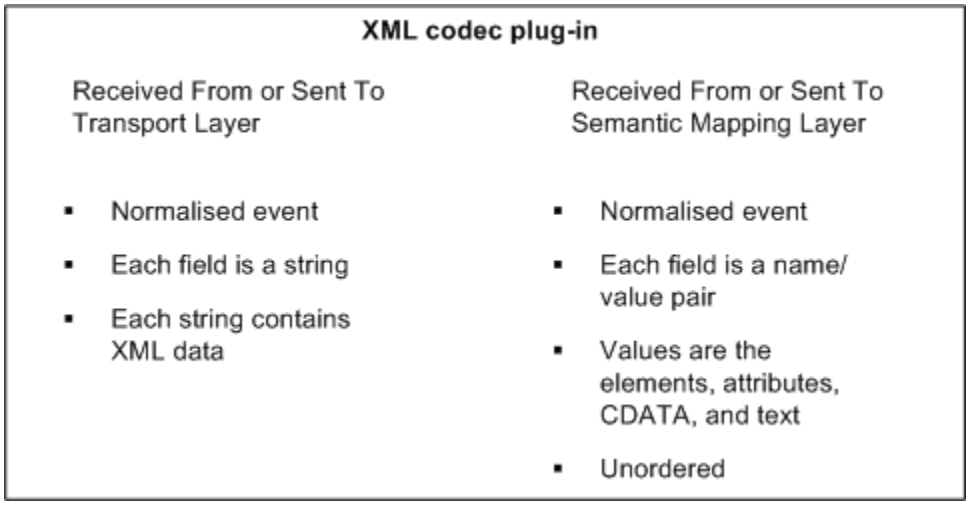
```
<property name="filter" value="<remove>"/>
```

The XML codec plug-in

The Apama XML codec converts messages between the following two formats:

- IAF normalized event whose field values are strings that contain XML data.
- Normalized event in which each field is a name/value pair. These unordered fields contain elements, attributes, CDATA, and text.

Figure 2. XML codec plug-in



To use the XML codec, you must add some information to the IAF configuration file and then set up the classpath. After you do this, you can launch the adapter by running the IAF executable.

For an example configuration file, see `adapters\config\XMLCodec-example.xml.dist` in the Apama installation directory. This file can be changed as required for the purposes of your data and the content added to the adapter configuration file in which the codec is to be used.

Use the information in the following topics to help you configure the XML codec:

Supported XML features

The XML codec can convert messages that contain the following:

- Elements
- Attributes
- Text nodes
- CDATA nodes, including CDATA nodes that contain an XML document to be parsed

CDATA nodes are supported only in the downstream direction.

- Namespace prefixes and definitions (only basic support)
- XPath expressions, including functions

Result types of XPath expressions must be simple. For example,

```
string contains();
```

The XML codec cannot convert XML data that contains the following XML features:

- Document type specifiers
- Processing instructions
- Notations and entities
- XML with more than one top-level (root) element
- Node or nodeset XPath expressions

For Node or nodeset XPath expressions, only the first match is returned.

Adding XML codec to adapter configuration

To include the XML codec in the adapter configuration, add the following to the `<codecs>` section of the IAF configuration file:

```
<codec name="XMLCodec"
```

```
   className="com.apama.iaf.codec.xml.XMLCodec"
   jarName="@ADAPTERS_JARDIR@\XMLCodec.jar"
   >
   <!-- Properties go here -->
</codec>
```

Typically, `@ADAPTERS_JARDIR@` is the `APAMA_HOME\adapters\lib` directory.

For details about the properties that you can specify, see "Specifying XML codec properties" on page 148.

Setting up the classpath

To use the XML codec, ensure the following JAR files in the `APAMA_HOME\lib` directory are in the adapter classpath when you run the IAF.

```
jplugin_public@LIBRARY_VERSION@.jar
util@LIBRARY_VERSION@.jar
jdom.1.0.jar
```

If the XML codec JAR file is in the `APAMA_HOME\adapters\lib` directory, you are all set. The IAF finds these dependencies automatically. Otherwise, set the classpath either as an environment variable or in the `<java>` section of the IAF configuration file.

About the XML parser

On startup, the XML codec logs the names of the classes it is using for XML parsing and XML generation. For example:

```
INFO  [11808] - XMLCodec: Encoder initialized: using XML Document builder
                'org.apache.xerces.jaxp.DocumentBuilderImpl'
INFO  [11808] - XMLCodec: Decoder initialized: using Streaming API for XML (StAX)
                'com.ctc.wstx.stax.WstxInputFactory'
```

Apama uses Xerces for encoding (creating XML docs) and Woodstox StAX for decoding (parsing).

XML namespace support

If your application relies on the standard XML parsing/generation behavior (that is, not XPath) there is no concept of "declaring namespaces" in the XML codec nor is it required as long as the XML document is valid (that is, it declares any namespace prefixes it uses) then you can just use `namespaceprefix:elementName` when referring to elements in your mapping rules. If there is any doubt, you can run your sample message through the XMLCodec property `logFlattenedXML=true` and it will show you what to specify in your mapping rules, for example, consider the following sample message:

```
<h:table xmlns:h="http://www.myco.com/apama/test/testnamespace_h/"
         xmlns="http://www.myco.com/apama/test/testnamespace_default">
    <h:tr>
    <h:td>Apples</h:td>
    <td>Bananas</td>
    </h:tr>
    </h:table>
```

With the above sample message you could use mapping rules such as:

```
<map type="string" default="" apama="default_namespace"
       transport="Body.h:table/@xmlns"/>
```

```
<map type="string" default="" apama="prefix_namespace"
       transport="Body.h:table/@xmlns:h"/>
<map type="string" default="" apama="prefixed_element_text"
       transport="Body.h:table/h:tr/h:td/text()"/>
<map type="string" default="" apama="non_prefixed_element_text"
       transport="Body.h:table/h:tr/td/text()"/>
```

If you use XPath in your application, XPath itself contains operators to access the local (non-namespace) name and namespace URI of any XML content. However it is often convenient to define some global prefixes to make it easier to refer to namespaced elements. Apama supports this by allowing any number of `XPathNamespace:myprefix` codec properties, whose value is the URN that the specified prefix should point to. For example,

```
<property name="XPathNamespace:b" value="urn:xmlns:mynamespace"/>
```

would allow XPath expressions to use "`b`" to refer to elements in the "`mynamespace`" namespace:

```
<property name="XPath:Test.root/b:elementname/text()"/>
```

Specifying XML codec properties

In the XML codec section of the IAF configuration file, you can set a number of XML properties. For details about setting properties in the IAF configuration file, see "Plug-in <property> elements" on page 28.

When you reload the IAF, any changes to these configuration properties take effect in the codec. In addition to specifying these properties, you must also set up event mappings for XML messages. See "Event mappings configuration" on page 29.

Properties are described in the following topics:

Required XML codec properties

The XML codec requires you to set the `XMLField` and `transportName` properties. All other properties are optional.

`XMLField` — This property identifies the field name that XML will be read from when decoding, and will be written to when encoding. The flattened XML representation is stored in fields with names prefixed with the value you specify for the `XMLField` property.

When you are familiar with how the XML codec behaves, you can specify the `XMLField` property multiple times to parse/generate multiple XML documents per event. Parsing follows the order in which `XMLField` properties appear, and generating XML follows the reverse order.

It is possible to use this mechanism to parse an XML string embedded as CDATA in another XML string. To do this, specify the flattened field name of the CDATA node as an `XMLField`. However, note that sequence fields across separate CDATA nodes are not supported.

`transportName` — The XML codec sends upstream events to the transport that this property identifies. This transport must be defined in the same IAF configuration file.

XML codec transport-related properties

This codec plug-in supports standard Apama properties that are used to specify the name of the transport that will send upstream measages.

Transport-related properties

- **`transportName`** - This property specifies the transport that the codec should send upstream events to. The property can be used multiple times. The codec maintains a list of all transport names specified in the IAF configuration file. A `transportName` property with an empty value is ignored by the codec.

  If no transports are provided in the configuration file then the codec saves the last added `EventTransport` as the default transport. An upstream event is sent to the default transport if no transport information is provided in the normalized event or in the IAF configuration file.

- **`transportFieldName`** - This property specifies the name of the normalized event field whose value gives the name of the transport that the codec should send the upstream event to. You can also provide a transport name by specifying a value in the `__transport` field. Empty values of these fields are ignored and treated as if not present.

- **`removeTransportField`** - The value of this property specifies whether the transport related fields should be removed from the upstream event before sending it to transport. The default value is `true`. If the property is set then the field specified by the `transportFieldName` property and the field named `__transport` are removed from the upstream event if they are present. Values '`yes`', '`y`', '`true`', '`t`', '`1`' ignore cases and are treated as `true` for this property; any other value is treated as `false`.

Upstream behavior

The plug-in's behavior when an upstream event is received proceeds in this order:

1. The codec gets the name of the field that contains the transport name from the value of `transportFieldName` property. From the specified field, the codec then gets the transport name and sends the event to that transport. If the `transportFieldName` property is not specified, if the value of the property is empty, if the field is not present in the event, or if the transport name is empty then then codec tries [2].

