TIBCO Hawk®

Programmer's Guide

Software Release 6.2
September 2019

Important Information

Contents

Preface

This manual is intended for use by programmers.

It contains detailed description of Hawk Console API, Hawk Configuration Object API, and Hawk Application Management Interface (AMI) API.

This manual assumes you are familiar with TIBCO Rendezvous architecture, and the concepts of system monitoring.

Topics

Changes from the Previous Release of this Guide

Removed information about the TIBCO DataGrid transport.

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Hawk Documentation

The following documents form the TIBCO Hawk documentation set:

- *TIBCO Hawk Release Notes:* Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.
- *TIBCO Hawk Concepts:* This manual includes basic descriptions of TIBCO Hawk concepts.
- *TIBCO Hawk Installation, Configuration, and Administration:* Read this book first. It contains step-by-step instructions for installing TIBCO Hawk software on various operating system platforms. It also describes how to configure the software for specific applications, once it is installed. An installation FAQ is included.
- *TIBCO Hawk Methods Reference*: A reference to the microagents and methods used by a TIBCO Hawk Agent for system and application monitoring.
- *TIBCO Hawk WebConsole User's Guide:* This manual includes complete instructions for using TIBCO Hawk WebConsole.
- *TIBCO Hawk Programmer's Guide*: All programmers should read this manual. It contains detailed descriptions of Application Management Interface (AMI), Application Programming Interface (API) concepts, and the TIBCO Hawk security framework and its classes. It also contains detailed descriptions of each class and method for the following APIs:

  — AMI API

    Java, C++ and C API

  — Console API

    Java API

  — Configuration Object API

    Java API

  Programmers should refer to the appropriate language reference sections for the AMI API details. The TIBCO Hawk Application Management Interface (AMI) exposes internal application methods to TIBCO Hawk.

- *TIBCO Hawk Plug-in Reference Guide:* Contains details about the Enterprise Message Service, Messaging and JVM microagents methods that are used to administer and monitor the TIBCO Enterprise Message Service server.
- *TIBCO Hawk Plug-ins for TIBCO Administrator:* Contains detailed descriptions of the TIBCO Hawk plug-ins accessed via TIBCO Administrator.
- *TIBCO Hawk HTTP Adapter User's Guide:* Contains information about performing discovery, monitoring of agent status, monitoring of agent alerts, method invocation, method subscription, and many more activities on TIBCO Hawk and third-party products.
- *TIBCO Hawk Admin Agent Guide:* Contains basic configuration details for TIBCO Hawk Admin Agent and complete instructions for using the web interface of TIBCO Enterprise Administrator for TIBCO Hawk.
- *TIBCO Hawk Security Guide:* Provides guidelines to ensure security within the components of TIBCO Hawk and within the communication channels between the components.

Other TIBCO Product Documentation

You may find it useful to read the documentation for the following TIBCO products:

- TIBCO® Enterprise Administrator
- TIBCO ActiveSpaces®
- TIBCO Rendezvous®
- TIBCO Enterprise Message Service™

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
ENV_HOME *TIBCO_HOME* *HAWK_HOME* *CONFIG_FOLDER*	TIBCO products are installed into an installation environment. A product installed into an installation environment does not access components in other installation environments. Incompatible products and multiple instances of the same product must be installed into different installation environments. An installation environment consists of the following properties: • **Name** Identifies the installation environment. This name is referenced in documentation as *ENV_NAME*. On Microsoft Windows, the name is appended to the name of Windows services created by the installer and is a component of the path to the product shortcut in the Windows Start > All Programs menu. • **Path** The folder into which the product is installed. This folder is referenced in documentation as *TIBCO_HOME*. TIBCO Hawk installs into a directory within a *TIBCO_HOME*. This directory is referenced in documentation as *HAWK_HOME*. The default value of *HAWK_HOME* depends on the operating system. For example on Windows systems, the default value is `C:\tibco\hawk\6.0`. A TIBCO Hawk configuration folder stores configuration data generated by TIBCO Hawk. Configuration data can include sample scripts, session data, configured binaries, logs, and so on. This folder is referenced in documentation as *CONFIG_FOLDER*. For example, on Windows systems, the default value is `C:\ProgramData\tibco\cfgmgmt\hawk`.
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.

Table 1 General Typographical Conventions (Cont'd)

Convention	Use
`bold code font`	Bold code font is used in the following ways: • In procedures, to indicate what a user types. For example: Type **`admin`**. • In large code samples, to indicate the parts of the sample that are of particular interest. • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, `MyCommand` is enabled: MyCommand [**`enable`** | disable]
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO BusinessWorks Concepts*. • To introduce new terms For example: A portal page may contain several portlets. *Portlets* are mini-applications that run in a portal. • To indicate a variable in a command or code syntax that you must replace. For example: `MyCommand` *pathname*
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
📝	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.
💡	The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.
⚠	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

TIBCO Product Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the TIBCO Product Documentation website mainly in the HTML and PDF formats.

The TIBCO Product Documentation website is updated frequently and is more current than any other documentation included with the product. To access the latest documentation, visit https://docs.tibco.com.

Documentation for TIBCO Hawk is available on the TIBCO Hawk Product Documentation page.

How to Contact TIBCO Support

You can contact TIBCO Support in the following ways:

- For an overview of TIBCO Support, visit https://www.tibco.com/services/support.
- For accessing the Support Knowledge Base, viewing the latest product updates that were not available at the time of the release, and getting personalized content about products you are interested in, visit the TIBCO Support portal at https://support.tibco.com.
- For creating a Support case, you must have a valid maintenance or support contract with TIBCO. You also need a user name and password to log in to https://support.tibco.com. If you do not have a user name, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. In addition, users can submit and vote on feature requests from within the TIBCO Ideas Portal. For a free registration, go to https://community.tibco.com.

Chapter 1 Introduction to TIBCO Hawk Programming

TIBCO Hawk software monitors distributed systems and applications.

You can interact with TIBCO Hawk applications through the TIBCO Hawk Console API or through TIBCO Hawk AMI APIs.

Topics

Programming Tools and Interfaces

REST API

TIBCO Hawk REST API provide set of HTTP methods to manage and monitor the Hawk agent. You can use these methods to create your own console, if needed.

After you start the Hawk Console, you can view these REST APIs at the Swagger page URL

`http://`*<Console_host_IP>:<Host_port>*`/HawkConsole/v1/docs`

The Swagger page enables you to view the details of each API, such as model schema, required parameters, response messages, and so on. Also, you can try out the APIs by providing the parameters to the API in the Swagger URL.

Console API

The Console API is a comprehensive set of Java interfaces that allow you to manage and interact with TIBCO Hawk agents and monitor alerts generated by these agents. Both the TIBCO Hawk WebConsole and TIBCO Hawk Event Service implement the Console API to monitor and manage agent behavior. Programmers can use the Console API to write custom applications similar to these applications to monitor agent behavior, subscribe to alert messages, and invoke microagent methods.

Configuration Object API

The Configuration Object API is a Java interface for writing custom rulebases. Rulebases are used by TIBCO Hawk agents to monitor and manage systems and applications. The Configuration Object API provides classes to define rules, tests and actions. Instances of these classes are put together to define a new rulebase.

AMI API

The AMI API allows application to instrument their applications with the Hawk API and make them manageable using Hawk Agent. AMI APIs are available in Java, C and C++.

Security API

The TIBCO Hawk Security API is used to build security plug-in modules used for secure agent and console interactions. The security mechanism actually involves two modules, one that is used by the agent and another that is used by the console. If you use the Console API to write console applications that will operate in a secure TIBCO Hawk environment, you must have access to the console-side security plug-in class to be able to perform management operations on agents. The TIBCO Hawk WebConsole requires a security plug-in class to manage agents in a secure environment.

Chapter 2 Console API

The TIBCO Hawk Console API is a Java language interface for writing programs that can perform monitoring and management of TIBCO Hawk agents. It provides all the facilities required to perform discovery, monitoring of agent-alive status, monitoring of agent alerts, and the monitoring of agent configuration changes. It also allows you interact with agents and their managed objects through remote method invocations on microagents.

This chapter provides a brief overview of the major components of the TIBCO Hawk Enterprise Monitor and how the TIBCO Hawk Console API relates to them.

Topics

How the TIBCO Hawk Console API Fits In

The TIBCO Hawk Console API allows you to write programs that can monitor the alerts generated by TIBCO Hawk agents and perform management operations on their microagents. Applications using this API can be referred to as console applications.

Both the TIBCO Hawk WebConsole and the TIBCO Hawk Event Service use the TIBCO Hawk Console API to monitor or manage agents, or both. They are thus both good examples of the types of applications that can be built with this interface.

Applications built with the Console API can monitor every node within a given TIBCO Hawk domain. However, additional instances of console applications can run simultaneously on multiple nodes in the network with little or no additional network impact.

The TIBCO Hawk Console API does not provide facilities for the creation of rulebases.

The TIBCO Hawk software distribution includes sample Console API programs that helps you to better understand how to use the Console API. These samples can be found in the following directory:

HAWK_HOME`/examples/console_api`

These Console API sample applications use the hawk_example.props property file that is located at :

HAWK_HOME`/examples`

Monitoring Component

The monitoring component of the TIBCO Hawk Console API provides events that notify your application when:

- Agents are discovered
- Agents expire
- Alerts and clears are generated by an agent's rules engine
- An agent's list of microagents changes
- An agent's list of rulebases changes

Management Component

The management component allows you to interact with an agent's microagents by:

- Invoking their methods to return management data or perform management functions
- Subscribing to their management data in a way that produces an updating data stream of management information
- Performing group operations on multiple Microagents across the network simultaneously

Concepts

The following sections provides some background information on using the classes.

- Structure, page 9
- Monitoring Operations, page 9
- The AgentInstance Class, page 12
- Management Operations, page 13
- Management Operations, page 13
- Microagent Descriptors, page 14
- Invoking Methods, page 15
- Subscribing to Method Results, page 16
- Group Operations, page 17
- Lightweight Console, page 18
- MicroAgent Plug-in, page 19

Structure

The TIBCO Hawk Console API is partitioned along the functional lines of monitoring and management. At the root is the `TIBHawkConsole` class from which you can access the `AgentMonitor` and `AgentManager`, which encompass the monitoring and management facilities of the API.

Monitoring Operations

The TIBCO Hawk Console API allows you to monitor all agents simultaneously from a single program.

For each agent, the APIs monitoring capabilities can be decomposed into four areas:

- Management Operations
- Microagent List Monitoring
- Rulebase List Monitoring
- Alert Monitoring

All monitoring is performed using event notification. Listeners are registered with the `AgentMonitor` for the type of monitoring required and the relevant status messages are delivered as events when they are detected.

A registered listener for a particular type of event will receive notifications for all monitored agents that generate that event. For example, registering a `AgentMonitorEventLister` with the console will allow it to receive `AgentMonitorEvents` for all agents known to the console, thus allowing it to track the alive or expired status of every agent.

Agent-Alive Monitoring

The most basic type of monitoring is the monitoring of agent existence. This is achieved by registering an `AgentMonitorListener` with the `AgentMonitor`. When you register an `AgentMonitorListener`, it will receive one agent-alive event for every agent the console can detect.

An agent-alive event is actually represented by the delivery of an `AgentMonitorEvent` to the on`AgentAlive()` method of the listener. This event contains a reference to an `AgentInstance` object which identifies which agent it pertains to. As new agents appear in the network, new agent-alive events will be generated to identify them. When the console is no longer able to communicate with an agent, it will issue an agent-expired event for that agent. This is performed by delivering a `AgentMonitorEvent` to the on`AgentExpired()`

method of the listener. As before, this event contains a reference to an `AgentInstance` object which identifies which agent it pertains to. The console can lose communication with an agent for several reasons, for example if the agent process is no longer running, the machine it was running on has crashed, or because of a problem in the underlying communications infrastructure such as a network outage.

Microagent List Monitoring

Each agent contains a collection of objects called microagents. Microagents have methods through which monitoring and management is performed. Microagents represent managed entities such as the operating system's subsystems, log files, event logs, or applications. A newly launched application instrumented with AMI will dynamically appear as a microagent on its managing agent (by default, the one located on the same processor). When the instrumented application terminates, the corresponding microagent will also be removed.

Microagent list monitoring is used to track the dynamic list of microagents in an agent. Events are delivered when microagents are added and removed.

To perform this type of monitoring you simply register a `MicroAgentListMonitorListener` with the `AgentMonitor`. Events of type `MicroAgentListMonitorEvent` are then delivered to either the `onMicroAgentAdded()` or `onMicroAgentRemoved()` method of the listener. The event contains an `AgentInstance` object, which identifies the agent, and a `MicroAgentID` object, which identifies the microagent.

Rulebase List Monitoring

Rulebases direct the monitoring activities of an agent. Each rulebase is a collection of rules which are usually grouped together to monitor an application or system resource. An agent may have more than one loaded rulebase and this list can change dynamically.

Rulebase list monitoring is structurally similar to microagent list monitoring. It is used to monitor the dynamically changing list of rulebases that are loaded on an agent.

To perform this type of monitoring you register a `RuleBaseListMonitorListener` or `onRulebaseUpdated` with the `AgentMonitor`. Events of type `RuleBaseListMonitorEvent` are then delivered to either the `onRuleBaseAdded()` or `onRuleBaseRemoved()` method of the listener. The event contains an `AgentInstance` object, which identifies the agent, and a `RuleBaseStatus` object, which is used to identify the rulebase being added or removed.

The `onRulebaseUpdated()` listener provides an atomic update callback for rulebase update events. To receive `onRulebaseUpdated()`, implement `ExtendedRuleBaseListMonitorListener` as per the definition provided in API Reference, page 26 section.

Alert Monitoring

One of the actions a rulebase can take is to generate an alert, which is usually done to signal that a problem condition has been detected. Alerts contain a state, to indicate the problem severity, and text, which provides the problem description. If the alert condition, as defined by the rulebase, ceases to exist, the rulebase will generate a clear against that alert.

Every alert is associated with a particular rulebase and every rulebase is associated with a particular agent. (Technically, rulebases are said to be associated with the `RuleBaseEngine` microagent instance registered with that agent, therefore the association with the agent is indirect.) The alert state of a rulebase will always be highest alert state of all the active (non-cleared) alerts associated with it. The alert state of an agent (or its `RuleBaseEngine` microagent) is the highest alert state of any of its currently loaded rulebases. Therefore, alerts and clears can potentially change the state of the rulebase and agent that generated them.

Alert monitoring is performed by registering an `AlertMonitorListener` with the `AgentMonitor`. Alerts and clears generated by an agent are detected by the console and delivered as `AlertMonitorEvent` objects to the listener. Since `AlertMonitorEvent` is an abstract class, one of its concrete subclasses is delivered: `PostAlertEvent` for alerts and `ClearAlertEvent` for clears.

Every `AlertMonitorEvent` contains flags that indicate whether it caused a change in the state of the `RuleBaseEngine` or the rulebase it is associated with. It also holds the current states of those objects after taking into account the current event. Other attributes include an alert ID, which uniquely identifies an alert, and the time the event was generated.

Because `AlertMonitorEvent` extends `AgentMonitorEvent`, it also has an `AgentInstance` attribute to identify the agent that generated the event.

The `PostAlertEvent` class extends `AlertMonitorEvent` and adds the state of the alert and the alert text. The `ClearAlertEvent` class extends `AlertMonitorEvent` and adds the reason-cleared text. This string identifies the reason the alert was cleared.

All alert states are represented as integer values that are mapped using the `AlertState` interface. A `PostAlertEvent` with a state of `AlertState.NO_ALERT` is also called a notification.

The AgentInstance Class

Common to all monitoring events, and that which relates them to each other, is the agent instance they pertain to. This agent instance is represented in the TIBCO Hawk Console API by the `AgentInstance` class. The following definitions and discussion describe this class, the related `AgentID` class, and their role in monitoring.

Agent ID

The agent ID specifies the attributes of an agent process that are used to uniquely identify it. These attributes are

- Agent name
- Agent domain
- TIBCO Hawk domain name.

Agent Instance

The agent instance is an instantiation of a agent; the actual agent process running on some machine in the network. If an agent is restarted, the new process represents a new agent instance. Agent instances have a life span that begins with the starting of an agent process and ends with its termination. Agent instances are represented by the `AgentInstance` class.

Ensuring Each Agent Instance has a Unique ID

In order for the TIBCO Hawk system to function correctly, every concurrently running agent instance must have a unique agent ID. The console will attempt to detect violations of this condition and produce a warning. However, this may not be detectable in all circumstances and so the system administrator must insure that every agent has a unique agent ID.

If a console client is interested in tracking agent status across agent instances or console agent instances then it may do so using the console agent ID, that is, the `AgentID` class. If a agent is restarted several times (without changing its ID attributes) then this will result in the creation of several distinct instances of `AgentInstance` in the console. These different instances, however, will all have equal `AgentID` values to reflect the fact that they all relate to the same logical agent. Note that these multiple instances of `AgentInstance` will never be referenced simultaneously by the console. This is because the console will never hold two `AgentInstance` objects with equal `AgentID` attributes at any one time.