   For example, the following configuration specifies two transports and the filter codec specifies a transport field named `TRANSPORT`:

   ```
   <transports>
       <transport name="MARKET_DATA" library="transport-lib">
           <property name="Host" value="datahost.com" />
           <property name="Port" value="444" />
       </transport>
       <transport name="ORDER_MANAGEMENT" library="transport-lib">
           <property name="Host" value="orderhost.com" />
           <property name="Port" value="1234" />
       </transport>
   </transports>
   <codecs>
       <codec name="XMLCodec"
   ```

```
            className="com.apama.iaf.codec.xml.XMLCode"
            jarName=""@ADAPTERS_JARDIR@\XMLCodec.jar">
        <property name="transportFieldName" value="TRANSPORT"/>
         ...
    </codec>
</codecs>
```

The IAF can now route any upstream event that defines a `TRANSPORT` field to one of these two transports. The value of the `TRANSPORT` field, either `MARKET_DATA` or `ORDER_MANAGEMENT`, determines the transport. Note: If the `removeTransportField` property is set `true` or not defined, then the `TRANSPORT` field and `__transport` will be removed (if present) from the upstream event before sending it to transport.

2. The codec gets the transport name from the `_transport` field of the normalized event and sends the event to specified transport. If the `_transport` field is not present or if the transport name specified is empty, the codec then tries [3].

   For example, in the above configuration, consider an upstream event that does not have a `TRANSPORT` field or the value of the field is empty. If this event has a value in the `__transport` field of either `MARKET_DATA` or `ORDER_MANAGEMENT`, then that value determines the transport.

3. The codec loops through all transports specified in the `transportName` property and sends the event to the transport. If no transport is specified then the codec tries [4]. Note that the codec ignores all transport names that are empty.

   If an exception occurs while sending the event to any transport, then the codec logs the exception and continues sending events to the remaining transports. If the codec was able to send the event to at least one transport, then it does not throws an exception; otherwise, it throws the last captured exception.

   For example, the following configuration specifies two transports:

   ```
   <transports>
       <transport name="MARKET_DATA" library="transport-lib">
           <property name="Host" value="datahost.com" />
           <property name="Port" value="444" />
       </transport>
       <transport name="ORDER_MANAGEMENT" library="transport-lib">
           <property name="Host" value="orderhost.com" />
           <property name="Port" value="1234" />
       </transport>
   </transports>
   <codecs>
       <codec name="XMLCodec"
                 className="com.apama.iaf.codec.xml.XMLCodec"
                 jarName="@ADAPTERS_JARDIR@\XMLCodec/jar">
           <property name="transportName" value="ORDER_MANAGEMENT"/>
           ...
       </codec>
   </codecs>
   ```

   In this example, the codec has not defined the `transportFieldName` property. The IAF will route any upstream event that does not contain a `__transport` field or has empty value in that field to the `ORDER_MANAGEMENT` transport.

4. If a default transport name is present, then the codec sends the event to that transport. The default transport is the last-added transport. If a default transport is also not found then, it throws an exception.

Message logging properties

`logFlattenedXML` — If true, the IAF log contains a list of the name/value pairs generated by the XML codec when flattening XML received from the transport, at `CRIT` level. Each field is on a different line, which makes it easy to see what fields are being generated and what the mapping's transport field names should be set to. Turning this on in production impacts performance. The default is false.

`logAllMessages` — If true, the IAF log contains the full contents of every message sent upstream or downstream, before and after encoding, and before and after decoding, all at `CRIT` level. Turning this on in production impacts performance. The default is false.

Downstream node order suffix properties

`generateTwinOrderSuffix` — If true, all field names for text, CDATA and element nodes are appended with "", "`[2]`", "`[3]`", and so on. The number specifies the position of this node relative to 'twins', that is, nodes of the same type and name. These order suffixes provide a partial order for the XML nodes. Note that the first child node with a given name is defined to have no suffix (rather than an explicit "`[1]`"), to improve readability. The default is false.

Use this property when you need to map fields without sensitivity to the precise order in which differently named nodes appear in the XML. This is probably a more useful option than setting the `generateSiblingOrderSuffix` property for most users of the XML codec.

`generateSiblingOrderSuffix` — If true, all field names for text, CDATA and element nodes (except the root element) are appended with "`#1`", "`#2`", and so on. The number specifies the position of this node relative to all its siblings (of any type, such as element or CDATA.). These order suffixes provide a total order for the XML nodes. The default is true.

Use this property when you need to map fields using the precise order in which differently named nodes appear in the XML, or for total control over node ordering when generating XML upstream.

Examples of both suffixes are in "Description of event fields that represent normalized XML" on page 153 and "Examples of conversions" on page 155.

You can set both node order properties to true. For sample output when both are set to true, see "Examples of conversions" on page 155. The default values of these two properties may change in a future release, so the recommendation is to explicitly specify both properties according to the behavior required.

Additional downstream properties

`XPath: XMLField -> ResultField` — The value of this property specifies an XPath expression that should be evaluated for the specified *`XMLField`*, with the result put into the *`ResultField`* in the normalized event. Only simple data types (boolean/float/string) can be returned at present, so XPath expressions that match multiple nodes only return the first matching node. See "XPath examples" on page 158.

`trimXMLText` — If true, the XML codec removes any leading or trailing whitespace characters from XML text data in downstream messages before adding the text to the normalized event. The default is true.

Sequence field properties

`sequenceField` — The value of this property is a field that is treated as a sequence. This means that all XML nodes that match this name are translated to a single entry in the normalized event, in the form

of an EPL `sequence` of type `string`. The element name should be a plain name, without a node order suffix. In other words, the value of this property and the field in the outgoing event should be in the form: `elementA/elementB/@attrib`. You can specify this property multiple times.

`ensurePresent` — This property specifies an attribute, text string or CDATA node of an element that will be added to the output event as a blank string even if it is not present in the XML. This is mostly useful for fields identified with the `sequenceField` property, as empty strings get added to the sequence for optional attributes. You can specify this property multiple times.

`separator: elementName` — Whenever the specified element occurs in the XML message, the value of this property is prepended to any sequences in nodes below the specified element. See "Sequence field example" on page 157.

Upstream properties

`indentGeneratedXML` — If true, the generated XML is indented to make it easier to read. The default is false.

`omitGeneratedXMLDeclaration` — If true, the `<?xml ... ?>` declaration at the start of the generated XML is not included. The default is false.

Performance properties

`skipNullFields` — A boolean that indicates whether you want the XML codec to omit nodes with null values from downstream, flattened, normalized events. Specify true to omit nodes with null values. The default is false.

The `skipNullFields` property applies to the name/value pairs for XML elements themselves. These have no associated data, so generating normalized event fields for them is not necessary unless they are required for ID rules. The `skipNullFields` property does not apply to a node whose value is an empty string.

Setting `skipNullFields` to true has no effect on the ordering suffixes that the codec adds to nodes. For example, consider an XML element that is deep within an XML hierarchy such as the following:

```
<root>
   <a>
      <b>
         <c>
            I want this string
         </c>
      </b>
   </a>
</root>
```

In the downstream direction, the XML codec creates a normalized event that contains a dictionary of name/value pairs that includes an entry for each element. If you specify sibling suffixes and `Test` as the XML field name, the dictionary contains the following:

```
{  "Test.root/":null ,
   "Test.root/a#1/":null ,
   "Test.root/a#1/b#1/":null ,
   "Test.root/a#1/b#1/c#1/":null ,
   "Test.root/a#1/b#1/c#1/text()#1:"I want this string"  }
```

Unless you require one of the null value fields for an ID rule, you do not need the null value fields. If you set `skipNullFields` to true, the XML codec drops the null value fields from the normalized event. In this example, the result is a dictionary with one entry:

```
{ "Test.root/a#1/b#1/c#1/text()#1:"I want this string" }
```

As you can see, this is much more lightweight. Turning this feature on can sometimes improve throughput by up to `1.5` times.

`parseNode` — Specify this property one or more times to identify only those nodes that you want parsed, flattened, and added to the normalized event.

By default, the XML codec parses, flattens, and adds all nodes to the normalized event. If you specify one or more `parseNode` property entries, the XML codec processes only the node or nodes specified by a `parseNode` property.

The value of a `parseNode` property can be any node path. The codec ignores order suffixes (`#`*n* or `[`*n*`]`) if you specify them in node paths. In other words, the codec parses all elements of the type specified in the `parseNode` property.

For example, suppose the value of the XML field property is `Test` and you have the following XML:

```
<root>
   <a>ignore me</a>
   <b>look at me</b>
   <c>look at me</c>
   <b>look at me again</b>
<root>
```

You can specify the following `parseNode` properties:

```
<property name="parseNode" value="Test.root/b/text()" />
<property name="parseNode" value="Test.root/c[99999999999]/text()" />
```

The XML codec produces the following dictionary entries:

```
"Test.root/b#1/text()#1" = "look at me"
"Test.root/c#2/text()#1" = "look at me"
"Test.root/b#3/text()#1" = "look at me again"
```

As you can see, the XML codec ignores the [99999999999] suffix.

Typically, you would specify the following `parseNode` properties:

- For each mapping rule, specify a `parseNode` property whose value is the transport field for that rule.
- For each ID rule in the adapter configuration file, specify a `parseNode` property whose value is the field name.