Management Operations

Management functions are performed by invoking management operations on microagents. As described in Microagent List Monitoring on page 10, each agent contains a collection of objects called microagents. Microagents have methods through which all monitoring and management is performed. Microagents represent managed entities such as the operating system's subsystems, log files, event logs, applications, and even the agent itself.

The `AgentManager` class is used to interact with microagents. An instance of this class is obtained by invoking the `getAgentManager()` method of the `TIBHawkConsole` class.

The first step in interacting with a microagent is to obtain its descriptor. This descriptor enumerates the available methods and describes their signature and their return types. With this information you can then invoke methods and subscribe to method results for an individual microagent. You can also perform a group operation that simultaneously performs a method invocation on multiple instances of the same microagent across multiple agents.

Microagent Descriptors

Descriptors are represented by the `MicroAgentDescriptor` class. They are obtained by invoking the `describe()` method of the `AgentManager` class. A `MicroAgentDescriptor` fully describes a microagent. It contains a list of `MethodDescriptor` objects that describe each available method and all that is needed to invoke them.

The methods of a microagent are divided into three categories depending on their impact. These categories are enumerated by the following static variables of the `MethodDescriptor` class:

- **`IMPACT_ACTION`**: Methods of type `IMPACT_ACTION` take some action that can potentially change the state of the managed object represented by the microagent.
- **`IMPACT_INFO`**: Methods of type `IMPACT_INFO` simply retrieve some information in a manner that does not change the state of the managed object.
- **`IMPACT_ACTION_INFO`**: Methods of type `IMPACT_ACTION_INFO` return data but may also change the sate of the managed object.

The `describe()` method of the `AgentManager` requires an argument of type `MicroAgentID`. There are two general ways to obtain `MicroAgentID` objects:

- If your application is also performing monitoring then the `MicroAgentIDs` for microagents loaded on a particular agent can be obtained from the method `AgentInstance.getStatusMicroAgents()`. A registered `MicroAgentListMonitor` object also receives `MicroAgentIDs` in the events delivered to it.
- The method `AgentManager.getMicroAgentID()` can also be used to obtain `MicroAgentID` objects. It takes a microagent name as its argument and returns an array of all microagents of that name that are currently loaded on all agents the console can communicate with. This method blocks for a period of time while it queries the agents on the network. A second version of this method accepts an integer that indicates the minimum number of desired microagent ids in the return. This method generally returns more quickly than the first version if the minimum number specified is less than or equal to the actual number of matching microagents on the network.

Invoking Methods

Microagent methods are invoked using the `invoke()` method of the `AgentManager`. This method takes a `MicroAgentID` and a `MethodInvocation` object. The `MicroAgentID` may be obtained as described in Microagent Descriptors on page 14. The `MethodInvocation` can be constructed using the data provided in the microagent descriptor.

Method invocations return an instance of the `MicroAgentData` class, which acts as a container for transmitting invocation results. `MicroAgentData` objects contain source and data attributes. The source attribute is a `MicroAgentID` object identifying the source of the data. The data attribute contains the actual method invocation results.

Error Handling

Two general types of errors can occur with method invocations:

1. During the delivery of a method invocation to, or the results back from, a microagent. This condition will cause the `invoke()` method to throw a `MicroAgentException`.
2. During the invocation of the method on the microagent itself. This condition will cause this method to deliver a `MicroAgentException` in the data field of the `MicroAgentData` return.

Thus, a successful method invocation is one that doesn't throw an exception and does not deliver a `MicroAgentException` in the data field of its `MicroAgentData` return.

If the method invocation is successful, the data field of its `MicroAgentData` return will contain the method's return value or null if the method doesn't return a value.

Subscribing to Method Results

The `AgentManager` also allows you to register a subscription for microagent methods with its `subscribe()` method. Registering for a subscription is analogous to registering for an event. It results in a continuous stream of `MicroAgentData` return values from the method being subscribed to. This data is asynchronously delivered to a `SubscriptionHandler` object supplied during registration.

In addition to a `SubscriptionHandler`, the `subscribe()` method of `AgentManager` requires a `MicroAgentID`, to identify the target, and a `MethodSubscription` object.

The MethodSubscription Class

The `MethodSubscription` class extends the `MethodInvocation` class. A `MethodSubscription` should be constructed in accordance with the `MethodDescriptor` of the target method.

Registering a subscription returns a `Subscription` object that can be used to cancel the subscription and to examine how the subscription was created

Only methods that return data should be subscribed to. This includes methods of type `IMPACT_INFO` and `IMPACT_ACTION_INFO`.

Synchronous and Asynchronous Method Subscriptions

You can use either of two constructors to build a `MethodSubscription`, depending on whether you are creating a subscription for a synchronous or asynchronous method. The `MethodDescriptor` for a method indicates whether the method is synchronous or asynchronous. An asynchronous method is analogous to an event source. Listeners that subscribe to an asynchronous method will receive the method's declared return value each time a particular event occurs.

The `MethodSubscription` constructor for synchronous method subscriptions requires an interval parameter. It is used to specify a desired subscription service rate. Subscriptions made of synchronous methods will use this value as a hint to determine how often to deliver data to service a subscription. Most microagents will usually enforce a minimum rate, which will be used if the supplied value is below their minimum.

The `MethodSubscription` constructor for asynchronous method subscriptions does not accept an interval parameter. Asynchronous methods deliver data at a rate determined by the microagent, typically, whenever it is available.

Error Handling

Two general types of errors can occur with subscriptions:

1. During the registration of the subscription. This condition will cause the `AgentManager.subscribe()` method to throw a `MicroAgentException`.
2. During the servicing of the subscription These errors are delivered to the `SubscriptionHandler`.

Group Operations

Group operations are performed with the `groupOp()` method of the `AgentManager`. They are very similar to method invocations. The difference is that they require an array of `MicroAgentID` targets and return an array of `MicroAgentData` values.

A group operation effectively performs a method invocation simultaneously on all of the specified target microagents. It is useful for affecting a group of microagents in a single operation.

Lightweight Console

By default, the console monitors and can communicate with every agent in a given TIBCO Hawk domain. If, however, you are building an application that needs to interact with only a single agent, you can initialize the console in such a way that limits its communication to just that agent. This results in a lightweight console instance that uses fewer resources.

Creating a Lightweight Console

Instantiating a lightweight console involves constructing a TIBHawkConsole object with a fully qualified hawkDomain parameter. A fully qualified hawkDomain narrows down the domain to a single agent and contains the three parts that uniquely identify an agent, in a dot-separated format:

```
<hawkDomain>.<agentDNS>.<agentName>
```

- *<hawkDomain>* is what would be used as the hawkDomain parameter of the TIBHawkConsole constructor if you were instantiating a console that communicates with all agents. If an agent is configured without a hawkDomain specification, it uses the value "default".
- *<agentDNS>* should match the agentDNS (also called agent domain) parameter with which the agent was configured. If the agent was not configured with an agentDNS, it uses the value "none".
- *<agentName>* must match the agent name. If not set, this defaults to the hostname the agent is running on. See The AgentInstance Class on page 12 for more information

Using the Dot and Underscore Characters

None of the three elements may contain the dot '.' character. The dot is used as a separator in a fully qualified hawkDomain. If any of the three components require a dot (agentDNS typically does), replace all occurrences with the underscore '_' character. The agent automatically performs this translation on its end.

For example, if an agent is configured with the following values:

- hawkDomain = `testDomain`
- agentDNS = `myfirm.com` (note the dot in the name)
- agentName = `host1`

the fully qualified hawk domain is:

```
testDomain.myfirm_com.host1
```

Console Application with Secure Domain

To build a console application with secure domain, you need to use a secure transport. You can connect to TIBCO EMS transport with SSL using Console API for Agent-Console communication.

Refer to `HAWK_HOME/examples/console_api/TestConsoleSSL.java` for a sample java program to demonstrate a console test application which uses SSL parameters to connect to SSL.

MicroAgent Plug-in

The `COM.TIBCO.hawk.agent.nest` package is used to write TIBCO Hawk microagents that run inside the agent. To run as a microagent in the TIBCO Hawk agent, an object must implement either the `MicroAgent` or `ServiceMicroAgent` interface. Note that this package is dependent on the `COM.TIBCO.hawk.console.talon` package.

TIBCO Hawk Console API Class Structure

This section contains UML diagrams showing the structure of the following two packages:

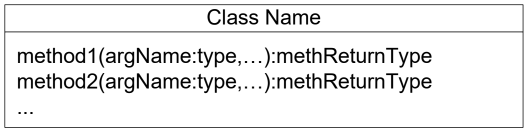
`COM.TIBCO.hawk.hawkeye`: classes used to monitor agents

`COM.TIBCO.hawk.talon`: classes used to manage agents

Key to the UML Diagrams

These class diagrams use a subset of UML notation. Here is a brief key:

Classes and their methods are shown in rectangles:

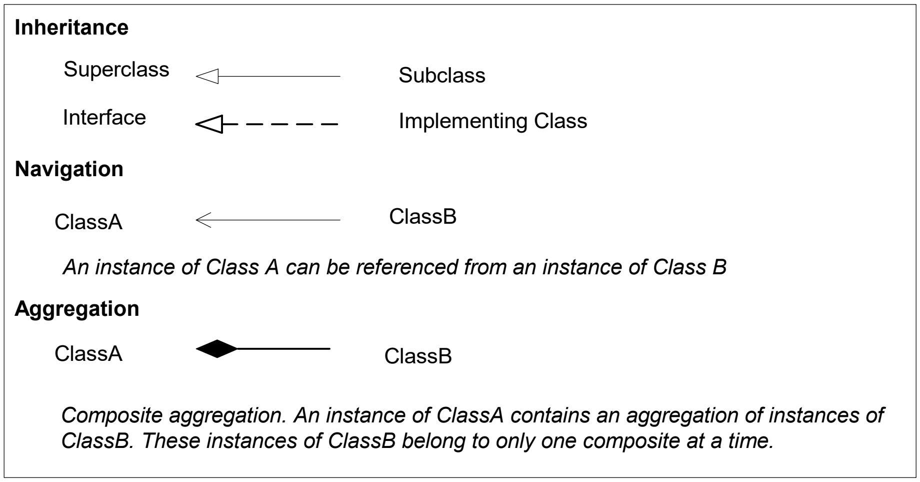

Class Name
method1(argName:type,…):methReturnType
method2(argName:type,…):methReturnType
...

Some methods and argument names are omitted for clarity. A third part, between the class name and methods, may be used for class variables. Underlined members are static.

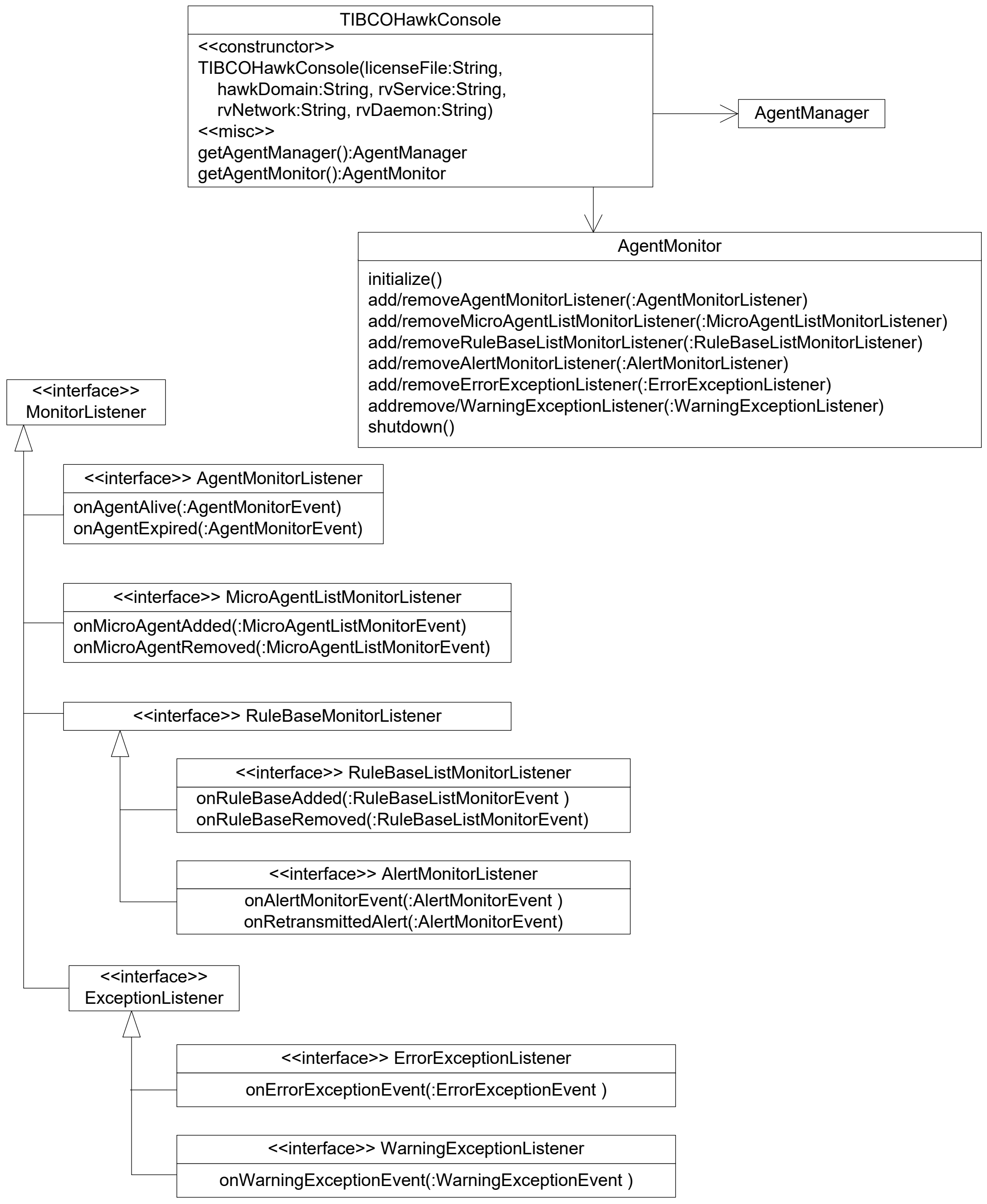
Comments are shown in a rectangle with a turned down corner. A dotted line shows what the comment applies to.

Associations

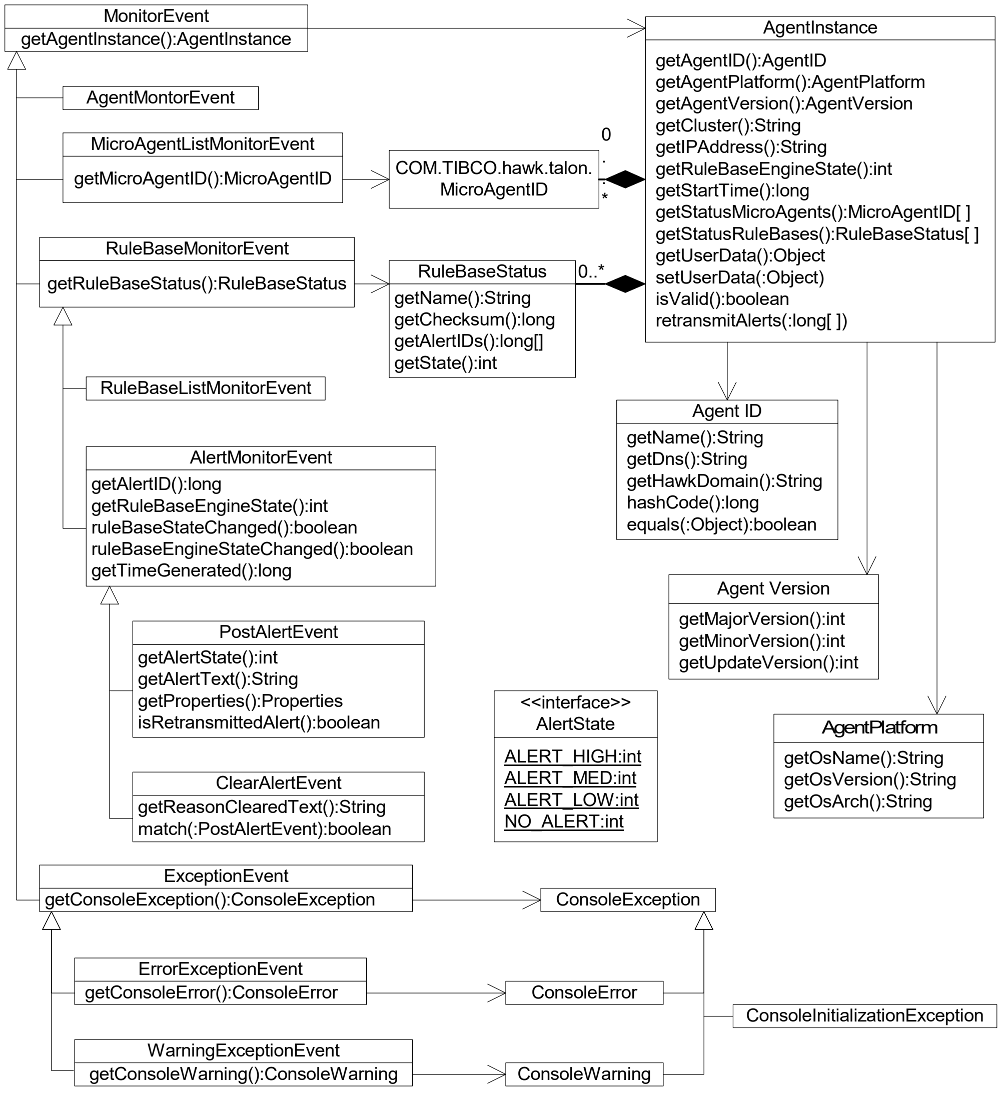
Associations are indicated with lines:


Inheritance
Superclass
Subclass
Interface
Implementing Class
Navigation
ClassA
ClassB
An instance of Class A can be referenced from an instance of Class B
Aggregation
ClassA
ClassB
Composite aggregation. An instance of ClassA contains an aggregation of instances of ClassB. These instances of ClassB belong to only one composite at a time.

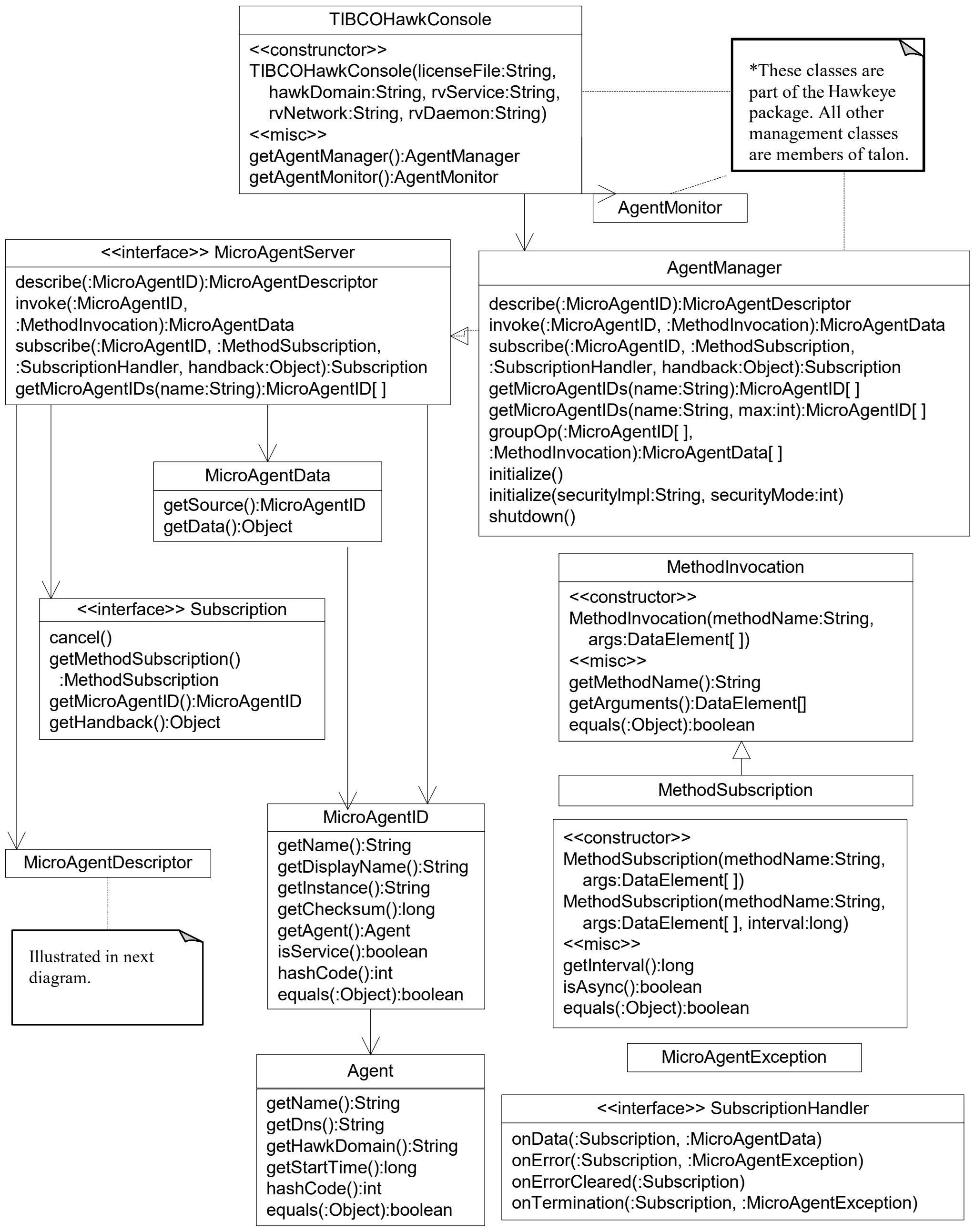
Agent Monitoring Classes (Hawkeye)


TIBCOHawkConsole
<<construnctor>>
TIBCOHawkConsole(licenseFile:String, hawkDomain:String, rvService:String, rvNetwork:String, rvDaemon:String)
<<misc>>
getAgentManager():AgentManager
getAgentMonitor():AgentMonitor
AgentManager
AgentMonitor
initialize()
add/removeAgentMonitorListener(:AgentMonitorListener)
add/removeMicroAgentListMonitorListener(:MicroAgentListMonitorListener)
add/removeRuleBaseListMonitorListener(:RuleBaseListMonitorListener)
add/removeAlertMonitorListener(:AlertMonitorListener)
add/removeErrorExceptionListener(:ErrorExceptionListener)
addremove/WarningExceptionListener(:WarningExceptionListener)
shutdown()
<<interface>> MonitorListener
<<interface>> AgentMonitorListener
onAgentAlive(:AgentMonitorEvent)
onAgentExpired(:AgentMonitorEvent)
<<interface>> MicroAgentListMonitorListener
onMicroAgentAdded(:MicroAgentListMonitorEvent)
onMicroAgentRemoved(:MicroAgentListMonitorEvent)
<<interface>> RuleBaseMonitorListener
<<interface>> RuleBaseListMonitorListener
onRuleBaseAdded(:RuleBaseListMonitorEvent)
onRuleBaseRemoved(:RuleBaseListMonitorEvent)
<<interface>> AlertMonitorListener
onAlertMonitorEvent(:AlertMonitorEvent)
onRetransmittedAlert(:AlertMonitorEvent)
<<interface>> ExceptionListener
<<interface>> ErrorExceptionListener
onErrorExceptionEvent(:ErrorExceptionEvent)
<<interface>> WarningExceptionListener
onWarningExceptionEvent(:WarningExceptionEvent)

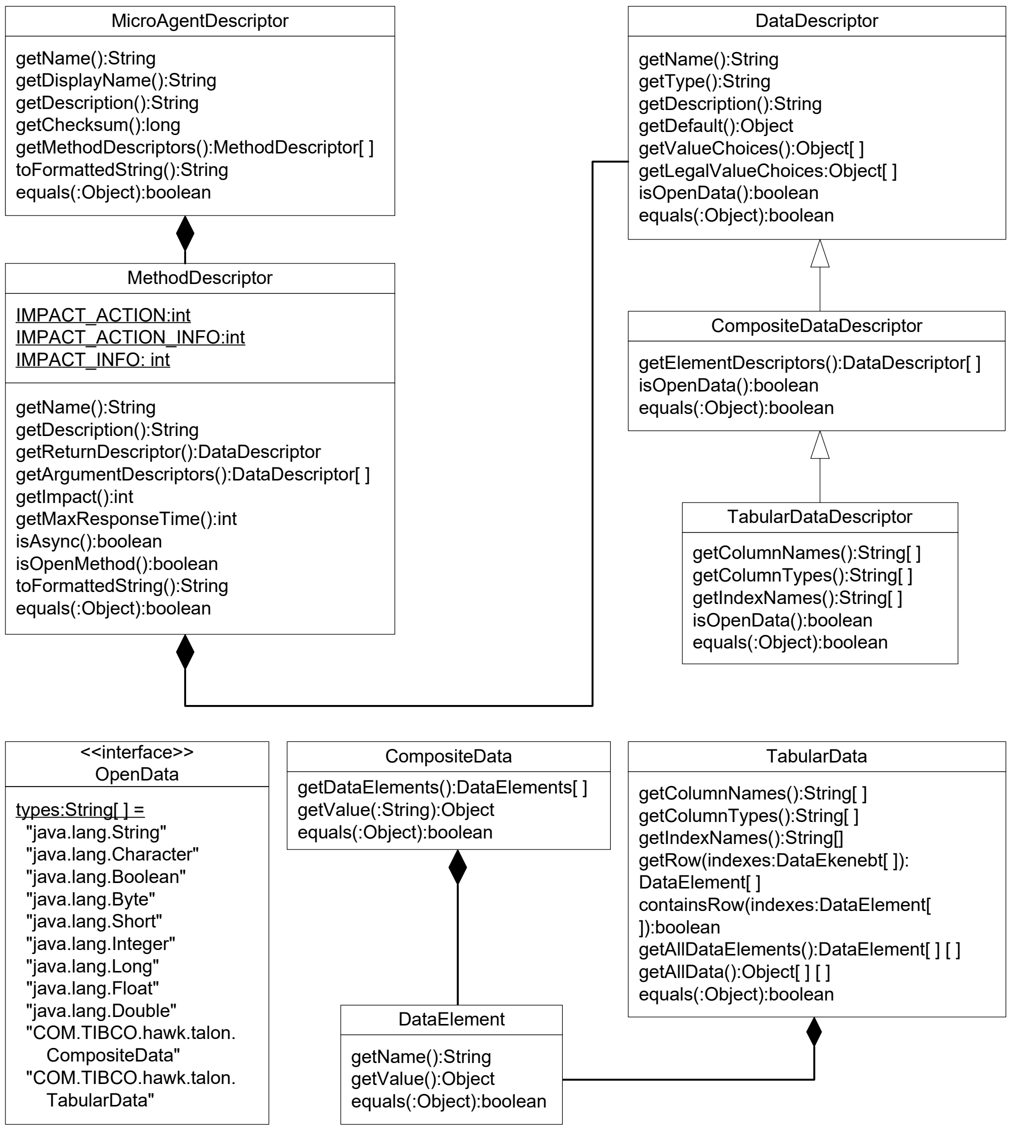
Agent Monitoring Classes (Cont)



Agent Management Classes


TIBCOHawkConsole
<<construnctor>>
TIBCOHawkConsole(licenseFile:String, hawkDomain:String, rvService:String, rvNetwork:String, rvDaemon:String)
<<misc>>
getAgentManager():AgentManager
getAgentMonitor():AgentMonitor
*These classes are part of the Hawkeye package. All other management classes are members of talon.
AgentMonitor
<<interface>> MicroAgentServer
describe(:MicroAgentID):MicroAgentDescriptor
invoke(:MicroAgentID, :MethodInvocation):MicroAgentData
subscribe(:MicroAgentID, :MethodSubscription, :SubscriptionHandler, handback:Object):Subscription
getMicroAgentIDs(name:String):MicroAgentID[ ]
AgentManager
describe(:MicroAgentID):MicroAgentDescriptor
invoke(:MicroAgentID, :MethodInvocation):MicroAgentData
subscribe(:MicroAgentID, :MethodSubscription, :SubscriptionHandler, handback:Object):Subscription
getMicroAgentIDs(name:String):MicroAgentID[ ]
getMicroAgentIDs(name:String, max:int):MicroAgentID[ ]
groupOp(:MicroAgentID[ ], :MethodInvocation):MicroAgentData[ ]
initialize()
initialize(securityImpl:String, securityMode:int)
shutdown()
MicroAgentData
getSource():MicroAgentID
getData():Object
MethodInvocation
<<constructor>>
MethodInvocation(methodName:String, args:DataElement[ ])
<<misc>>
getMethodName():String
getArguments():DataElement[]
equals(:Object):boolean
<<interface>> Subscription
cancel()
getMethodSubscription() :MethodSubscription
getMicroAgentID():MicroAgentID
getHandback():Object
MethodSubscription
<<constructor>>
MethodSubscription(methodName:String, args:DataElement[ ])
MethodSubscription(methodName:String, args:DataElement[ ], interval:long)
<<misc>>
getInterval():long
isAsync():boolean
equals(:Object):boolean
MicroAgentID
getName():String
getDisplayName():String
getInstance():String
getChecksum():long
getAgent():Agent
isService():boolean
hashCode():int
equals(:Object):boolean
MicroAgentDescriptor
Illustrated in next diagram.
Agent
getName():String
getDns():String
getHawkDomain():String
getStartTime():long
hashCode():int
equals(:Object):boolean
MicroAgentException
<<interface>> SubscriptionHandler
onData(:Subscription, :MicroAgentData)
onError(:Subscription, :MicroAgentException)
onErrorCleared(:Subscription)
onTermination(:Subscription, :MicroAgentException)

Agent Management Classes (Cont)



API Reference

The following link provides access to the TIBCO Hawk Console API Javadocs.

- Console API Javadocs

Chapter 3 Configuration Object API

The TIBCO Hawk Configuration Object API is a Java language interface which is used to build and update TIBCO Hawk Agents configuration objects.

When the TIBCO Hawk Configuration Object API is used with the TIBCO Hawk Console API, you can use the combined APIs to manage configuration objects through remote method invocations on TIBCO Hawk agents or repositories.

This chapter provides a brief overview and usage of the major components of the TIBCO Configuration Object API. It includes a summary of the Java packages in the Configuration Object API and a discussion of related TIBCO Hawk APIs.

Topics

Overview

Configuration Objects

There are three types of configuration objects in TIBCO Hawk:

- Rulebase
- Schedule
- Rulebase Map

A TIBCO Hawk Agent manages and monitors managed objects by processing rulebases. A rulebase is a named collections of rules that contain management logic. The management logic in a rule is defined by the tests and actions to be taken from data collected from a given data source. A rulebase can be loaded on a single agent, on a group of agents, or on every agent in the network depending on the TIBCO Hawk Agents' configuration mode.

A schedule defines when a rulebase, rule, test or action is active. If schedule is not specified in a rulebase, the rulebase is always active when loaded. A schedule contains inclusion and/or exclusion periods that will determine if a schedule is in-schedule or out-of-schedule at a specified time.

A rulebase map maps rulebases to TIBCO Hawk Agents on the network. It directs TIBCO Hawk Agents or groups of agents on the network to load particular rulebases at startup. It is used by TIBCO Hawk Agents running in either Manual Configuration Mode or Repository Configuration Mode.

Together with the Console API, these configuration objects can be retrieved and sent to TIBCO Hawk Agents or Repositories from a Java application.

How the TIBCO Hawk Configuration Object API Fits In

The TIBCO Configuration Object API provides classes to define configuration objects such as rulebases, schedules, and rulebase maps. These classes enable you to write programs that can create and modify rulebases, schedules, and rulebase maps programmatically without using the editors in the TIBCO Hawk WebConsole. Rulebases, schedules, and rulebase maps are used by TIBCO Hawk agents to monitor and manage systems and applications.

When using the Rulebase Editor, the context of the rulebase is implied. Using this context, the Rulebase Editor presents you with the data sources and actions that are available to the agent. These data sources and actions are in the form of microagents, methods and arguments. When using the Configuration Object API, rulebase objects specify their data sources and actions using the MethodSubscription and MethodInvocation classes of the TIBCO Hawk Console API.

When the TIBCO Hawk Configuration Object API is used with the TIBCO Hawk Console API, the application can dynamically create or update configuration objects on a TIBCO Hawk agent or repository.

Any complex tests with valid operators can be built using the rulebase related classes in Configuration Object API. However, not all the test conditions built using the Configuration Object API can be modified using the rulebase editor in TIBCO Hawk WebConsole. In such cases, the rulebase editor will simply display the test as a string. However, such complex tests can be edited using a custom-built editor based on the TIBCO Configuration Object API.

Concepts

This section provides some background information on the configuration objects and the Configuration APIs.

Configuration Objects

A TIBCO Hawk Agent manages and monitors applications and systems based on configuration objects such as rulebases, schedules, and a rulebase map loaded on the Agent.

A rulebase map directs TIBCO Hawk agents or groups of agents on your network to load particular rulebases at startup. For example, using a rulebase map you can instruct an agent to load a rulebase designed specifically for the operating system where it runs.

Every rulebase contains rules which are made up of data sources, tests, and actions. Each rule contains management logic. The management logic in a rule is defined by the tests and actions to be taken from data collected from a given data source. If a schedule is specified in a rulebase, rule, test or action, it will determine if these objects should be active or not at a specified time.

Details of rulebases, schedules, and rulebase maps are described below.

Rulebases

A rulebase is a configuration object that provides the rules for the monitoring activities that are to be autonomously performed on an agent. At the core of all rulebase monitoring activity is the collection of data, testing of that data, and taking actions based on the test results. All monitored data is provided by the agent's microagents through microagent subscriptions. All actions taken by a rulebase are in the form of method invocations. Rulebase objects specify their data sources and actions using the MethodSubscription and MethodInvocation classes of the Console API. Therefore, understanding these, and related classes, is a prerequisite for using the Configuration API. For more information on these classes, refer to Chapter 2, Console API.