It is not necessary to specify `parseNode` properties for nodes identified by `sequenceField` or `separator:`*elementName* properties.

Setting the `parseNode` property prevents some nodes from being parsed. Consequently, the order of subsequent nodes might change, and therefore they would have different node order suffixes. For this reason, you probably want to set the `logFlattenedXML` property to true to see in what order suffixes are being generated before you add `parseNode` properties. Then add the `parseNode` properties and update the node paths used in mapping and ID rules as needed.

Specifying `parseNode` properties instead of parsing the entire document can result in very substantial throughput improvements. This is especially true for documents in which only a small proportion of the XML is actually going to be mapped.

Description of event fields that represent normalized XML

As mentioned before, a single XML field on the transport side is represented on the correlator side as a series of name/value fields, all prefixed by the value you specified for the `XMLField` property. This section describes how the XML codec names fields, based on the XML data.

Note, any field not specified as an `XMLField` for the XMLCodec will pass through the system as normal. These fields are not dropped/ignored.

If there is any uncertainty about the correct transport field names to use in the IAF mapping rules, try setting the `logFlattenedXML` codec property to true.

To preserve XML node ordering information, the codec adds ordering information to node names by appending a suffix according to the suffix generation mode enabled — either "", "`#2`", "`#3`", and so on or "`[1]`", "`[2]`", "`[3]`", and so on.

The "`#`*n*" sibling format provides a total ordering across all child nodes under a given parent, specifying each node's position relative to all of its sibling nodes. This suffix mode is the default. To turn it off, set the `generateSiblingOrderSuffix` codec property to false. Note that the root node never has a sibling order suffix because only one root exists. Sample field names:

```
Field1.message/element#1/
Field1.message/other_element#2/
Field1.message/other_element#3/
```

The twin "`[n]`" format is insensitive to the order in which nodes appear as long as they have different names, and it specifies a node's position relative to its twin nodes. (Twins are siblings with the same node name.) This suffix mode is disabled by default (for backwards compatability). To turn it on, set the `generateTwinOrderSuffix` codec property to true. To improve readability the first sibling node with a given name has no suffix. That is, the `[1]` suffix is implicit. Sample field names:

```
Field1.message/element/
Field1.message/element[2]/
Field1.message/other_element/
Field1.message/other_element[2]/
Field1.message/other_element[3]/
Field1.message/yet_another_element/
Field1.message/yet_another_element[2]/
```

Note that for a message to be correctly translated in the upstream direction (from the correlator), there do not have to be enough suffixes in the event to form a total order, but any suffixes that are provided will be used. In the absence of sibling order suffixes to determine ordering of different node types, the XML codec generates the XML nodes in the following order:

1. Text data
2. CDATA
3. Elements

The XML codec maps XML elements, attributes, CDATA and text data as described in the following sections. In the following topics, assume that the value of the `XMLField` property is `Test`.

Elements

An XML element maps to a field with the following characteristics:

- The name is separated and terminated with the '`/`' character.
- The value is an empty string ("").

For example, an element `B` nested inside an element `A` is represented in the normalized event as follows:

```
"Test.A/B#1/" = ""
```

When the XML codec generates XML for upstream events, it is not a requirement to have an associated field for every element. The XML codec automatically creates ancestor XML elements when they do not have associated fields. For example, consider the following field:

```
"Test.A/B#1/@att" = ""
```

If necessary, the codec creates the `A` and `B` element nodes.

Element attributes

XML element attributes map to fields with names equal to the parent element's field name, followed by '`@att`' where *`att`* is the name of the attribute, and the field's value is the attribute value. For example, an attribute `B` of an element `A` with the value `Hello` is represented as follows:

```
"Test.A/@B" = "Hello"
```

CDATA

XML CDATA in an element maps to a field with a name equal to the parent element's field name followed by `CDATA()` and a value that contains the text data. For example, an element `A` with CDATA `" Hello "` followed by sub-element B followed by CDATA `" World "` is represented as follows:

```
"Test.A/CDATA()#1" = " Hello "
"Test.A/B#2/"      = ""
"Test.A/CDATA()#3" = " World "
```

Text data

Text data in an XML element maps to a field with a name equal to the parent element's field name followed by `text()`. The value of the field is the text data. Unless the `trimXMLText` is false (the default is that it is true), the codec strips leading and trailing whitespace from text data. For example, an element `A` that contains the text `" Hello World "` followed by sub-element `B` followed by text `" ! "` is represented as follows:

```
"Test.A/text()#1" = "Hello World"
"Test.A/B#2/"     = ""
"Test.A/text()#3" = "!"
```

In the event of errors during XML parsing, the parser

- Logs the errors in the IAF log file
- Tries to send to the semantic mapper a flattened, normalized event that contains the remaining fields

Examples of conversions

Suppose that the value of the `XMLField` property is `Test`, and the value of the `trimXMLText` property is true. Consider the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
  <Message>
    <ElementA>
      Hello there
      <ElementB/>
      !
      <ElementC/>
      <![CDATA[Sample CDATA (with < and > comparison operators). ]]>
      <ElementB att1="X" att2="Y">
```

```
      <![CDATA[More CDATA in the same element.]]>
    </ElementB>
  </ElementA>
</Message>
```

With sibling order suffixing, this XML maps to the following normalized event fields:

```
"Test.Message/"                              =
"Test.Message/ElementA#1/"                   =
"Test.Message/ElementA#1/text()#1"           = "Hello there"
"Test.Message/ElementA#1/ElementB#2/"        =
"Test.Message/ElementA#1/text()#3"           = "!"
"Test.Message/ElementA#1/ElementC#4/"        =
"Test.Message/ElementA#1/CDATA()#5"          =
         "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA#1/ElementB#6/"        =
"Test.Message/ElementA#1/ElementB#6/@att1"   = "X"
"Test.Message/ElementA#1/ElementB#6/@att2"   = "Y"
"Test.Message/ElementA#1/ElementB#6/CDATA()#1" =
         "More CDATA in the same element."
```

With twin order suffixing, the same XML maps to the following normalized event fields:

```
"Test.Message/"                            =
"Test.Message/ElementA/"                   =
"Test.Message/ElementA/text()"             = "Hello there"
"Test.Message/ElementA/ElementB/"          =
"Test.Message/ElementA/text()[2]"          = "!"
"Test.Message/ElementA/ElementC/"          =
"Test.Message/ElementA/CDATA()"            =
         "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA/ElementB[2]/"       =
"Test.Message/ElementA/ElementB[2]/@att1"  = "X"
"Test.Message/ElementA/ElementB[2]/@att2"  = "Y"
"Test.Message/ElementA/ElementB[2]/CDATA()" = "More CDATA in the same element."
```

To construct the XML above (assuming element ordering matters, but allowing for text() concatenation), the following name/value pairs are all that is required:

```
"Test.Message/ElementA#1/text()#1"           = "Hello there"
"Test.Message/ElementA#1/ElementB#2/"        =
"Test.Message/ElementA#1/text()#3"           = "!"
"Test.Message/ElementA#1/ElementC#4/"        =
"Test.Message/ElementA#1/CDATA()#5"          =
         "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA#1/ElementB#6/@att1"   = "X"
"Test.Message/ElementA#1/ElementB#6/@att2"   = "Y"
"Test.Message/ElementA#1/ElementB#6/CDATA()#1" = "More CDATA in the same element."
```

With both sibling order suffixing and twin order suffixing set to true, the XML codec generates two field/value pairs for each node. For example, the same XML used in the previous two examples maps to the following:

```
"Test.Message/"                              =
"Test.Message/ElementA/"                     =
"Test.Message/ElementA#1/"                   =
"Test.Message/ElementA/text()"               = "Hello there"
"Test.Message/ElementA#1/text()#1"           = "Hello there"
"Test.Message/ElementA/ElementB/"            =
"Test.Message/ElementA#1/ElementB#2/"        =
"Test.Message/ElementA/text()[2]"            = "!"
"Test.Message/ElementA#1/text()#3"           = "!"
"Test.Message/ElementA/ElementC/"            =
"Test.Message/ElementA#1/ElementC#4/"        =
"Test.Message/ElementA/CDATA()"              =
         "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA#1/CDATA()#5"          =
         "Sample CDATA (with < and > comparison operators). "
"Test.Message/ElementA/ElementB[2]/"         =
"Test.Message/ElementA#1/ElementB#6/"        =
"Test.Message/ElementA/ElementB[2]/@att1"    = "X"
```

```
"Test.Message/ElementA#1/ElementB#6/@att1"     = "X"
"Test.Message/ElementA/ElementB[2]/@att2"      = "Y"
"Test.Message/ElementA#1/ElementB#6/@att2"     = "Y"
"Test.Message/ElementA/ElementB[2]/CDATA()"    = "More CDATA in the same element."
"Test.Message/ElementA#1/ElementB#6/CDATA()#1" = "More CDATA in the same element."
```

Since the suffix properties are orthogonal, you can set both to true, and the XML codec generates normalized fields with each kind of suffix. This allows you to use the same instance of the XML codec for XML elements that need sibling suffixing and XML elements that need twin suffixing. While this impacts memory usage according to the amount of XML data being normalized, you can specify mapping rules to filter for the fields of interest.

Sequence field example

Consider the following XML fragment:

```
<root>
   <prices instr="MSFT">
      <info>1.04</info>
      <info type="SELL">1.03</info>
   </prices>
   <prices instr="IBM">
      <info type="BUY"></info>
      <info type="SELL">1.06</info>
   </prices>
</root>
```

Suppose that the following properties are set in the XML codec section of the IAF configuration file:

```
<property name="XMLField" value="Test"/>
<property name="sequenceField" value="Test.root/prices/@instr"/>
<property name="sequenceField" value="Test.root/prices/info/@type"/>
<property name="sequenceField" value="Test.root/prices/info/text()"/>
<property name="ensurePresent" value="Test.root/prices/info/@type"/>
<property name="ensurePresent" value="Test.root/prices/info/text()"/>
<property name="separator:Test.root/prices" value="(prices)"/>
```

With these property values, the XML fragment maps to the following normalized event fields:

```
"Test.root/"                  =
"Test.root/prices#1/"         =
"Test.root/prices#1/info#1/"  =
"Test.root/prices#1/info#2/"  =
"Test.root/prices#2/"         =
"Test.root/prices#2/info#1/"  =
"Test.root/prices#2/info#2/"  =
"Test.root/prices/@instr"     = ["(prices)", "MSFT", "(prices)", "IBM"]
"Test.root/prices/info/@type" = ["(prices)", "", "SELL", "(prices)", "BUY", "SELL"]
"Test.root/prices/info/text()"= ["(prices)", "1.04", "1.03", "(prices)", "",    "1.06"]
```

If you define the following mapping rules in the IAF configuration file, you can map these normalized event fields to and from string fields in a sequence field of an Apama event.

```
<mapping-rules>
   <map transport="Test.root/prices/@instr"
        apama="instruments" type="reference"
        referencetype="sequence &lt;string&gt;" default="[]"/>
   <map transport="Test.root/prices/info/@type"
      apama="types" type="reference"
      referencetype="sequence &lt;string&gt;" default="[]"/>
   <map transport="Test.root/prices/info/text()"
      apama="prices" type="reference"
      referencetype="sequence &lt;string&gt;" default="[]"/>
</mapping-rules>
```

XPath examples

Consider the following XML:

```
<root>
   text1
   <a att="100.1">A text 1</a>
   <a>A text 2</a>
   <b att="300.0">
      <a att="400.4"/>
   </b>
   This is an interesting text string
</root>
```

Suppose that the following properties are set in the XML codec section of the IAF configuration file:

```
<property name="XMLField" value="Test"/>
<property name="XPath:Test->MyXPathResult.last-a"        value="*/a[last()]"/>
<property name="XPath:Test->MyXPathResult.first-att"     value="//@att"/>
<property name="XPath:Test->MyXPathResult.first-a-text"  value="/root/a[1]/text()"/>
<property name="XPath:Test->MyXPathResult.att>200"       value="//@att>200"/>
<property name="XPath:Test->MyXPathResult.att-count"     value="count(//@att)"/>
<property name="XPath:Test->MyXPathResult.text-contains"
            value="contains(/cdata-root/text()[last()], &quot;interesting&quot;)"/>
```

With these property values, the XML fragment maps to the following normalized event fields:

```
"MyXPathResult.last-a"             = "A text 2"
"MyXPathResult.first-att"          = "100.1"
"MyXPathResult.first-a-text"       = "A text 1"
"MyXPathResult.att>200"            = "true"
"MyXPathResult.att-count"          = "3"
"MyXPathResult.text-contains"      = "true"
```

The CSV codec plug-in

The CSV codec plug-in (`JCSVCodec`) translates between comma separated value (CSV) data and a sequence of string values. This codec (or the Fixed Width codec plug-in) can be used with the standard Apama File adapter to read data from files and to write data to files. (For more information on the Fixed Width codec, see "The Fixed Width codec plug-in" on page 160; for more information on the standard Apama File adapter, see .)

CSV format is a simple way to store data on a value by value basis. Consider an example CSV file that contains stock tick data. The lines in the file are ordered by Symbol, Exchange, Current Price, Day High, and Day Low, as follows:

```
TSCO, L, 395.50, 401.5, 386.25
MKS, L, 225.25, 240.75, 210.25
```

In this example, each field is separated from the next by a comma. You can use other characters as separators as long as you identify the separator character for the CSV codec.

To specify a separator character other than a comma, do one of the following:

- Send a configuration event from the transport that is communicating with the CSV codec using the method described in the topic, "Multiple configurations and the CSV codec" on page 159.
- Set the `separator` property in the IAF configuration file that you use to start the File adapter. For example:

```
<property name="separator" value=" " />
```

If you set the `separator` property, the codec uses the separator you specify by default. If you do not specify the `separator` property, and the codec does not receive any configuration events before receiving messages to encode or decode, the codec refuses to process messages. The codec throws an exception back to the module that called it, which is either the transport or the semantic mapper depending on whether the data is flowing downstream or upstream.

For an example configuration file, see `adapters\config\JCSVCodec-example.xml.dist` in the Apama installation directory. The `JCSVCodec-example.xml.dist` file itself should not be modified, but you can copy relevant sections of the XML code, modify the code as required for the purposes of your data, and then add the modified content to the adapter configuration file in which the codec is to be used.

Multiple configurations and the CSV codec

The CSV codec supports multiple configurations for interpreting separated data from different sources. A transport that is using the CSV codec can use the `com.apama.iaf.plugin.ConfigurableCodec` interface to set up different configurations for interpreting data from multiple sources that use different formats.

The transport can set a configuration by calling the following method on the codec:

```
public void addConfiguration(int sessionId,
                             NormalisedEvent configuration)
    throws java.io.IOException
```

The `sessionId` represents the ID value for this configuration.

The normalized event should contain the following key/value pair stored as strings that will be parsed in the codec:

Key	Value
`separator`	A string that contains the character to be used as the separator value, for example, "," or ";".

The transport can remove a configuration by calling the following method:

```
void removeConfiguration(int sessionId) throws java.io.IOException
```

The `sessionId` represents the ID value initially used to add the configuration with the `addConfiguration()` method.

Decoding CSV data from the sink to send to the correlator

To decode an event into a sequence of fields, the transport can then call:

```
public void sendTransportEvent(Object event, TimestampSet timestamps)
    throws CodecException, SemanticMapperException
```

The event object is assumed to be a `NormalisedEvent` instance. It must contain a key of `data`, which has a value of `string` type that contains the data to decode. That is, the `string` contains the line containing

the separated data. The codec then decodes the data, and stores the value from each field in a string sequence. This value from each field replaces the value for the data key.

If the event object also contains a `sessionId` key with an integer value associated with it, the value of the key identifies the configuration the codec uses to interpret the data. If the event does not contain a `sessionId`, the codec uses the default configuration as specified in the adapter configuration file.

Encoding CSV data from the correlator for the sink

Encoding CSV data works in the exact opposite way as decoding. The semantic mapper calls:

```
public void sendNormalisedEvent(NormalisedEvent event,
                                TimestampSet timestamps)
    throws CodecException, TransportException
```

The `sendNormalisedEvent()` method retrieves the data associated with the `data` key. The retrieved data is a sequence of strings, each of which contains the value of a field. The method then encodes the sequence into a single line to send to the transport so the transport can write the data to the sink. The CSV codec stores the result of the encoding in the `data` field. If the event contains a `sessionId` value, this is the configuration that the codec uses to encode the data. If the event does not contain a `sessionId`, the codec uses the default adapter configuration as specified in the adapter's configuration file initially used to start the adapter.

For a given event mapping in the IAF configuration file, it is not possible to dynamically specify the event decoder property, which identifies the codec that sends this event to the transport. Consequently, an adapter that is using several different codecs is unable to receive the same type of event from each codec. If it is necessary for your adapter to receive the same type of event from multiple codecs, set the event decoder property to the Null codec. This lets the transport receive the event and subsequently reroute the event back to the CSV codec by calling the following method:

```
sendNormalisedEvent(NormalisedEvent event, TimestampSet timestamps)
```

The CSV codec then returns the encoded data to the transport.

The Fixed Width codec plug-in

The Fixed Width codec plug-in (`JFixedWidthCodec`) translates between fixed width data and a sequence of string values. This codec (or the CSV codec plug-in) can be used with the standard Apama File adapter to read data from files and write data to files. (For more information on the CSV codec, see "The CSV codec plug-in" on page 158; for more information on the Apama File adapter, see .)

Fixed width data is a method of storing data fields in a packet or a line that is a fixed number of characters in size. Data stored in a fixed width format can be expressed by the following three parameters:

- The field widths used (that is, the number of characters used for storing each field)
- The padding character used if the data for a given field can be stored in less than the number of characters allocated for it
- Whether or not the data is left or right aligned within the field.

For example, consider the following, which describes a tick with ordered properties:

symbol	6 characters
exchange	4 characters
current price	9 characters
day high	9 characters
day low	9 characters

If the pad character is '-', an example of a left-aligned line is as follows:

```
TSCO--L---392.25---400.25---382.25---
```

The following is an example of a right-aligned line:

```
--TSCO---L---392.25---400.25---382.25
```

To specify fixed width data properties, do one of the following:

- Send a configuration event from the transport that is communicating with the Fixed Width codec using the method described in the topic, "Multiple configurations and the Fixed Width codec" on page 161.
- Set the fixed width properties in the IAF configuration file you use to start the adapter. For example, to obtain the left-aligned fixed width data above:

```
<property name="fieldLengths" value="[6,4,9,9,9]"/>
<property name="padCharacter" value="-"/>
<property name="isLeftAligned" value="true"/>
```

If you set all these properties, the codec uses them by default when decoding or encoding events.

If you do not set any of these properties, the codec expects to receive configuration events (as described in "Multiple configurations and the Fixed Width codec" on page 161), prior to receiving messages to encode or decode. Otherwise, the codec refuses to process these messages. The codec throws an exception back to the module that called it, which is either the transport or the semantic mapper depending on whether the data is flowing downstream or upstream.

If you require a default configuration, be sure to set all of these properties in the configuration file. If you set some of the properties, but not all of them, the codec cannot start.

For an example configuration file, see `adapters\config\JFixedWidthCodec-example.xml.dist` in the Apama installation directory. The `JFixedWidthCodec-example.xml.dist` file itself should not be modified, but you can copy relevant sections of the XML code, modify the code as required for the purposes of your data, and then add the modified content to the adapter configuration file in which the codec is to be used.

Multiple configurations and the Fixed Width codec

The Fixed Width codec supports multiple configurations for interpreting fixed width data from different sources. A transport that is using the Fixed Width codec can use the `com.apama.iaf.plugin.ConfigurableCodec` interface to set the configuration that you want the adapter to use.

The transport can set a configuration by calling the following method on the codec:

```
public void addConfiguration(int sessionId,
                             NormalisedEvent configuration)
   throws java.io.IOException
```

The *sessionId* represents the ID value for this configuration.

The normalized event should contain key/value pairs that are stored as strings the Fixed Width codec can parse.

Key	Value
fieldLengths	A string sequence that contains the number of characters each field value is stored in. For example, "[5,6,5,9"] where the first value is stored in the first 5 characters, the second value is stored in the next 6 characters, and so on.
isLeftAligned	"true" or "false", depending on whether data is left or right aligned in a field.
padCharacter	"_" where '_' is the pad character used when the data requires padding to fill the field.

The transport can remove a configuration by calling the following method:

```
void removeConfiguration(int sessionId) throws java.io.IOException
```

The `sessionId` represents the ID value initially used to add the configuration using the `addConfiguration()` method.

Decoding fixed width data from the sink to send to the correlator

To decode an event into a sequence of fields, the transport calls the `sendTransportEvent()` method as follows:

```
public void sendTransportEvent(Object event, TimestampSet timestamps)
   throws CodecException, SemanticMapperException
```

The event object is assumed to be a `NormalisedEvent`. It must contain the key '`data`', which has a value of string type containing the data to decode. That is, the line that contains the fixed width data. The Fixed Width codec then decodes the data and stores the value from each field in a string sequence.This value from each field replaces the value for the data key.

If the event also contains a `sessionId` key with an integer value associated with it, this is the configuration that the codec uses to interpret the data. If the event does not contain a `sessionId` the codec uses the default configuration as specified in the configuration file.

Encoding fixed width data from the correlator for the sink

Encoding fixed width data works in the exact opposite way to decoding. The semantic mapper calls:

```
public void sendNormalisedEvent(NormalisedEvent event,
      TimestampSet timestamps)
   throws CodecException, TransportException
```

This method retrieves the data associated with the `data` key. The data is in a string sequence where each member contains the value of a field. The method encodes the sequence members into a single line to send to the transport so the transport can write the data to the sink. Finally, the method stores the result of the encoding in the `data` field again.

If the event contains a `sessionId` value, this is the configuration that the codec uses to encode the data. If the event does not contain a `sessionId`, the codec uses the default File adapter configuration as specified in the File adapter configuration file initially used to start the file adapter.

For a given event mapping in the IAF configuration file, it is not possible to dynamically specify the event decoder property, which identifies the codec that sends the event to the transport. Consequently, an adapter that is using several different codecs is unable to receive the same type of event from each codec. If it is necessary for your adapter to receive the same type of event from multiple codecs, set the event decoder property to the Null codec. This lets the transport receive the event and subsequently reroute the event back to the Fixed Width codec by calling the following method:

```
sendNormalisedEvent(NormalisedEvent event, TimestampSet timestamps)
```

The Fixed Width codec then returns the encoded data to the transport.

Chapter 13: Apama File Adapter

This chapter provides a description of the Apama File adapter that is included when you install the Apama software. Apama is distributed with several other standard adapters. See the deployment guide for information about the following standard adapters:

- Apama Database Connector
- Apama Web Services Client

Each Apama standard adapter includes the transport and codec plug-ins it requires, along with any required EPL service monitor files. The C++ plug-ins are located in the Apama installation's `adapters\bin` directory (Windows) or `adapters/lib` directory (UNIX); the Java plug-ins are located in `adapters\lib`. The EPL files are located in the `adapters\monitors` directory.

If you develop an application in Apama Studio, when you add a standard adapter to the project, Apama Studio automatically creates a configuration file for it. In addition, the standard Apama adapters include bundle files that automatically add the adapter's plug-ins and associated service monitor files to the Apama Studio project.

If you are not using Apama Studio, you need to create a configuration file that will be used by the IAF to run the adapter. Each adapter includes a template file that can be used as the basis for the configuration file. The template files are located in the installation's `adapters\config` directory and have the forms *`adapter_name`*`.xml.dist` and *`adapter_name`*`.static.xml`. These template files are not meant to be used as the adapters' actual configuration files — you should always make copies of the template files before making any changes to them.

The Apama File adapter uses the Apama Integration Adapter Framework (IAF) to read information from text files and write information to text files by means of Apama events. This lets you read files line-by-line from external applications or write formatted data as required by external applications.

With some caveats, which are mentioned later in this section, the File adapter supports reading and writing to multiple files at the same time. Information about using the File adapter can be found in the following topics:

File Adapter plug-ins

The Apama File adapter uses the following plug-ins:

- `JMultiFileTransport.jar` — The `JMultiFileTransport` plug-in manages the connections to the files opened for reading and writing.
- `JFixedWidthCodec.jar` or `JCSVCodec.jar` — These plug-ins parse lines of data in fixed-width format or comma separated value format (CSV) into fields>

  For details about using these codec plug-ins, see "The CSV codec plug-in" on page 158 and "The Fixed Width codec plug-in" on page 160.
- `JNullCodec.jar`

These plug-ins need to be specified in the IAF configuration file used to start the adapter. If you add this adapter to an Apama Studio project, Apama Studio automatically adds these plug-ins to the configuration file. If you are not using Apama Studio, you can use the File.xml.dist template file as the basis for the configuration file. See "Configuring the File adapter" on page 167 for more information about adding the necessary settings to the adapter's configuration file.

Apama File Adapter

File Adapter service monitor files

The File adapter requires the event definitions in the following monitors which are in your Apama installation directory. If you are using Apama Studio, the project's default run configuration automatically injects them. If you are not using Apama Studio, you need to make sure they are injected to the correlator in the order shown before running the IAF.

1. `monitors\StatusSupport.mon`
2. `adapters\monitors\IAFStatusManager.mon`
3. `adapters\monitors\FileEvents.mon`
4. `adapters\monitors\FileStatusManager.mon`

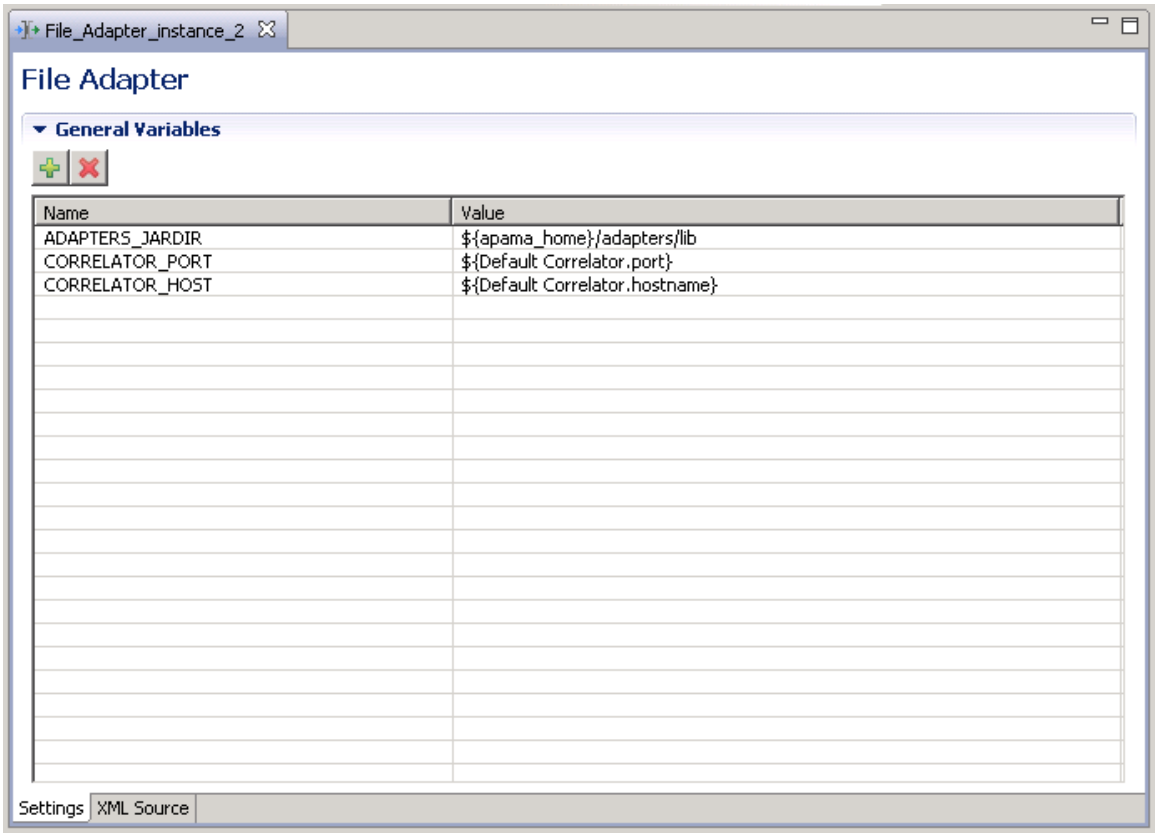
Apama File Adapter

Adding the File adapter to an Apama Studio project

If you are developing an application in Apama Studio, add the File adapter as follows:

1. In the Project Explorer, right-click name of the project and select Apama > Add Adapter. The **Add Adapter Instance** dialog is displayed.
2. Select File Adapter (File adapter for reading and writing to ASCII files). Apama Studio adds a default name to the Adapter instance name field that ensures this instance of this adapter will be uniquely identified. You can change the default name, for example, to indicate what type of external system the adapter will connect to. Apama Studio prevents you from using a name already in use.
3. Click OK.

   Apama Studio adds a File adapter entry that contains the new instance to the project's `Adapters` node and opens the instance's configuration file in the Apama Studio adapter editor as shown in the following illustration.



For the File adapter, the adapter editor's Settings tab displays a listing of General Variables. When first created it lists variables that are used in the Apama Studio project's default launch configuration. You can add variables by clicking the Add button and filling in the variable's name and value.

For editing other configuration properties for the File adapter, display the adapter editor's XML Source tab and add the appropriate information.

Apama File Adapter

Configuring the File adapter

Before using the File adapter, you need to add information to the IAF configuration file used to start the adapter. When you add an adapter to an Apama Studio project, a configuration file for each instance of the adapter is automatically created. In Apama Studio, double-click the name of the adapter instance to open the configuration file in the adapter editor.

If you are using Apama Studio's adapter editor, you can edit or add variables to the General Variables section as displayed on the Settings tab. For other properties, you need to edit the XML code directly; to do this, select the adapter editor's XML Source tab.

If you are not using Apama Studio, the configuration file can be derived from the template `adapters\config\File.xml.dist` configuration file shipped with the Apama installation. *Caution:* before changing any values, be sure to make a copy of the `File.xml.dist` file and give it a unique name, typically with an `.xml` extension instead of `.xml.dist`.

The template configuration file references the `adapters\config\File-static.xml` file using the XML `XInclude` extension. The `File-static.xml` file specifies the adapter's codec and its mapping rules.

Normally you do not need to change any information in this file. See "The IAF configuration file" on page 26 for more information on the contents on an adapter's configuration file.

In Apama Studio, adapters are configured using the adapter editor. To open an adapter instance, in the Project Explorer, right-click on *<project name>*`\Adapters\File Adapter\`*<instance name>* and select Open Instance from the pop-up menu.

You can set the variables used by the File adapter in the main Settings tab. Values of the form `${...}`, such as `${Default Correlator:port}` are set to the correct value automatically by the Apama Studio project's default launch configuration and do not need to be modified. To configure other properties used by the adapter, edit the XML code directly by selecting the XML Source tab.

If you are not using Apama Studio, all adapter properties are configured by editing the adapter `.xml` file in an XML or text editor.

Customize the following properties:

- `<logging level="INFO" file="logs/FileAdapter.log"/>`

  If you start the IAF from the Management and Monitoring console, specify an absolute path for the log filename.