How Rulebases are interpreted by the RuleBaseEngine

While rulebases are merely configuration objects, it is useful to think of them as having runtime behavior in order to understand how the RuleBaseEngine processes them. Thus this section discusses rulebases, rules, tests, and actions as if they contain logic which carries out their execution.

Structure of a Rulebase

A rulebase object is primarily composed of a set of rule objects. Each rule has a Data Source and a list of Test objects. Each Test has a TestExpressionOperator object and a list of ConsequenceAction objects. Thus, a rulebase can be represented as a tree structure with a single Rulebase object as the root and Action objects as the leaves.

RuleBaseElement Class

The RulebaseElement class is the super class of the following classes:

- Rulebase
- Rule
- Test
- Action

The `RulebaseElement` class provides common methods to get and set the element's name and schedule parameters. The Rule, Test and Action classes do not require a name to be specified in the constructor. Only the Rulebase class requires a name specified in the constructor. In places where an array of RulebaseElement objects is required, all elements in the array must have unique names.

These includes the constructors for:

- Rulebase (requires array of Rule)
- Rule (requires array of Test)
- Test (requires array of Action)

and the following methods:

- Rulebase.setRules()
- Rule.setTests()
- Test.setConsequenceActions()
- Test.setClearActions()

Rule Class

A Rule consists of a data source and a list of tests. The DataSource of a rule specifies a MethodSubscription, which supplies a stream of data samples to be monitored. The method used in the MethodSubscription can be either synchronous or asynchronous. Every new data sample from a Rule Object 's Data Source is distributed to all Test objects contained within that Rule object.

For more information on MethodSubscription refer to the Chapter 2, Console API.

DataSource Class

The data source for a Rule is its source of input data, and is always a method subscription to a microagent. The data source of a Rule provides information about some condition on a managed node. After information is received, one or more tests are applied to evaluate it. The `MethodSubscription` of a data source provides a stream of data objects.

The `microAgentName` and the method name used to construct the DataSource can be obtained from `MicroAgentDescriptor` and `MethodDescriptor`, respectively.

The MicroAgentDescriptor which is used to construct the MethodSubscription defines the type of data the subscription will yield.

This data will be one of the following OpenData types:

- String
- Char
- Boolean
- Byte
- Short
- Integer
- Long
- Float
- Double
- CompositeData
- TabularData (represents a table composed of rows uniquely indexed by uniformly structured CompositeData objects)

Test Class

Tests define the tests which are performed on the rule's data source and what actions to take. Each test uses the data to compute a true or false value which is used in determining when to trigger actions. Test objects have a state that is either true or false. The initial state of a new Test object is false. State transitions are caused by evaluating the received data based on the specified conditions and the policies of the test. The possible Test object state transitions are:

- false to false (F->F)
- false to true (F->T)
- true to false (T->F)

- true to true (T->T)

All `Test` object state transitions cause its `ConsequenceAction` objects to be evaluated. The policy of the `ConsequenceAction` objects govern whether an evaluation results in an action execution. The `ClearAction` objects are a list of actions that will be executed when the `Test` object undergoes the T->F transition.

State transitions resulting from the receipt of data start with an evaluation of the TestExpressionOperator against the data. The resulting true or false value of the TestExpression, in conjunction with the Test object's TrueConditionPolicy and ClearConditionPolicy, determines the type of Test object state transition, as follows:

- F->F and T->T Test Object StateTransitions

  If the previous state of the Test object was false and the current evaluation of the Test Expression is false, then the Test object undergoes a F->F transition.

  If the previous state of the Test object was true and the current evaluation of the TestExpression is true, then the Test object undergoes a T->T transition.

  These two transitions are not affected by the true or clear policy.

- F->T Test Object State Transitions

  This transition is governed by the TrueConditionPolicy.

  The policy TrueCountThreshold specifies how many sequential true evaluations of the TestExpression must occur before the Test object transitions F->T.

- T->F Test Object Transitions - "Clearing" the Test object

  This transition is governed by the ClearConditionPolicy. This transition is synonymous with "Clearing" the Test object.

  The policy FirstFalse indicates that the T->F transition should occur upon the first false evaluation of the TestExpression operator. This is the default ClearConditionPolicy.

  The policy ClearTimer indicates the number of seconds which must elapse without a true evaluation of the TestExpression before the T->F occurs. A T->F transition caused as a result of a clear timer expiration occurs independently of the Test object's receipt of data.

  The ClearTest policy specifies an additional test expression (clear test expression), which governs when the T->F transition occurs. The Clear Test Expression receives data each time the Test object receives data. It will cause a T->F transition of the Test object if (the current state of the Test object is true and) the clear test expression evaluates to true.

TestExpressionOperators Class

TestExpressionOperator are created using the Operator class. The static method Operator.getOperatorDescriptors() returns a list of descriptors describing all available operators. Using the information in the OperatorDescriptor, you can then build instances of the Operator class by supplying the operator name and a list of operands. The operands you supply must be of the same number and type as those specified by the corresponding descriptor. An operand of an operator may itself be another operator, as long as its stated return type matches the operand position it occupies. Operators can thus be nested to form more complex operators.

Although operators can return different types, only those which return a Boolean value may be used in tests (i.e. as arguments to `Test.setTestExpressionOperator()`). The other non-Boolean operators are used only as nested operators.

Test operators access the rule's data source through the COM.TIBCO.hawk.config.rbengine.rulebase.operators.getRuleData operator. This operator takes a name and returns the associated data. As described, if a data source produces TabularData then that data is decomposed into CompositeData objects before seen by the tests and thus the getRuleData operator. The name parameter to this operator references the corresponding data element of the CompositeData object which is then returned by the getRuleData operator. If the data source produces one of the remaining OpenData types (String, Char, Boolean, Byte, Short, Integer, Long, Float, Double) then that value is accessible via the getRuleData operator using the name assigned to the return type in the MethodDescriptor for this data source.

ConsequenceAction Class

The ConsequenceAction object extends the Action object. The Test object invokes its ConsequenceAction objects each time the Test object makes a state transition. The type of transition along with the ConsequenceAction object's PerformActionPolicy and EscalationPeriod determines whether or not the action is executed.

A True Series of transitions is defined as a series of transitions that begins with F->T and is followed by one or more T->T transitions. A T->F marks the end of a true series but is not part of it.

Actions are not enabled during an entire true series. The EscalationPeriod specifies the number of seconds that must elapse since the start of a true series before the action becomes enabled. An EscalationPeriod of 0 indicates that the action is always enabled. Actions may only execute when enabled.

The PerformActionPolicy controls how many times and how often the action executes during a true series, after the action has been enabled.

The PerformOnceOnly policy causes the action to be executed only once during a true series. An exception to this rule involves variable substitution. If variable substitution would result in a different action than the last one that has executed within the current true series (For example, raise an alert with different text), then the action will also be re-executed on the current T->T transition.

The PerformAlways policy causes the action to be executed upon every evaluation within a true series (after the action has become enabled).

The PerformCountOnInterval policy is more involved. It causes the action to be executed at the start of a true series (or as soon as it becomes enabled), and on subsequent evaluations within the same series that occur at a time greater than Y seconds since the last action execution within the current true series. This continues until the action has executed for a maximum of X times within the current true series.

Alerts and Clears

Alerts are generated when a ConsequenceAction invokes the sendAlertMessage on the RulebaseEngine microagent. The method takes a single argument named 'message'. The value of the argument may be one of the following objects:

- AlertLow
- AlertMedium
- AlertHigh
- Notification

AlertLow, AlertMedium, and AlertHigh correspond to alert with level from low to high. They are useful for sending non-alert type messages. All methods take a single string argument called 'alertMsg'. Alerts are cleared when the Test Object (that generated the alert) transitions T->F.

The following code fragment constructs a valid ConsequenceAction which generates a medium alert with text "process down":

```
DataElement[] args =
                {new DataElement("message", new AlertMedium("process down"))};
        MethodInvocation mi =
                new MethodInvocation("sendAlertMessage", args);
        ConsequenceAction ca =
           new ConsequenceAction("COM.TIBCO.hawk.microagent.RuleBaseEngine", mi);
```

Posted Conditions

Posted Conditions are "posted" when a ConsequenceAction object invokes the method postCondition on the RuleBaseEngine microagent. A posted condition is an internal status message, similar to an alert message. It takes a single argument called 'condition'. The following code fragment constructs a valid ConsequenceAction which posts the condition "disk full":

```
DataElement[] args =
                {new DataElement("condition", new PostedCondition("disk full"))};
        MethodInvocation mi =
                new MethodInvocation("postCondition", args);
        ConsequenceAction ca =
                new ConsequenceAction("COM.TIBCO.hawk.microagent.RuleBaseEngine",
mi);
new ConsequenceAction("COM.TIBCO.hawk.microagent.RuleBaseEngine", mi);
```

A ClearAction may not contain a MethodInvocation with the postCondition method. Posted Conditions are "cleared" or "unposted" when the enclosing Test object transitions T->F. Posted conditions provide a mechanism for different rules within the same rulebase to communicate. One of the restrictions on posted conditions is that no two ConsequenceAction objects in the same rulebase may post the same condition (conditionName). This is enforced by the methods that construct and edit Rulebase objects.

Another restriction is that a posted condition may not be referenced (used in a test operator) from within the same Rule that generates it. (Rules contain tests, tests contain actions, and actions can post conditions. Thus all posted conditions are posted within the context of a particular rule but may only be referenced in tests of other rules in the same rulebase.) This is enforced by the methods that construct and edit Rule objects.

For more information on posted conditions, see the TIBCO Hawk Administrator's Guide.

Variable Substitution

The string arguments of all action MethodInvocation objects may contain variables which are evaluated by the rules engine before invocation. By referencing variables, the rulebase can adapt to changes on multiple machines.

For more information on variable substitution, see the TIBCO Hawk Administrator's Guide.

Legal Characters

A rulebase name may contain any alphanumeric character (letter or digit), or the symbols _ (underscore) or - (dash).

A character is considered to be alphanumeric if and only if it is specified to be a letter or a digit by the Unicode 2.0 standard (category "Lu", "Ll", "Lt", "Lm", "Lo", or "Nd" in the Unicode specification data file). The latest version of the Unicode specification data file can be found at http://www.unicode.org/ucd.

For a more complete specification that encompasses all Unicode characters, see *The Java Language Specification* by Gosling, Joy, and Steele.

Overruling

Overruling is a way to have a rule in one rulebase override or overrule a rule in another rulebase in a way that causes only one to be active. Overruling is a way of setting precedence among similar rules.

For more information on overruling, see the *TIBCO Hawk Administrator's Guide*.

Rulebase configuration management

Rulebase use by the agent are maintained in a Rulebase object. Agent stores and retrieves the each Rulebase to and from a rulebase file. The filename of the rulebase correspond to name of the rulebase and has an extension of .hrb. If the filename of the rulebase does not correspond to the name of the rulebase, TIBCO Hawk Agents will not load the rulebase and an error is logged. When TIBCO Hawk Agents is running in auto config mode, rulebases are loaded from the autoconfig directory. When TIBCO Hawk Agents is running in repository config mode, rulebases are loaded from the specified repository.

Schedule

A Schedule is a configuration object that can be used for determining if a rulebase or part of the rulebase should be 'in-schedule' or 'out-of-schedule' at a given time. If a schedule is not specified in a rulebase, then the rulebase is always in-schedule.

Structure of a Schedule

A schedule object is primarily composed of a list of inclusion periods, and a list of exclusion periods. A schedule is in-schedule if at least one of its inclusion periods is in-schedule and none of its exclusion periods are in-schedule. Otherwise, the schedule is out-of-schedule. The inclusion and exclusion periods contain a list of Period objects or PeriodGroup objects.

Period Class

A Period defines the time intervals, days or months that should be included or excluded in a schedule. It is composed of 4 distinct period components: MinutesInDay, DaysInWeek, DaysInMonth and MonthsInYear. A Period object is in-schedule only if all of its 4 components are in-schedule. Otherwise, it is out-of-schedule.

MinutesInDay contains a set of 1440 continuous 1-minute intervals in a day. The MinutesInDay object is in-schedule if the time for checking the schedule is included in the MinutesInDay.

DaysInWeek contains a set of 7 days in a week. A DaysInWeek is in-schedule if the day of date for checking the schedule is included in the DaysInWeek.

DaysInMonth contains a set of 31 days in a month. A DaysInMonth is in-schedule if the day in the date for checking the schedule is included in the DaysInMonth.

MonthsInYear contains a set of 12 months in a year. A MonthsInYear is in-schedule if the month of the date used for checking the schedule is included in the MonthsInYear.

PeriodGroup Class

A PeriodGroup object is a logical group of Period object useful for defining an abstract group of periods. Period groups are useful when you use a set of periods regularly in defining schedules. It also eases the maintenance of those schedules because you can make a change in the period group and have it automatically reflected in all the schedules that use it.

Use of Schedules in Rulebases

Schedules may be used to control when a monitoring activity or action is performed. Schedules may be applied to a RuleBase, Rule, Test, and Action by specifying the schedule name in the attribute of these objects. If a RuleBase, Rule, Test, or Action makes use of a schedule name that is not defined either because the agent couldn't load the Schedule object or because the Schedule object does not exist then it will be flagged as an error. However, the rulebase processing will continue as if no schedule was specified for that component; the component will behave as if always in-schedule

If the schedule name applied to a rulebase component begins with "!" then it refers to the inverse of a schedule. For example, if the schedule BusinessHours is defined in the schedules configuration, a rulebase component may use either BusinessHours or !BusinessHours to refer to it. When using BusinessHours, that component is in-schedule whenever the BusinessHours schedule is in-schedule. When using !BusinessHours, that component is in-schedule whenever the BusinessHours schedule is not in-schedule. If the schedule BusinessHours is not defined in the scheduler then components using either BusinessHours or !BusinessHours will behave as if no schedule is defined (they both will always be in-schedule).

A rulebase is a hierarchical structure: rulebases contain rules, rules contain tests, and tests contain actions. Therefore, a schedule applied to one node in the hierarchy affects all nodes below it. The following sections describe the behavior of RuleBases, Rules, Tests, and Actions when valid schedules are applied.

RuleBase

When a rulebase is loaded it is not activated unless its applied schedule is currently in-schedule. Thereafter, when its applied schedule transitions to an in-schedule state, the rulebase is activated. When its applied schedule transitions to an out-of-schedule state, the rulebase is deactivated. Before a rulebase becomes active, no rules are processed no monitoring is taking place by that rulebase. When a rulebase is activated, its rules are loaded and monitoring may begin. When a rulebase is deactivated, all of its rules are unloaded which results in the clearing of outstanding alerts (generated from those rules) and the cessation of all monitoring by that rulebase.

Rule

When a rule is loaded it isn't activated unless its applied schedule is currently in-schedule. Thereafter, when its applied schedule transitions to an in-schedule state, the rule is activated. When its applied schedule transitions to an out-of-schedule state, the rule is deactivated. Before a rule becomes active, no tests are processed and no monitoring is performed by this rule. When a rule is activated, its tests are loaded and monitoring may begin. When a rule is deactivated, all of its tests are unloaded which results in the clearing of outstanding alerts (generated from those tests) and the cessation of all monitoring by that rule. When a rule is inactive, its enclosing rulebase behaves as if that rule is not there.

Test

When a test is loaded it isn't activated unless its applied schedule is currently in-schedule. Thereafter, when its applied schedule transitions to an in-schedule state, the test is activated. When its applied schedule transitions to an out-of-schedule state, the test is deactivated. Before a test becomes active, no actions are loaded and no monitoring is performed by this test. When a test is activated, its actions are loaded and monitoring begins. When a test is deactivated, all of its actions are unloaded which results in the clearing of outstanding alerts (generated from those actions) and the cessation of all monitoring by that test. When a test is inactive, its enclosing rule behaves as if that test is not there.

Action

When an action is loaded it isn't activated unless its applied schedule is currently in-schedule. Thereafter, when its applied schedule transitions to an in-schedule state, the action is activated. When its applied schedule transitions to an out-of-schedule state, the action is deactivated. Before an action becomes active, it performs no action and does not respond in any way to its test's state transitions. When an action is activated, it begins tracking and responding to its test's state transitions. When an action is deactivated, any outstanding alert it may have generated is cleared and the action ceases to track and respond to the state transitions of its test. When an action is inactive, its enclosing test behaves as if that action is not there.

Schedule Configuration Management

In most respects, configuration management for Schedules is identical to that for rulebases (when in auto-config mode, the agent will load and store this file from auto-config-dir, etc.). However, all schedules use by the agent are maintained in a single Schedules object. Agent stores and retrieves the Schedules to and from the file schedules.hsf.

Schedules and Agent Performance

Because all schedules are stored in a single file, each agent will load the schedules at startup. However, the scheduler in the agent will evaluate a schedule only if the agent has loaded rulebases that reference that schedule. Such schedules are referred to as *active* because there is active interest in them.