  ```
  <classpath path="@ADAPTERS_JARDIR@/JNullCodec.jar" />
  <classpath path="@ADAPTERS_JARDIR@/JFixedWidthCodec.jar" />
  <classpath path="@ADAPTERS_JARDIR@/JCSVCodec.jar" />
  <classpath path="@ADAPTERS_JARDIR@/JMultiFileTransport.jar" />
  ```

  Replace `@ADAPTERS_JARDIR@` with the actual path to the `.jar` files. Typically, this is the *apama_install_dir*`\adapters\lib` directory.

  If you are using Apama Studio, these jar files are automatically added to the classpath in the configuration file and you do not need to replace the `@ADAPTERS_JARDIR@` token.

- In the `<sink>` and `<source>` elements, replace `@CORRELATOR_HOST@` and `@CORRELATOR_PORT@` with valid attribute values:

  ```
  <apama>
    <sinks>
       <sink host="@CORRELATOR_HOST@" port="@CORRELATOR_PORT@" />
    </sinks>
    <sources>
       <source host="@CORRELATOR_HOST@" port="@CORRELATOR_PORT@"
               channels="FILE" />
    </sources>
  </apama>
  ```

  If you are using the adapter in an Apama Studio project, the default launch configuration uses the default correlator host and port settings and you do not need to replace the @CORRELATOR_HOST@ and @CORRELATOR_PORT@ tokens.

- `<property name="simpleMode" value="false" />`

  Indicate whether or not to start the File adapter in simple mode. In simple mode, the File adapter reads lines from a single file or writes lines to a single file. In non-simple mode, you can use the fixed width or CSV codecs to decode/encode field data. Also, the File adapter can read/write to multiple files and additional controls are available for communication between the adapter and the correlator. Non-simple mode is recommended for most situations. Details about simple mode and non-simple mode are in the `File.xml.dist` file. If you are using Apama Studio, switch to the adapter editor's XML Source tab if you want to view these details or to edit the settings.

Apama File Adapter

Overview of event protocol for communication with the File adapter

The `adapters\monitors\FileEvents.mon` file defines the event types for communication with the File adapter. The following event types in the `com.apama.file` package are defined in the `FileEvents.mon` file. These events enable I/O operations on files. See `FileEvents.mon` for details about the events that are not described in the subsequent topics.

- `OpenFileForReading`
- `OpenFileForWriting`
- `FileHandle`
- `FileLine`
- `ReadLines`
- `NewFileOpened`
- `EndOfFile`
- `CloseFile`
- `FileClosed`
- `FileError`
- `LineWritten`

Apama File Adapter

Opening files for reading

The File adapter can read from multiple files at the same time. Send an `OpenFileForReading` event for each file you want the File adapter to read. This involves emitting an event to the channel specified in the adapter's configuration file, typically `FILE`, for example:

```
emit OpenFileForReading(...) to "FILE"
```

The `OpenFileForReading` event definition is as follows:

```
event OpenFileForReading
{  string   transportName;
   integer  requestId;
   string   codec;
   string   filename;
   integer  linesInHeader;
   string   acceptedLinePattern;
   dictionary<string, string> payload;
}
```

Parameter	Description
`transportName`	Name of the transport being used within the File adapter. This must match the transport name specified in the IAF configuration file so that the transport can recognize events intended for it.

`requestId`	Request identifier for this open file event. The response, which is either a `FileHandle` event or a `FileError` event, contains this identifier.
`codec`	Name of the codec to use with the file. This must match one of the codecs specified in the *adapter*`-static.xml` IAF configuration file. When you want the File adapter to read and write entire lines of data just as they are, specify the null codec (`JNullCodec`). When you want the File adapter to interpret file lines in some way, you can specify either the CSV codec (`JCSVCodec`) or the Fixed Width codec (`JFixedWidthCodec`) according to how the data in the file is formatted. To open fixed width or CSV files, you must add some information to the `payload` field of the `OpenFileForReading` event. The codecs needs this information to correctly interpret the data. For details about adding to the `payload` field, see "Opening comma separated values (CSV) files" on page 173 or "Opening fixed width files" on page 173.

`filename`	Absolute path, or file pattern (for example, `*.txt`, `*.csv`) within absolute directory path if intending to read all files matching a pattern in order of last time modified.While a relative path might work, an absolute path is recommended. A relative path must be relative to where the IAF has been started, which can be unpredictable. For example:
	`c:\logfiles\*.log`
	`/user/local/jcasablancas/logfiles/*.log`
	For more information, see "Specifying file names in OpenFileForReading events" on page 172.

`linesInHeader`	The number of lines in the header, or `0` if there is no header.Text files sometimes contain a number of lines at the beginning of the file that explain the format. As these are usually of some specific format, the Apama File adapter cannot interpret them. By skipping these lines, the File adapter can process just the data contained in the file.
`acceptedLinePattern`	Regular expression pattern (in the same format supported by Java) to use to match lines to read. The File adapter reads only those lines that match this pattern. To read all lines, specify an empty string.
`payload`	String dictionary for storing extra fields for use with codecs. For fixed width files the following fields make up the `payload`; for other types of files, they will be ignored. • `sequence<integer> fieldLengths` The length (number of characters) in each field, in order, where the number of fields is given by `fieldLengths.size()` • `boolean isLeftAligned`

	Whether the data in the field is aligned to the left or not (that is, right aligned) • `string padCharacter` The pad character used when the data is less than the width of the field For CSV files, the following field makes up the `payload`; for other types of files it will be ignored. • `string separator` The separator character

Apama File Adapter

Opening files for reading with parallel processing applications

If your Apama application implements parallel processing, you may want to increase parallelism by processing the incoming events from the File adapter in a separate, private, context, rather than doing everything in the correlator's main context. To request that events from the File adapter are sent to the private context your monitor is running in, the monitor should open the file using the `com.apama.file.OpenFileForReadingToContext` event instead of `OpenFileForReading`. The `OpenFileForReadingToContext` event has a field that contains a standard `OpenFileForReading` event (see "Opening files for reading" on page 169), in addition to a field specifying the context that file adapter events should go to for processing, (which is usually the context the monitor itself is running in, `context.current()`), and the name of the channel the File adapter is using. When using the `OpenFileForReadingToContext` event, the `OpenFileForReadingToContext` event and all other file adapter events must not be emitted directly to the adapter, but rather enqueued to the correlator's main context, where the adapter service monitor runs. The File adapter's service monitor is responsible for emitting the events that are enqueued from other contexts to the File adapter, and for enqueuing the events received from the File adapter to whichever context should process them (as specified in the `OpenFileForReadingToContext` event).

The OpenFileForReadingToContext event is defined as follows:

```
event OpenFileForReadingToContext
{
  context instanceContext;
  string fileChannel;
  OpenFileForReading fileEvent;
}
```

Here is an example of how the OpenFileForReadingToContext event is used:

```
com.apama.file.OpenFileForReading openFileForReading :=
    new com.apama.file.OpenFileForReading;
  ... // populate the fields of the openFileForReading event as needed
  // Instead of emitting openFileForReading to "FILE", wrap it in
  // the OpenFileForReadingToContext event and enqueue it to the service
  // monitor in the main context.
enqueue com.apama.file.OpenFileForReadingToContext(context.current(),
    "FILE", openFileForReading) to mainContext;
com.apama.file.FileHandle readHandle;
on com.apama.file.FileHandle(
  transportName=openFileForReading.transportName,
  requestId=openFileForReading.requestId):readHandle
{
```

```
  // instead of emitting to the "FILE" channel, enqueue it to the main
  // context
  enqueue com.apama.file.ReadLines(openFileForReading.transportName, -1,
    readHandle.sessionId, 20) to mainContext;
  ...
}
```

Opening files for reading

Specifying file names in OpenFileForReading events

In an `OpenFileForReading` event, the value of the `filename` field can be a specific file name or a wildcard pattern. However, the filename cannot have multiple wildcards.

Specific filename

When you specify a specific filename in an `OpenFileForReading` event, when the adapter receives requests to read lines from the file, the adapter reads till the end of the file and waits until more data is available. An external process, or the adapter itself, might write more data to the file if it is open for write at the same time that it is being read. If more data becomes available, the File adapter sends it. If the File adapter receives a `CloseFile` event, the File adapter closes the file against further reading.

Each time the File adapter reaches the end of the file it is reading, the File adapter sends an `EndOfFile` event to the correlator. If, during this process, more data was appended to the file, the file operations will continue as normal — that is, the File adapter will send more lines if they were requested. Thus, when reading specific files, file appends are acceptable and have a well defined behavior. However, any other modifications, such as changing the lines that have already been read, may have undefined results. An application can ignore or react to an `EndOfFile` event. The definition of an `EndOfFile` event is as follows:

```
event EndOfFile
{
   /* The name of the transport being used within the file adapter */
   string transportName;

   /* The session ID this File is associated with */
   integer sessionId;

   /* The name of the file*/
   string filename;
}
```

Wildcard filenames

Now suppose that in an `OpenFileForReading` event, the value of the `filename` field is a wildcard pattern. In this case, the adapter does the following:

1. Opens a new file that matches the pattern
2. Reads that file in its entirety
3. Sends back an `EndOfFile` event
4. Opens the next file that matches the pattern if one is available

For the application's information, the File adapter sends back an event when it opens each new file. The `NewFileOpened` event contains the name of the file that was opened:

```
event NewFileOpened
{
  /* The name of the transport being used within the file adapter */
```

```
  string transportName;

  /* The session Id this File is associated with */
  integer sessionId;

  /* The filename opened */
  string filename;
```

The order of opening the files that match the wildcard pattern is not specified. Currently, the files are ordered by the modification date and then alphabetically by filename.

If a file that has been previously read is externally modified (while in the meantime, the File adapter is reading from other files that match the wildcard pattern), the file is read again in its entirety. That is, any files that are modified after reading from them will be read again (until the `CloseFile` is sent). Please note that this includes file appends.

Apama File Adapter

Opening comma separated values (CSV) files

An example of defining an `OpenFileForReading` event that opens a CSV file so that each field is automatically parsed appears below. The additional data required by the CSV codec is stored in the `payload` dictionary.

```
com.apama.file.OpenFileForReading openCSVFileRead :=
      new com.apama.file.OpenFileForReading;

   //matches transport in IAF config
   openCSVFileRead.transportName := JMultiFileTransport;

   //the request id to use
   openCSVFileRead.requestId := integer.getUnique();

   //read using JCSVCodec
   openCSVFileRead.codec := "JCSVCodec";

   //file to read
   openCSVFileRead.filename := "/usr/home/formby/stocktick.csv";

   //separator char is a ","
   openCSVFileRead.payload["separator"] := ",";

   //emit event to channel in config.
   emit openCSVFileRead to "FILE";
```

Subsequently, when the File adapter receives `FileLine` events, the adapter stores each field in the data sequence in order. You can access the ones you are interested in.

For details about using the CSV codec, see "The CSV codec plug-in" on page 158.

Apama File Adapter

Opening fixed width files

An example of defining an `OpenFileForReading` event that opens a fixed width file so that each field is automatically parsed appears below. The additional data required by the Fixed Width codec is stored in the `payload` dictionary.

```
com.apama.file.OpenFileForReading openFixedFileRead :=
      new com.apama.file.OpenFileForReading;
```

```
   //matches transport in IAF instance
   openFixedFileRead.transportName := JMultiFileTransport;

   //the request id to use
   openFixedFileRead.requestId := integer.getUnique();

   //read using CSV Codec
   openFixedFileRead.codec := "JFixedWidthCodec";

   //file to read
   openFixedFileRead.filename := "/usr/home/formby/stocktick.txt";
   //additional data required to interpret fixed width data

   //sequence of field lengths
   openFixedFileRead.payload["fieldLengths"] := "[6,4,9,9,9]";

   //it is left aligned
   openFixedFileRead.payload["isLeftAligned"] := "true";

   //the pad character
   openFixedFileRead.payload["padCharacter"] := "_";

   //emit event to channel in config.
   emit openFixedFileRead to "FILE";
```

Subsequently, when the File adapter receives `FileLine` events, the adapter stores each field in the data sequence in order. You can access the ones you are interested in.

For details about using the Fixed Width codec, see "The Fixed Width codec plug-in" on page 160.

Apama File Adapter

Sending the read request

After you construct an `OpenFileForReading` event, emit it to the `"FILE"` channel. For example:

```
com.apama.file.OpenFileForReading openFileWeWantToRead :=
      new com.apama.file.OpenFileForReading;

   //populate the openFileWeWantToRead event
   //..
   //..
   emit openFileWeWantToRead to "FILE";
```

Emitting an `OpenFileForReading` event from EPL code signals the File adapter to open the file. If the open operation is successful, the File adapter returns a `FileHandle` event, whose definition is as follows:

```
event FileHandle
{
     /* The name of the transport being used within the file adapter */
   string transportName;

     /* Request ID this file handle is in response to. */
   integer requestId;

     /* Session ID to use for further communication with the
        File adapter */
   integer sessionId;
}
```

The `sessionId` is the most important field; all communication related to this file references this value.

If the open operation is unsuccessful, the File adapter returns a `FileError` event, whose definition is as follows:

```
event FileError
```

```
{
     /* The name of the transport being used within the file adapter */
   string transportName;

     /* Request ID this file error is in response to. */
   integer requestId;

     /* This should contain relevant information as to why the
        error occurred */
   string message;
}
```

Apama File Adapter

Requesting data from the file

After your application receives a `FileHandle` event, it can emit a `ReadLines` event, which signals the adapter to start reading lines from the file. The definition of the `ReadLines` event is as follows:

```
event ReadLines
{
  /* The name of the transport being used within the file adapter */
  string transportName;

  /* request Id */
  integer requestId;

  /* The session Id this read line event is coming from */
  integer sessionId;

  /* The number of lines to read. As each line is read, a FileLine
     event will be sent from the Adapter in response to this request.
     When opening a single file, the adapter will continue to probe
     the file until lines are available.