The scheduler evaluates active schedules at the following times:

- Once each minute, when schedules are resolved.
- When the agent receives a new schedule (such as when using SendTo from the schedule editor).
- When a schedule first becomes active (such as the first time any rulebase references it).

In general, having a large number of schedules defined in the schedule file may marginally affect the size of the agent but it does not affect the CPU performance.

RulebaseMap

RulebaseMap is a configuration object that maps rulebases to agents. It is used when agent is running in a manual configuration mode to determine which rulebases should be loaded on the agent.

The RulebaseMap configuration object has three primary components:

- Group mapping - to organize agents into groups for the purpose of rulebase assignment.
- Rulebase mapping - to map rulebases to agents or groups of agents.
- Command mapping - to delegate the rulebase mapping function to an external command.

These components are described in the following sections.

Group Mapping

There are two types of groups in RulebaseMap, user defined and automatic. A user defined group is a group that a user creates. Automatic groups are groups that agents automatically belong to. A user can define the names of user-defined groups but not that of the automatic groups. A user defined group name begin with "+" and automatic group names begin with "++".

A user defined group can be composed of a number of agents, groups, or a combination of agents and groups. A user defined group may have both user defined and automatic groups as elements in its definition.

The OS groups are automatic groups whose names correspond to the operating systems of the machines the agents are running on. The OS groups have the form "++OSName" where OSName is the value of the Java system property "os.name". Examples of automatic group names are ++Windows 2000, ++Solaris, and ++HPUX. Examples of user defined group names are "+servers" and "+clients".

There is a special automatic group referred to as the ALL group. The ALL group includes every TIBCO Hawk Agent and is simply named "++".

For example:

```
+group1 agent1 agent2 agent3
+groupX agentX +group1
```

In the preceding example, `agentX` and `+group1` belong to `+groupX`. Also, `agent1` belongs to `+group1` as well as `+groupX`.

Rulebase Mapping

Rulebase mapping defines which rulebases are assigned to an agent or a group. It defines which agents or groups use a particular rulebase. In the following rulebase mapping:

```
rulebase    agent1 agent2 ++Windows
rulebase2   +group2
rulebase3   agent2 +group2
rulebase4   ++
```

`agent1` uses `rulebase1`. Agents in `+group2` uses `rulebase2` and `rulebase3`. All agents uses `rulebase4` as `rulebase4` maps to "++", the `all` group. All agents that are running under Windows operating system will uses `rulebase1`.

Command Mapping

Command mapping allows an external command or executable (script) to be specified for an agent or a group. If specified, it is executed and the returned string is parsed on white space to indicate which rulebases to load. When the executable is invoked, the agent name and its automatic group name are passed as parameters to the command.

The use of command mapping depends on a setting of one of the attributes of the RulebaseMap. The command mapping can be used as the only mechanism to generate the rulebases to be loaded or as a supplement to the groups and rulebases mapping of the RulebaseMap. It can also be ignored for generating the rulebases.

Agent processing of RulebaseMap

If the agent (more specifically, the RulebaseEngine MicroAgent) is configured in one of the manual configuration mode, it will attempt to load the RulebaseMap configuration object after initialization. It will first determine which automatic groups it belongs to. Then it will read and process the group definition component to determine which user defined groups it is also a member of. Next it will process the rulebase mapping component to determine which rulebases it should load. Finally it will use the command mapping mechanism, if one is specified, to get the names of additional rulebases it should load. Once the RulebaseMap has been fully processed, the agent will proceed to load the target rulebases.

The `-rulebases` option supported by the agent (`RulebaseEngine` MicroAgent) can be used together with the RulebaseMap to specify additional rulebases.

Configuration Object Integrity

When constructing or modifying any of the configuration objects, copies of the supplied parameters are made and used. When accessing the data of any configuration objects through one of the get methods, copies of the internal data are returned. This insures the integrity of the configuration objects and ensures that proper validity checking can be performed. It also means that changing a configuration object requires that you use one of the set methods on that component. For example, if you extract the tests from a rule using the Rule.getTests() method and then modify one of the tests in the array, the change will not be reflected in the rule until you call Rule.setTests() with the modified test array.

Dependence on the Console API

When a configuration object is created using the corresponding editor on the TIBCO Hawk WebConsole, the context of the configuration object is implied by the agent or repository for which the configuration object is defined. For example, when creating rulebases in the rulebase editor, this context is used when presenting to you the choices for data sources and actions, in the form of the related microagents, methods and arguments. When creating rulebases using the Configuration Object API, the microagent name, method name and the data item names must be passed to the methods. These are obtained from the following classes of the COM.TIBCO.hawk.talon package of the Console API:

- MethodSubscription
- MethodInvocation
- MicroAgentDescriptor
- MethodDescriptor
- OpenData

The data source of a rule requires a microagent name and a MethodSubscription object. Actions require a microagent name and a MethodInvocation object.

Microagent names, and the information required to build valid MethodSubscription and MethodInvocation objects, are available in MicroAgentDescriptor objects. The methods used as data sources must be Open Methods since the RuleBaseEngine can only process OpenData.

A MicroAgentDescriptor holds MethodDescriptor objects for all methods of a microagent. The MethodDescriptor for the method chosen as the data source describes the data the method returns. This information is needed to construct test expression operators. The getRuleData operator is used by test expressions to access a method's data. It requires the name of a data item. This name needs to be one of the names of the elements in the method's return which are specified in the MethodDescriptor. Obtaining MicroAgentDescriptor objects is accomplished with the `COM.TIBCO.hawk.hawkeye` package of the Console API.

Retrieving and updating configuration objects on a TIBCO Hawk agent or repository is accomplished by invoking methods on the RuleBaseEngine or Repository microagents.   This involved both the monitoring and management components of the TIBCO Hawk Console API as agents and microagents need to be discovered and method invocations are performed on the required microagents. See Appendix A, Common Configuration Object API Methods, for commonly used methods on RuleBaseEngine or Repository microagent when using the Configuration Object API.

See the *TIBCO Hawk Console API Reference* and the *TIBCO Hawk Methods Reference* for more information.

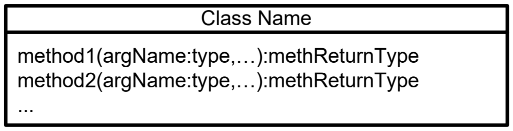
Configuration Object API Class Structure

This section contains UML diagrams showing the structure of the Configuration Object API.

Key to the UML diagrams

These class diagrams use a subset of UML notation. Here is a brief key:

Classes and their methods are shown in rectangles:

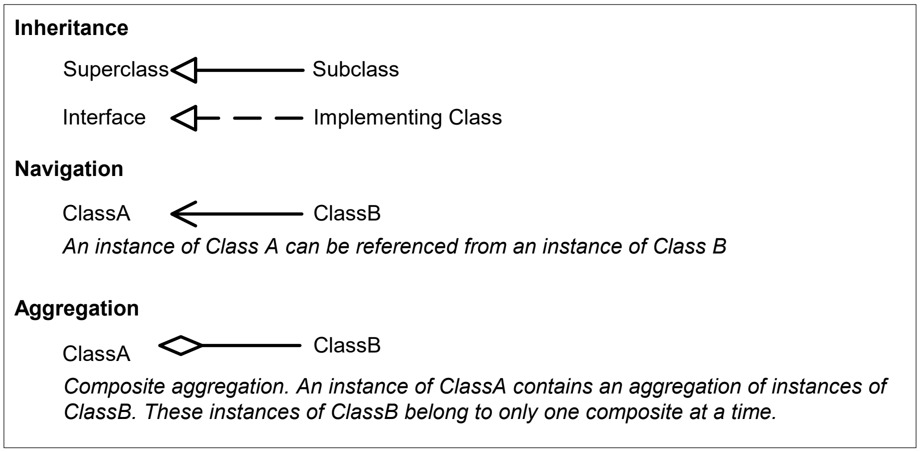

Class Name
method1(argName:type,…):methReturnType
method2(argName:type,…):methReturnType
...

Some methods and argument names are omitted for clarity. A third part, between the class name and methods, may be used for class variables. Underlined members are static.

Comments are shown in a rectangle with a turned down corner. A dotted line shows what the comment applies to.

Associations

Associations are indicated with lines:



An instance of Class A can be referenced from an instance of Class B

Composite aggregation. An instance of ClassA contains an aggregation of instances of ClassB. These instances of ClassB belong to only one composite at a time.

Configuration Object API Classes

Figure 1 Package Level

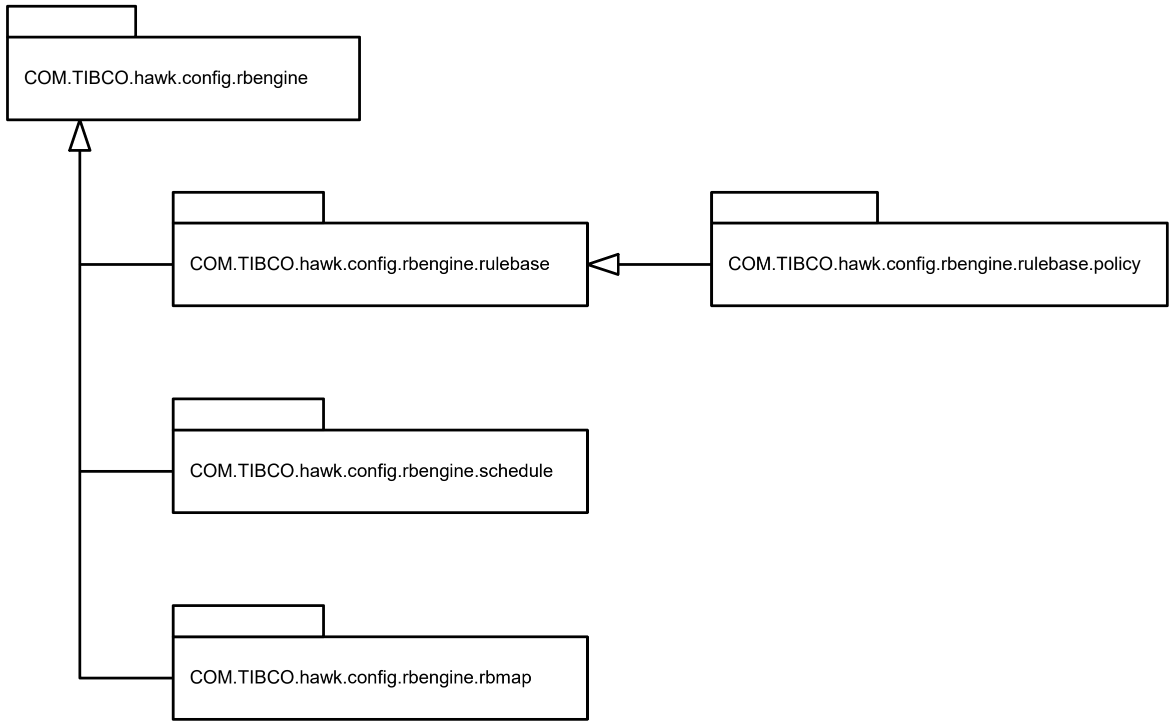
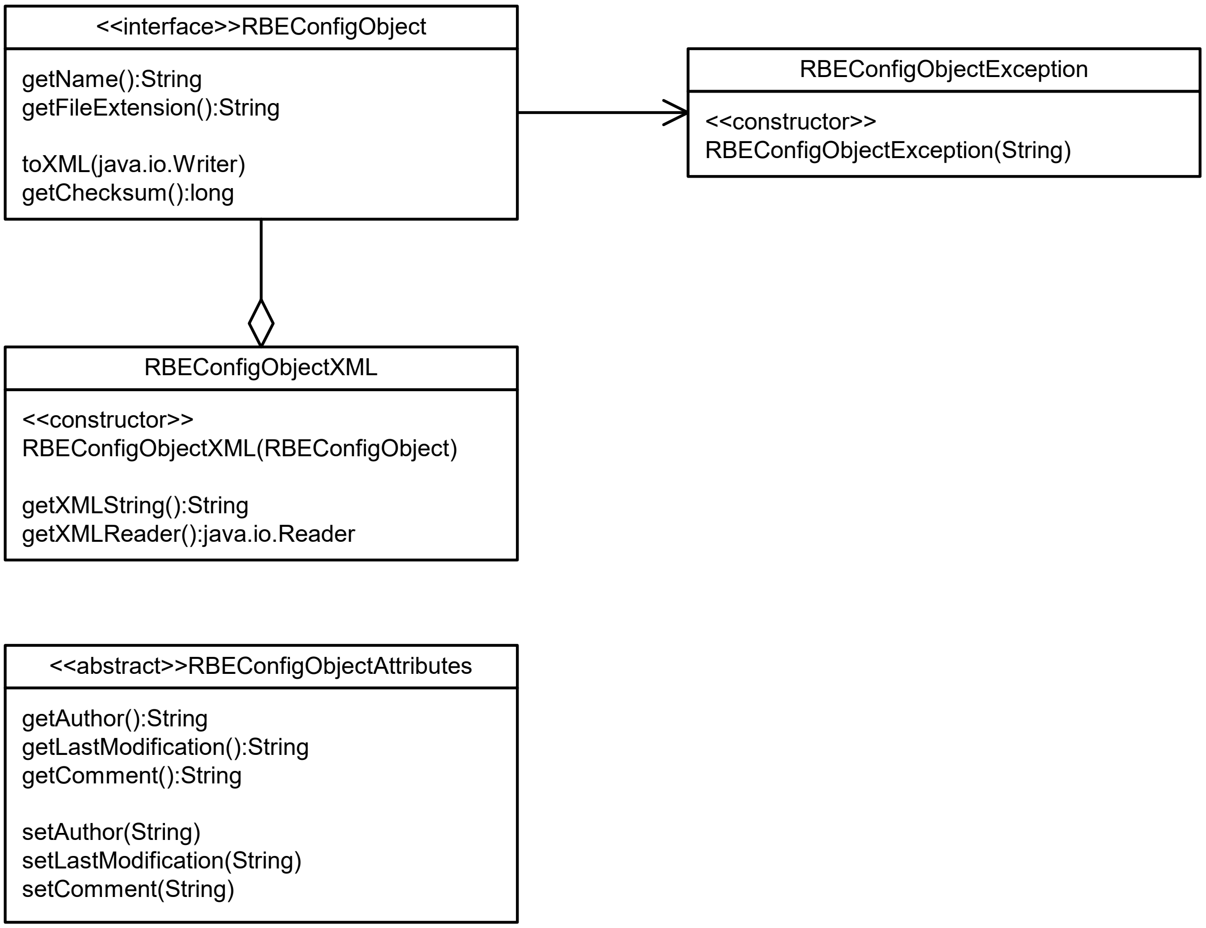

COM.TIBCO.hawk.config.rbengine
COM.TIBCO.hawk.config.rbengine.rulebase
COM.TIBCO.hawk.config.rbengine.rulebase.policy
COM.TIBCO.hawk.config.rbengine.schedule
COM.TIBCO.hawk.config.rbengine.rbmap

Figure 2 Base Classes Extended by Rulebase, Schedule, and Rulebase Map Classes



<<interface>>RBEConfigObject
getName():String
getFileExtension():String
toXML(java.io.Writer)
getChecksum():long
RBEConfigObjectException
<<constructor>>
RBEConfigObjectException(String)
RBEConfigObjectXML
<<constructor>>
RBEConfigObjectXML(RBEConfigObject)
getXMLString():String
getXMLReader():java.io.Reader
<<abstract>>RBEConfigObjectAttributes
getAuthor():String
getLastModification():String
getComment():String
setAuthor(String)
setLastModification(String)
setComment(String)

Figure 3   Rulebase, Rule, and DataSource-related Classes

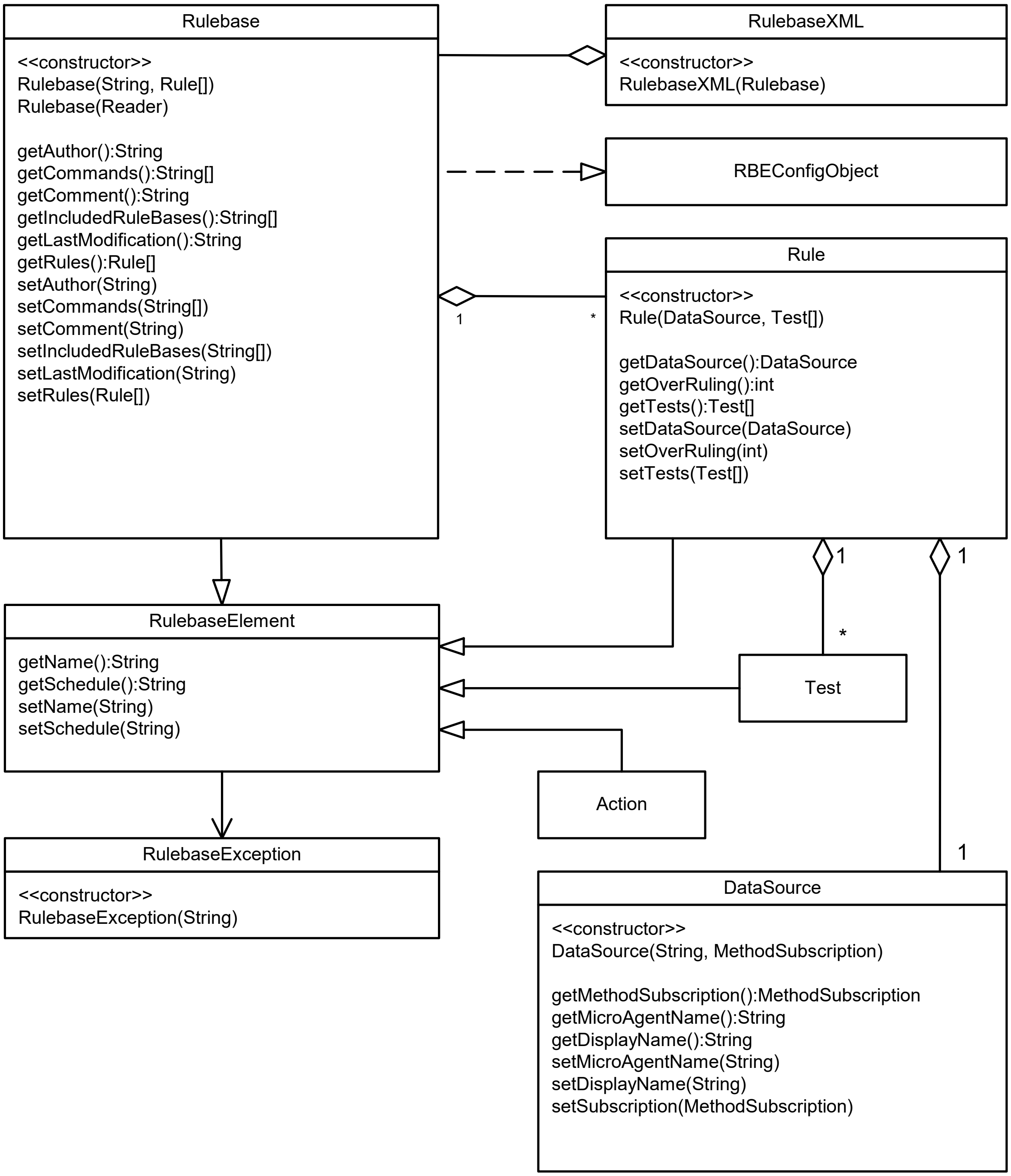


Figure 4 Test-Related Classes

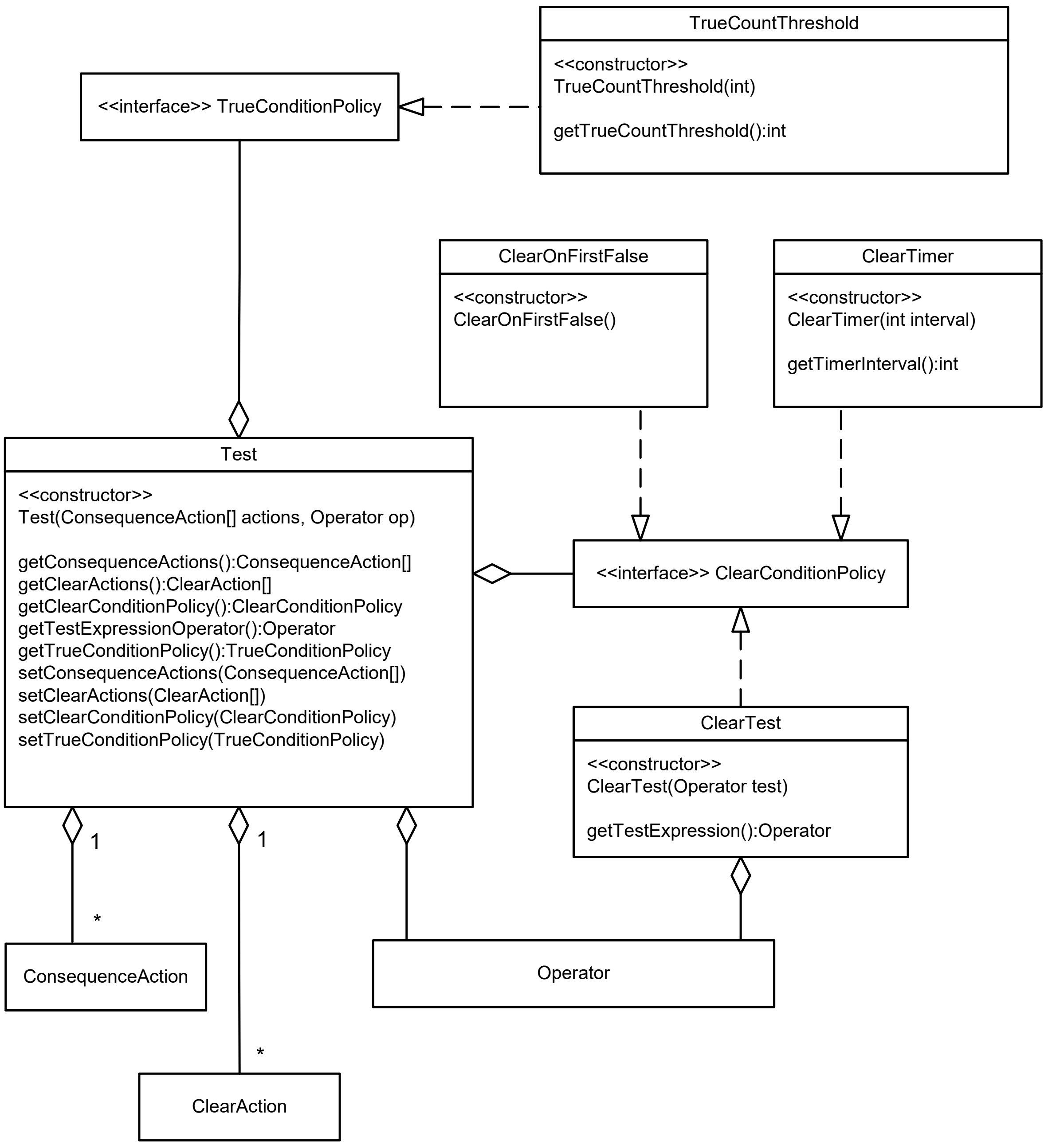

TrueCountThreshold
<<constructor>>
TrueCountThreshold(int)
getTrueCountThreshold():int
<<interface>> TrueConditionPolicy
ClearOnFirstFalse
<<constructor>>
ClearOnFirstFalse()
ClearTimer
<<constructor>>
ClearTimer(int interval)
getTimerInterval():int
Test
<<constructor>>
Test(ConsequenceAction[] actions, Operator op)
getConsequenceActions():ConsequenceAction[]
getClearActions():ClearAction[]
getClearConditionPolicy():ClearConditionPolicy
getTestExpressionOperator():Operator
getTrueConditionPolicy():TrueConditionPolicy
setConsequenceActions(ConsequenceAction[])
setClearActions(ClearAction[])
setClearConditionPolicy(ClearConditionPolicy)
setTrueConditionPolicy(TrueConditionPolicy)
<<interface>> ClearConditionPolicy
ClearTest
<<constructor>>
ClearTest(Operator test)
getTestExpression():Operator
1
1
*
*
ConsequenceAction
Operator
ClearAction

Figure 5 Operator-Related Classes

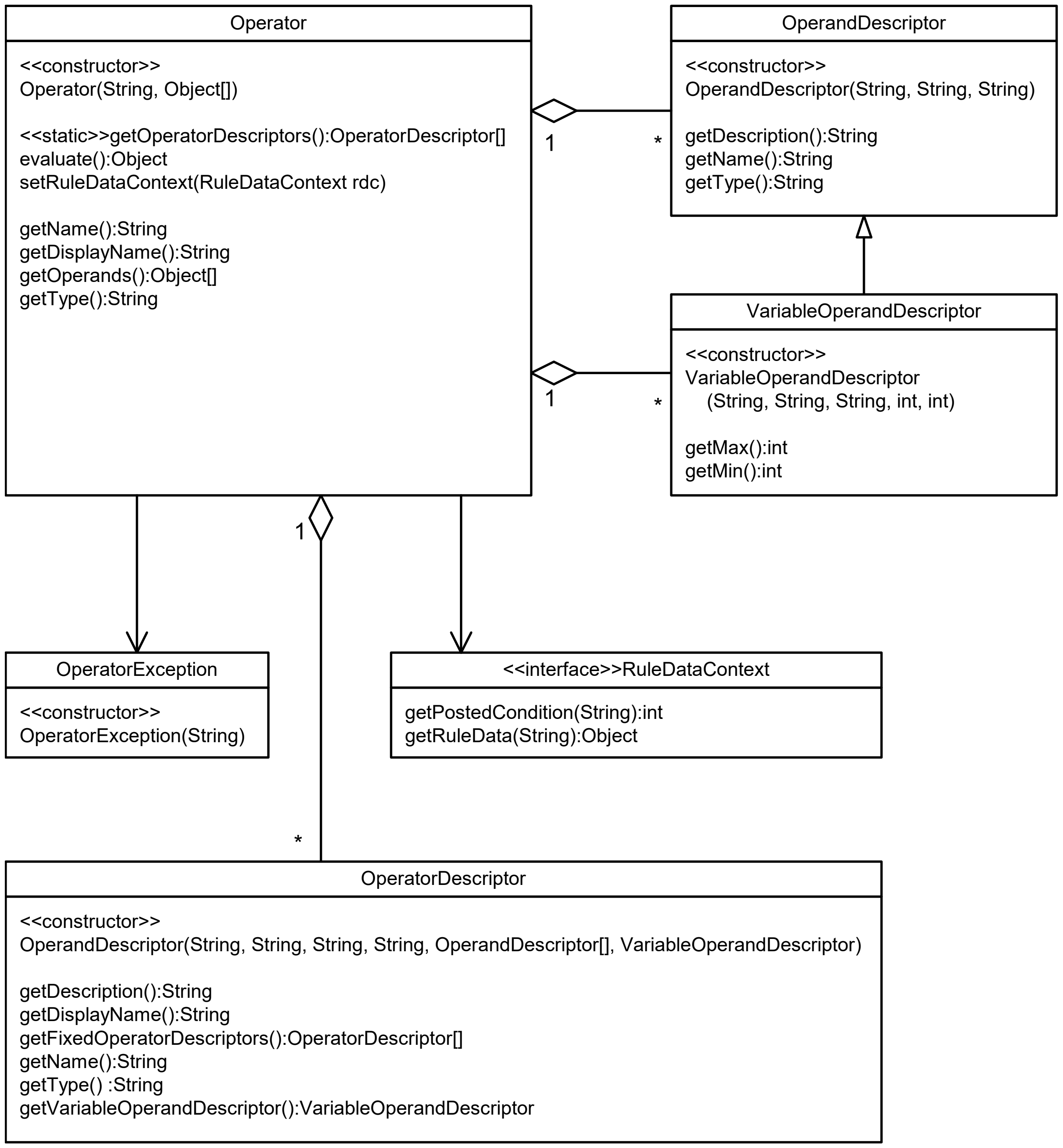

Operator
<<constructor>>
Operator(String, Object[])
<<static>>getOperatorDescriptors():OperatorDescriptor[]
evaluate():Object
setRuleDataContext(RuleDataContext rdc)
getName():String
getDisplayName():String
getOperands():Object[]
getType():String
OperandDescriptor
<<constructor>>
OperandDescriptor(String, String, String)
getDescription():String
getName():String
getType():String
VariableOperandDescriptor
<<constructor>>
VariableOperandDescriptor
(String, String, String, int, int)
getMax():int
getMin():int
1
*
1
*
1
*
OperatorException
<<constructor>>
OperatorException(String)
<<interface>>RuleDataContext
getPostedCondition(String):int
getRuleData(String):Object
OperatorDescriptor
<<constructor>>
OperandDescriptor(String, String, String, String, OperandDescriptor[], VariableOperandDescriptor)
getDescription():String
getDisplayName():String
getFixedOperatorDescriptors():OperatorDescriptor[]
getName():String
getType() :String
getVariableOperandDescriptor():VariableOperandDescriptor

Figure 6 Action-Related Classes

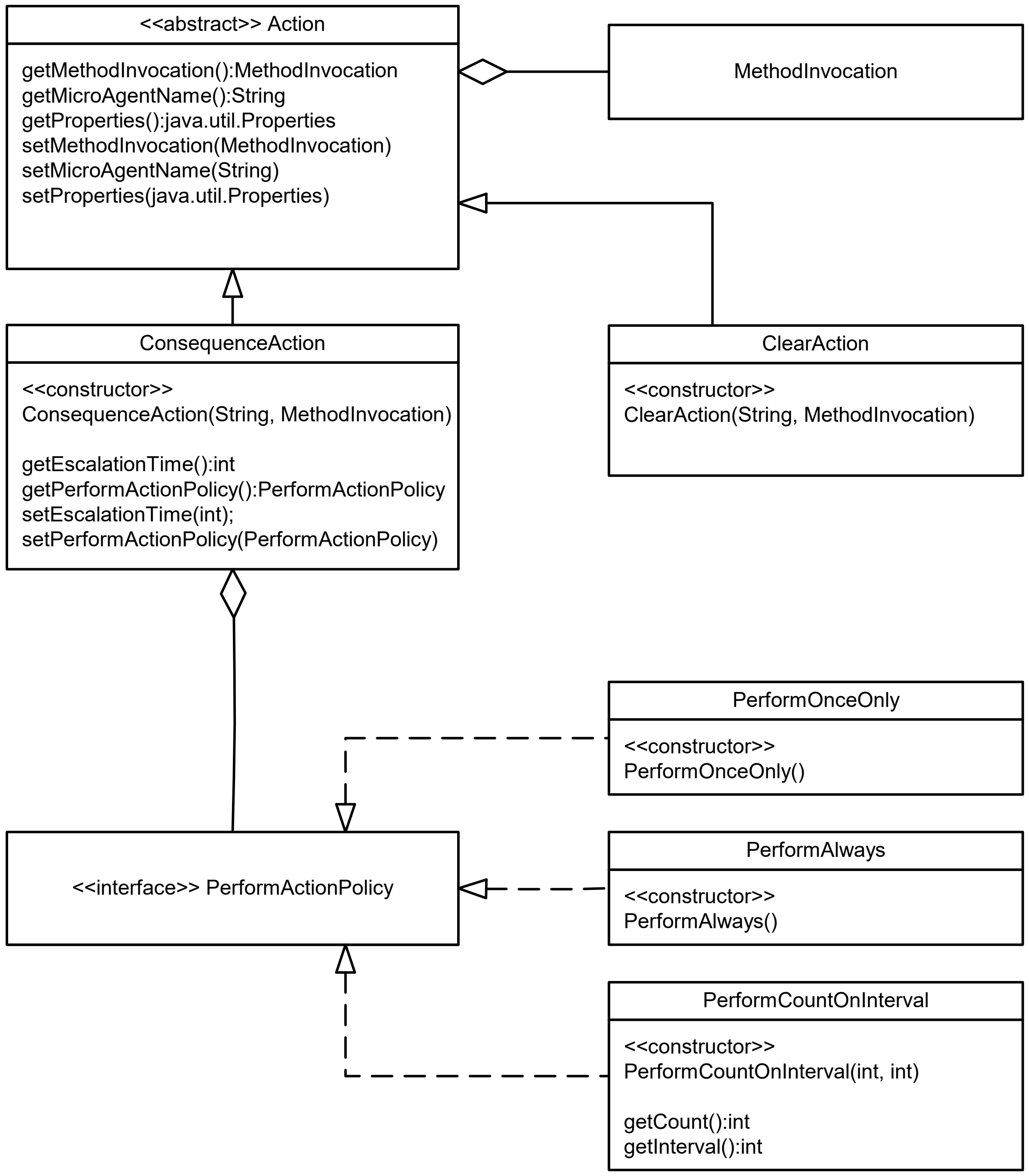

<<abstract>> Action
getMethodInvocation():MethodInvocation
getMicroAgentName():String
getProperties():java.util.Properties
setMethodInvocation(MethodInvocation)
setMicroAgentName(String)
setProperties(java.util.Properties)
MethodInvocation
ConsequenceAction
<<constructor>>
ConsequenceAction(String, MethodInvocation)
getEscalationTime():int
getPerformActionPolicy():PerformActionPolicy
setEscalationTime(int);
setPerformActionPolicy(PerformActionPolicy)
ClearAction
<<constructor>>
ClearAction(String, MethodInvocation)
PerformOnceOnly
<<constructor>>
PerformOnceOnly()
<<interface>> PerformActionPolicy
PerformAlways
<<constructor>>
PerformAlways()
PerformCountOnInterval
<<constructor>>
PerformCountOnInterval(int, int)
getCount():int
getInterval():int