     If a wildcard pattern was specified and the end of file is
     reached before the specified number of lines have been read,
     the file will be closed, an EndOfFile event will be sent, and
     if a new file is available, it will send a NewFileOpened event
     and read the remaining number of lines from this. If a new file
     is not available, the adapter will wait until one is available.
  */
  integer numberOfLines;
}
```

The `sessionId` in the `ReadLines` event must be the same as the `sessionId` stored in the `FileHandle` event that the application received when the file was opened.

The adapter tries to read as many lines as specified in the `ReadLines` event. If the file does not contain that many lines, what the adapter does depends on whether the original `OpenFileToRead` event specified a specific file or a wildcard pattern. According to that setting, the adapter either waits until the file contains more data, or tries to open a new file to deliver the balance from.

Apama File Adapter

Receiving data

As the File adapter reads the file, it returns `FileLine` events to your application. Each line is associated with a specific `sessionId`, and the data is stored within a sequence of strings. The definition of `FileLine` events is as follows:

```
event FileLine
```

```
{
  /* The name of the transport being used within the file adapter */
  string transportName;

  /* When receiving these events in response to read line events,
     this field can be ignored, as the protocol does not
     distinguish between ReadLine requests for the same session.*/
  integer requestId;

  /* The session Id this FileLine event is for or from */
  integer sessionId;

  /* The data as a sequence of strings. */
  sequence<string> data;
}
```

Notice that the data field is a sequence of strings, rather than a string. However, when you use the null codec for reading, the sequence contains only one element, which contains the entire line read:

```
//the whole line is stored in the first element, we used null codec
string line := fileLine.data[0];
```

For specialized codecs, each field is in a discrete element in the sequence:

```
//The app knows which field contains the data we are interested in:
string symbol := fileLine.data[0];
string exchange := fileLine.data[1];
string currentprice := fileLine.data[2];
//and so on
```

After the File adapter opens a file for reading, the file remains open as long as the adapter is running. If you want to close a file, you must send a `CloseFile` event that specifies the `sessionId` of the file you want to close. For example, if you want to replace the contents of a file, you must close the file before you send an `OpenFileForWriting` event. The definition of the `CloseFile` event is as follows:

```
event CloseFile
{
  /* The name of the transport being used within the file adapter */
  string transportName;
  /* Request ID for this CloseFile event. If an error occurs,
     the response will contain this ID */
  integer requestId;

  /* The session ID the file to close is associated with */
  integer sessionId;
}
```

If there is an error, the File adapter sends a `FileError` event. Otherwise, the File adapter closes the file and sends a FileClosed event, and then it is available to be opened again for writing or for reading.

Apama File Adapter

Opening files for writing

To open a file for writing, emit an `OpenFileForWriting` event. The definition of the `OpenFileForWriting` event is similar to the definition of the `OpenFileForReading` event:

```
event OpenFileForWriting
{
  /* The name of the transport being used within the file adapter.
     This should match the transport name used in the IAF config
     file so the transport can recognize events intended for it */
  string transportName;

  /* Request ID for this open file event. The response, either a
     FileHandle or a FileError event, will contain this ID */
```

```
  integer requestId;

  /* The name of the codec to use with the file. This should match
     one of the codecs specified in the (static) config file. Use
     the null codec (by default this is called JNullCodec) to write
     entire lines */
  string codec;

  /* Full filename to write to */
  string filename;

  /* Boolean representing whether the file is to be overwritten or
     appended */
  boolean appendData;

  /* This field is used to specify extra parameters to the codecs,
     such as the CSVCodec and the FixedWidthCodec */
  dictionary<string, string> payload;
}
```

The procedure for opening a file for writing CSV or fixed width files is effectively the same as for reading. Specify the relevant fields in the payload to describe the format of the file you want to write. When subsequently sending `FileLine` events, populate the data sequence field with the data for each field.

Again, once constructed, emit the `OpenFileForWriting` event to the `"FILE"` channel, for example:

```
emit new com.apama.file.OpenFileForWriting("FileTransport",
     integer.getUnique(), "JNullCodec","/home/writeFile.txt",false);
```

For fixed width files, you can construct a more complex `OpenFileForWriting` event in a similar way to that described in "Opening fixed width files" on page 173.

Again, as with reading a file, the File adapter sends a `FileHandle` or `FileError` event (see "Sending the read request" on page 174), which your application should listen for, filtering on the `requestId` for the `FileHandle` event you are interested in.

Once a `FileHandle` event has been received, the file has successfully opened and the application can begin to send `FileLine` events to be written:

```
event FileLine
{
  /* The name of the transport being used within the file adapter */
  string transportName;

  /* When sending these upstream (i.e. writing):
     1) If this value is negative, the adapter will send no
        acknowledgement.
        A FileError may be sent back in response if this request
        generates an error. If the user is always using the same
        requestId e.g. -1, then they will not be able to work out
        which line generated the error. The application writer is
        free to ignore listening for these errors should they wish.

     2) If this value is 0 or greater then the adapter will send an
        acknowledgement LineWritten event back to the application.
  */
  integer requestId;

  /* The session Id this FileLine event is for or from */
  integer sessionId;

  /* The data as a sequence of strings. */
  sequence<string> data;
}
```

Notice that the data field is a sequence of strings, rather than a string. This allows you to have the fields you want to write as separate entries in the sequence, and it lets the File adapter format

the sequence for writing according to the chosen codec. For the fixed width codec, the number of elements in the sequence should match the number of fields originally specified when opening the file. For the null codec, if the sequence contains more than one element, each field will be written out using a separator defined in the IAF configuration file. This separator can be blank, in which case each element will be written out immediately after the previous one, with a newline after the last element.

The `FileLine` event is exactly the same as the one received when reading; however, the `requestId` takes on a more important role. If you specify a positive `requestId`, your application receives an acknowledgement

When a file is already open for reading, you can write to that file only by appending new data. Of course, you must send an `OpenFileForWriting` event, and then the File adapter can process `FileLine` events for writing to that file. You receive a `FileError` event if the file is open for reading and for writing and you try to write data into the file but not by appending the new data.

Apama File Adapter

Opening files for writing with parallel processing applications

If your Apama application implements parallel processing, you may want to increase parallelism by processing the incoming events from the File adapter in a separate, private, context, rather than doing everything in the correlator's main context. To request that events from the File adapter are sent to the private context your monitor is running in, the monitor should open the file using the `com.apama.file.OpenFileForWritingToContext` event instead of `OpenFileForWriting`. The `OpenFileForWritingToContext` event has a field that contains a standard `OpenFileForWriting` event (see "Opening files for writing" on page 176), in addition to a field specifying the context that file adapter events should go to for processing, (which is usually the context the monitor itself is running in, `context.current()`), and the name of the channel the File adapter is using. When using the `OpenFileForWritingToContext` event, the `OpenFileForWritingToContext` event and all other File adapter events must not be emitted directly to the adapter, but rather enqueued to the correlator's main context, where the adapter service monitor runs. The File adapter's service monitor is responsible for emitting the events that are enqueued from other contexts to the File adapter, and for enqueuing the events received from the File adapter to whichever context should process them (as specified in the `OpenFileForWritingToContext` event).

The `OpenFileForWritingToContext` event is defined as follows

```
event OpenFileForWritingToContext
{
  context instanceContext;
  string fileChannel;
  OpenFileForWriting fileEvent;
}
```

Using the `OpenFileForWritingToContext` event is similar to using the `OpenFileForReadingToContext` event. See "Opening files for reading with parallel processing applications" on page 171 for an example use of the `OpenFileForReadingToContext` event.

Opening files for writing

LineWritten event

After the File adapter writes a line to a file, the adapter sends a `LineWritten` event. The event definition is as follows:

```
event LineWritten
{
  /* The name of the transport being used within the File adapter */
  string transportName;

  /* Request ID this event is in response to. */
  integer requestId;

  /* The sessionId the FileLine was sent for */
  integer sessionId;
}
```

This is useful when you want your application to send `FileLine` events in a batch to control flow. If you need to do flow control, you would typically set all the `requestIds` to *positive* values and emit the next `FileLine` events only after receiving the `LineWritten` notification for the previous `FileLine` event you sent. If you do *not* need to do flow control, you could set `requestId=-1` for all but the last `FileLine` event, but set it to a positive value for the very last `FileLine` event so you get a single `LineWritten` notification when everything has been written.

The file remains open for the lifetime of the adapter unless you emit a `CloseFile` event. See "Opening files for reading" on page 169 for the CloseFile event definition.

Apama File Adapter

Monitoring the File adapter

You can use the File adapter status manager (`FileStatusManager.mon` in the `adapters\monitors` directory) to monitor the state of the File adapter.

The File adapter sends status events to the correlator, some of which are asynchronous (not requested) status messages. This occurs as a result of connection status changes, which happen in response to a file being closed or opened.

For single files, the File adapter sends an `AdapterConnectionOpenedEvent` when it opens a new file for reading or writing, and an `AdapterConnectionClosedEvent` when it closes a file. When the File adapter uses a wildcard pattern to open a series of files, in addition to those events, the File adapter sends an `AdapterConnectionClosedEvent` event after it has read everything in a file, and an `AdapterConnectionOpenedEvent` event when it opens the next file. This is an analogous pattern to the `EndOfFile` and `NewFileOpened` events sent by the adapter itself.

Apama File Adapter