Figure 7 Schedules-Related Classes

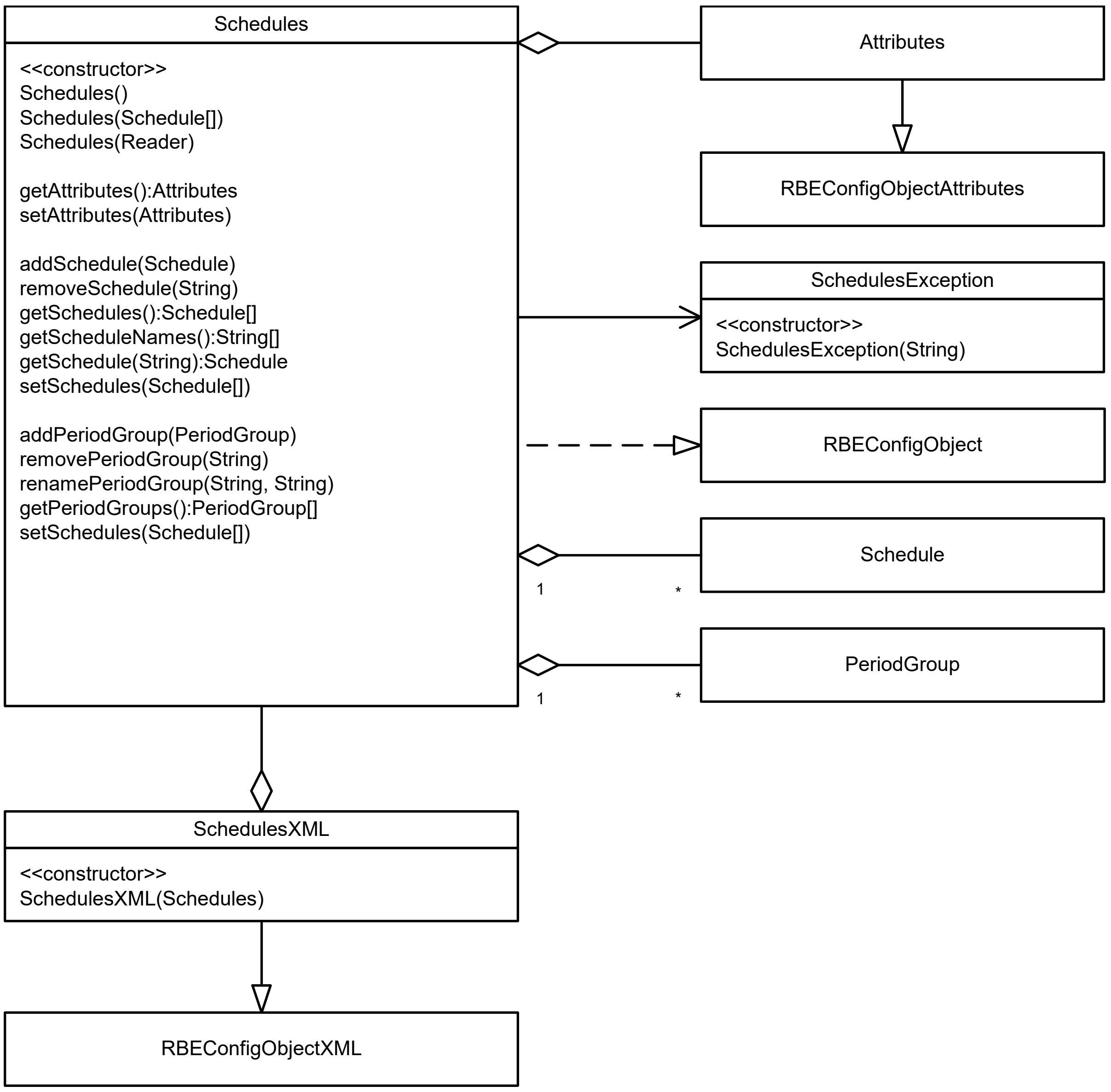


Figure 8 Schedule-Related Classes

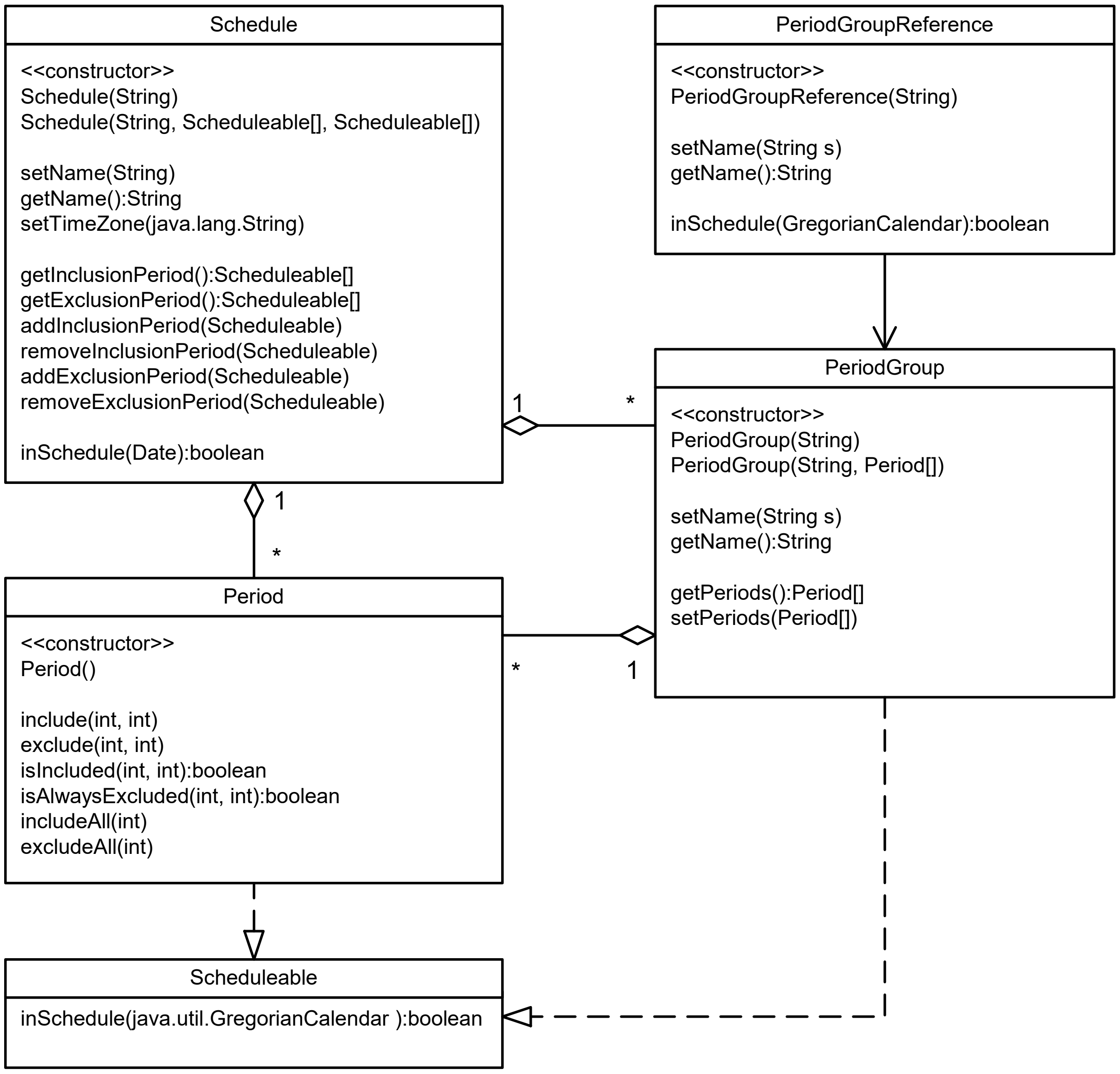
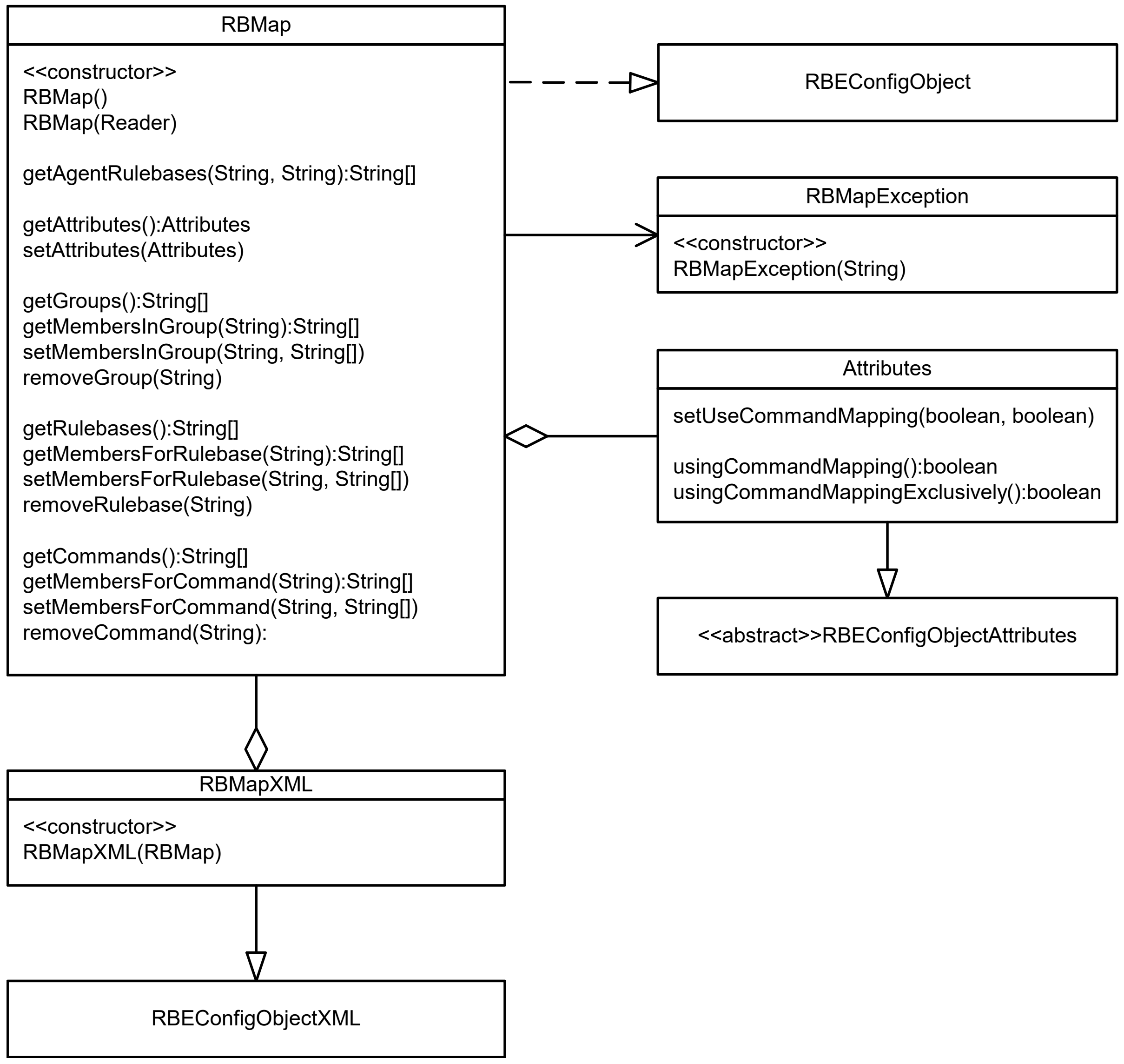

Schedule
<<constructor>>
Schedule(String)
Schedule(String, Scheduleable[], Scheduleable[])
setName(String)
getName():String
setTimeZone(java.lang.String)
getInclusionPeriod():Scheduleable[]
getExclusionPeriod():Scheduleable[]
addInclusionPeriod(Scheduleable)
removeInclusionPeriod(Scheduleable)
addExclusionPeriod(Scheduleable)
removeExclusionPeriod(Scheduleable)
inSchedule(Date):boolean
PeriodGroupReference
<<constructor>>
PeriodGroupReference(String)
setName(String s)
getName():String
inSchedule(GregorianCalendar):boolean
PeriodGroup
<<constructor>>
PeriodGroup(String)
PeriodGroup(String, Period[])
setName(String s)
getName():String
getPeriods():Period[]
setPeriods(Period[])
Period
<<constructor>>
Period()
include(int, int)
exclude(int, int)
isIncluded(int, int):boolean
isAlwaysExcluded(int, int):boolean
includeAll(int)
excludeAll(int)
Scheduleable
inSchedule(java.util.GregorianCalendar):boolean
1
*
1
*
*
1

Figure 9 Rulebase Map-Related Classes


RBMap
<<constructor>>
RBMap()
RBMap(Reader)
getAgentRulebases(String, String):String[]
getAttributes():Attributes
setAttributes(Attributes)
getGroups():String[]
getMembersInGroup(String):String[]
setMembersInGroup(String, String[])
removeGroup(String)
getRulebases():String[]
getMembersForRulebase(String):String[]
setMembersForRulebase(String, String[])
removeRulebase(String)
getCommands():String[]
getMembersForCommand(String):String[]
setMembersForCommand(String, String[])
removeCommand(String):
RBEConfigObject
RBMapException
<<constructor>>
RBMapException(String)
Attributes
setUseCommandMapping(boolean, boolean)
usingCommandMapping():boolean
usingCommandMappingExclusively():boolean
<<abstract>>RBEConfigObjectAttributes
RBMapXML
<<constructor>>
RBMapXML(RBMap)
RBEConfigObjectXML

API Reference

The following link provides access to the TIBCO Hawk Configuration Object API Javadocs.

- Configuration Object API Javadocs

Chapter 4 AMI API

This chapter explains the TIBCO Hawk AMI API basics and it's objects.

Topics

AMI Basics

This section presents a brief overview of the TIBCO Hawk Application Management Interface (AMI).

AMI API is an API used to instrument an application in order to make it manageable by the Hawk System. It is shipped in Java/C/C++ language bindings.

The instrumented application usually connects to the Hawk Agent on the same machine. However, this is not a requirement. You can connect it to a Hawk agent on another machine.

However, it is required that one instrumented application instance only connects to one Hawk Agent.

An Instrumented Application Looks like a Microagent

You interact with the instrumented application using the TIBCO Hawk agent by calling the methods of microagents associated with that agent. An AMI-instrumented application appears and acts as though it were a microagent in the TIBCO Hawk system.

Interaction with an AMI-instrumented application can use the following means:

- Interactive monitoring using the TIBCO Hawk WebConsole. See Monitoring an Instrumented Application through the TIBCO Hawk WebConsole on page 61 for more information on this.

  An example of this might be using the TIBCO Hawk WebConsole to survey many instances of the instrumented application by means of a network query. Another example might be to make simultaneous changes to many such instances by performing a network action.

- Automating monitoring and management of the application by creating rulebases to be processed by TIBCO Hawk agents.

  An example of this might be creating rulebases to monitor the application's error state, detect critical conditions, and increase the output of debug information until the problem is resolved.

Note: An instrumented application is not dependent on the presence of an TIBCO Hawk agent. The relationship between TIBCO Hawk agent and TIBCO Hawk instrumented application is completely voluntary. The management interface might be active only in some instances of the application, or only at specific times during an instance's activity.

A management interface can be divided into two major portions:

- The initialization code. This code creates a TIBCO Hawk AMI session, passes on the address of a callback function through which messages are to be received, and sets up the functionality to negotiate discovery with a manager.
- A callback function that receives messages from the TIBCO Hawk AMI session. The callback function examines the message and passes it on to one of several internal methods for processing.

Monitoring an Instrumented Application through the TIBCO Hawk WebConsole

The TIBCO Hawk WebConsole is a console application that displays the status of all agents. It is used to view alerts, interact with agents and their microagents, and edit rulebases and other configuration objects. A microagent is an object used by a TIBCO Hawk agent to carry out certain related tasks: to run scripts, to obtain file system information such as free hard disk space or to retrieve a process table. In the TIBCO Hawk WebCosole, instrumented applications appear in the list of microagents, using the name that the application provides through its AMI interface. See Chapter 2, Console API for more information on rulebases and microagents. Any methods thus made visible appear as if they are microagent methods, with arguments and results as described.

Interaction with an instrumented application can occur in the following ways:

- Using the TIBCO Hawk WebConsole, you can interact with an instance of your application from any location in the network.
- Using the TIBCO Hawk WebConsole, you can set up subscriptions to data that your methods provide and use tables and charts to view the results over time.
- Using the TIBCO Hawk WebConsole, you can interact with all instances of your application, across a network, by using network query and network action.
- Using the TIBCO Hawk WebConsole and the rulebase editor, you can create rules to automate monitoring your application from anywhere in the network. Normally, you will create a special rulebase for a managed application and

load it onto all TIBCO Hawk agents on computers where that application resides.

Connecting AMI Participants

An instrumented application can communicate with AMI manager through TIBCO Rendezvous transport. Such instrumented application should use a similar AMI transport configuration to the one being used by its manager.

If you are using a TIBCO Hawk agent as your manager, start the TIBCO Hawk agent with the same `service`, `network`, and `daemon` options for its primary monitoring session as for the TIBCO Rendezvous session set up in the application.

You can also configure the agent to use additional TIBCO Rendezvous sessions to monitor AMI activity. See the *TIBCO Hawk Installation, Configuration, and Administration Guide* for details on configuring agents to use additional AMI sessions.

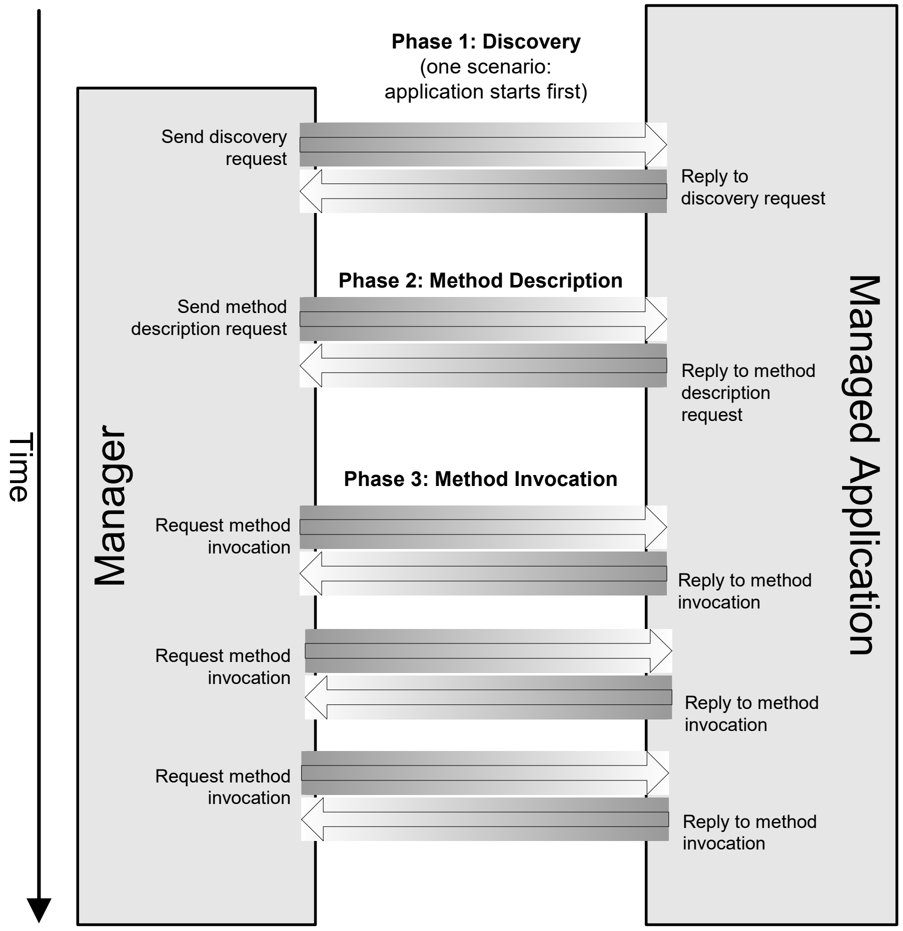
The AMI Conversation

This section describes in detail the three phases involved in communication between an instrumented application and its manager. The three phases of the AMI conversation are:

- discovery
- method description
- method invocation

Figure 10 illustrates the three phases of an AMI conversation.

Figure 10 Three Stages of AMI Conversation


Time
Manager
Managed Application
Phase 1: Discovery
(one scenario:
application starts first)
Send discovery request
Reply to discovery request
Phase 2: Method Description
Send method description request
Reply to method description request
Phase 3: Method Invocation
Request method invocation
Reply to method invocation
Request method invocation
Reply to method invocation
Request method invocation
Reply to method invocation

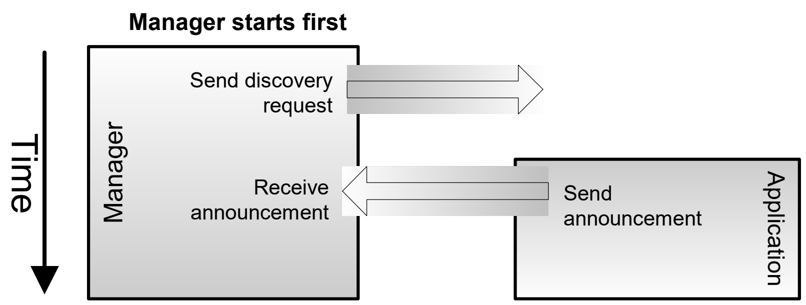
First AMI Phase: Discovering the Application

Discovery between the application and manager can happen in one of two ways, depending on which entity starts first.

If the Manager Starts First

If the manager starts before the application, the interaction is as shown in Figure 11.

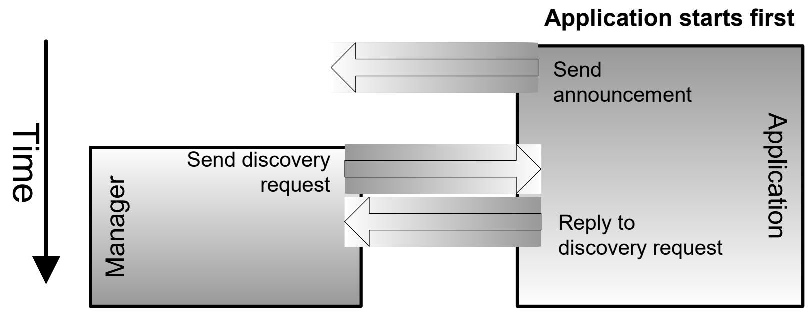
Figure 11 AMI Discovery When the Manager Starts First



If the Application Starts First

If the application starts before the manager, the interaction is as shown in Figure 12.

Figure 12 AMI Discovery when the Application Starts First

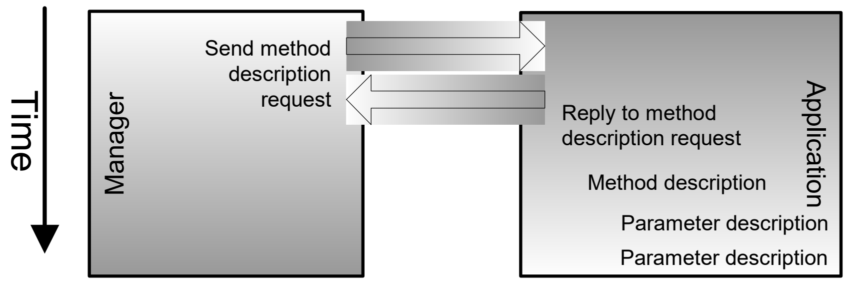


At the end of the discovery phase, the manager and the application have established a connection that allows them to communicate through the inbox address. However, the manager is not yet aware of any details of the application's interface.

Second AMI Phase: Describing the Methods

After the manager and application have exchanged announcement or discovery messages, the manager then sends a message asking the application to describe its methods. Every instrumented application must implement these methods.

Figure 13 AMI Method Description



The message sent to the manager in describe method has nested inside it a series of messages that describe each method. These are known as method descriptor messages. Each method descriptor message has messages nested inside it, which describe the types of arguments and types of returns applicable to each method. This is illustrated in Figure 13. From this description, the manager determines what structure of message to send when invoking the method, as well as how to interpret the message the application sends back.

At the end of the method description phase, the manager is aware of all message exchanges supported by the application and is ready to begin sending messages of the specified types.

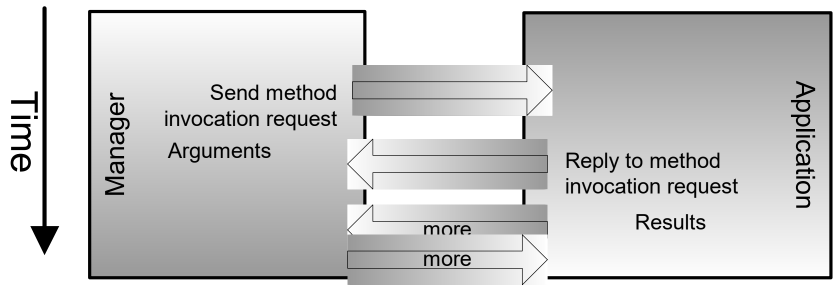
Third AMI Phase: Calling the Methods

After the application has been discovered and the methods have been described, the real work takes place. Acting either on cues from a human user or by processing the rules in a rulebase, the manager sends messages to the application invoking the described methods and awaits its reply. The method invocation messages sent by the manager conform to descriptions given by the application in the describe method as to which arguments are appropriate for each method.

The application sends back a message whose contents conform to the structure the application described to the manager in the describe method.

When the manager receives the response to its method call, it presents the returned information to the human user. This scenario repeats itself over and over again until the session is terminated by either the application or the manager.

Figure 14 AMI Method Invocation


Time
Manager
Send method invocation request
Arguments
Reply to method invocation request
Results
more
more
Application

AMI API Objects

The AMI API facilitates the development of AMI applications written in the Java, C++, and C programming languages. This API makes AMI application development easier and more foolproof because it takes care of AMI transport details for you. The API also ensures that your application will be compatible with future releases of TIBCO Hawk and AMI.

AMI Session

The AMI session handles the AMI application's entire interaction with the AMI manager. Each AMI session manifests itself as a microagent in the associated AMI manager. The API provides AMI manager with functions, which the AMI manager can invoke, to create the session, announce it, stop it, and also to exchange data, events, and errors with the AMI session.

AMI Methods

A method can return data and or perform a task. An AMI application consists of a set of AMI methods that can be invoked by an AMI manager to monitor and manage the AMI application. AMI methods can accept input parameters and return output parameters as required by the method. AMI method provides functions to create and add AMI methods to specific AMI session. The AMI session announces AMI methods (sends their descriptions to an AMI manager) and detects invocations of your AMI methods.

An AMI method can be synchronous or asynchronous. A synchronous AMI method returns data only when invoked by an AMI manager. An asynchronous AMI method can return data whenever data becomes available.

Synchronous Methods

Whenever an AMI manager invokes a synchronous AMI method, the AMI API will call an invocation callback function. You define the invocation callback functions for the synchronous AMI methods. The AMI API provides the callback function with the values of any input parameters and a mechanism for returning either output parameter values or an error. The invocation callback can return data, return no data, or return an error condition.

Asynchronous Methods

For asynchronous AMI methods, an AMI manager informs the AMI method when it should start or stop sending data. The asynchronous AMI method can define callback functions that the AMI API calls whenever an AMI manager has requested the AMI method to start or stop sending data. The AMI API provides the start callback function with a unique context (a subscription object) which identifies the start request, the values of any input parameters, and a mechanism for returning an error. The start callback routine can attach user data to the subscription object. If the start callback function does not return an error then the subscription is in effect and the asynchronous AMI method is free to return data at any time. Using the subscription object, the asynchronous AMI method can return data or error notifications asynchronously to the AMI manager. If the start callback function returns an error the AMI manager cancels the subscription. The AMI API provides the stop callback function with the subscription object and a mechanism for returning an error. The stop callback function can perform any necessary application cleanup. When the stop callback returns the AMI manager stops (cancels) the subscription.

There are two techniques that you can use to implement asynchronous methods. One technique is calling *AmiAsyncMethod::sendData()* for a subscription whenever new data becomes available. The AMI method, AMI method input parameters and any attached user data can always be obtained from the subscription object and output parameters can be constructed using the AMI parameter functions discussed below. Using the start and stop callback functions you must keep track of subscriptions. There can be multiple simultaneous subscriptions to a single asynchronous AMI method, potentially, from multiple AMI managers. Subscription handles are guaranteed to be unique among active subscriptions within the same AMI session and can, therefore, be used as an index for tracking purposes. This technique is often used when a separate thread is created to service each subscription.

The other technique is calling *AmiAsyncMethod::onData()* for a method whenever new data becomes available. This function will call the invocation callback function of the associated method once for each active subscription to that method. The invocation callback function is passed the subscription object allowing the callback to get any attached used data necessary to process the new data. This technique allows the AMI API to do all the work of tracking subscriptions requiring your application to simply provide the invocation callback. The invocation callback can return data, return no data, or return an error condition. If data is returned, it is sent asynchronously to the AMI manager. If no data is returned then no action takes place. This allows the invocation callback function to decide whether the new data should be returned for the subscription. If an error is returned, then it is sent asynchronously to the AMI manager.

Depending on your specific method, this technique may also eliminate the need to supply a start and stop callback. If the subscription object passed to the invocation callback function is null (zero), then the method is being called synchronously. You may return data or an error depending if synchronous invocation is supported by your asynchronous method.

Auto-Invoke Methods

The AMI API provides a mechanism for creating auto-invoke asynchronous methods. During the start asynchronous method callback, the application can set an auto-invoke callback interval for the subscription. If an auto-invoke callback interval is set, the AMI API will call the invocation callback function repeatedly for the method at the specified interval until the subscription is stopped. This can be used to turn a synchronous method into a pseudo-asynchronous method, eliminating the need for the AMI manager to repeatedly invoke the synchronous method. Typically, a method argument is defined to allow the user to specify the interval.

This functionality is critical for synchronous methods that require some setup such as priming counters used to calculate averages, deltas, or percents. Synchronous methods are required to return data on each invocation, making it impossible to return correct data for return values requiring two or more samples on the first invocation. The auto-invoke asynchronous method can return data whenever it is ready. It also has a start and stop callback to perform setup and clean-up operations, respectively. Since the invocation callback function has the option of returning no data, it can skip invocations. This is useful when many data samples are required before accurate data can be returned, or when perhaps it is waiting for some required service to initialize.

AMI Parameter

AMI methods accept input parameters and return output parameters. The AMI API provides functions to define input and output parameters for an AMI method, to set and get the values of those parameters. The API allows settings of multiple values for output parameters allowing multiple (tabular) instances to be returned. If multiple instances (rows) of output parameters are returned then certain output parameters must be defined as indexes. These index parameters must have unique values across output parameter instances returned. If more than one index parameter is defined then the defined indexes are considered a composite index with the primary index specified first.

Error Logging

The AMI API functions can detect and return AMI errors. The AMI error is an object that identifies the specific error and contains handle-based functions that create, destroy, and file stamp AMI errors. The file stamp records a file name and line number in the AMI error to indicate the error source location. Functions are also provided to get the error code, error description, thread ID, file name, and line number associated with the specific AMI error.

Java

The AMI Java API provides rolling trace files to log error or debug messages for your AMI session. The AmiTrace class provides methods to configure and log messages to your trace log. The AMI Java API has built-in tracing, which can be turned on and off, based on trace category (for example, INFO, WARNING, DEBUG).

Threading Model

The AMI API is multi-thread safe and uses multiple threads internally (to guarantee timely processing of AMI protocol related functions). However, it does not impose a threading model on your AMI application. You are free to use the AMI API in a single or multi-threaded application.

AMI API Sample Programs

The TIBCO Hawk software distribution includes sample AMI Java, C++, and C API programs that will help you to better understand how to use the API. These samples can be found in the following directories:

- For Java: *HAWK_HOME*`/examples/ami_api/java`
- For C++: *HAWK_HOME*`/examples/ami_api/cpp`
- For C: *HAWK_HOME*`/examples/ami_api/c.`

The Sample AMI API Applications

The TIBCO Hawk AMI API sample applications can be found in the directory:

HAWK_HOME`/examples/ami_api directory`

This directory contains three sub-directories `c`, `cpp`, and `java` that contain sample applications for the C, C++, and Java TIBCO Hawk AMI APIs, respectively. These applications are discussed in the following sections.

TIBCO Hawk AMI C API Sample Applications

To pass the parameters for executing AMI program, use the properties file hawk_example.props available at `HAWK_HOME/examples/hawk_example.props`. This properties file defines the transport to be used and the parameters required to establish the transport session.

The TIBCO Hawk AMI C API sample directory contains five sample applications. They have the following filenames and AMI application names, with the numbers 1-5 substituted for the x:

TIBCO Hawk AMI C API sample x:

— Filename: `ami_samplex.c`

— Display name: `ami_samplex`

— Internal name: `COM.TIBCO.hawk.ami_api.c.ami_samplex`

The five sample applications can be built by following the instructions in the `Makefile.sample` file, which is also located in the sample directory.

- ami_sample1.c

  This sample shows how to AMI instrument a user application. The AMI API does all the Hawk transport work under the covers.

This method limits the number of dispatching threads to one thread. This is the only thread which will call the user applications AMI API callback functions. As a result, the users application can be single threaded. If the users application is to be multi-threaded then the code in this sample would run on a dedicated thread. The users application would be responsible for thread safety regarding any of its own data structures.

- ami_sample2.c

  This sample shows how to AMI instrument a Rendezvous application. The users application creates a Rendezvous transport and queue and is responsible for dispatching that queue.

  The users application is free to be single or multiple threaded. The users application is responsible for synchronizing access to user application data in the multi-threaded case.

- ami_sample3.c

  This sample is identical to *ami_sample1.c* except that it demonstrates how to create methods that return tabular data.

- ami_sample4.c

  This sample shows how to create an asynchronous AMI method for a synchronous data source. This technique is used when synchronous data needs to be polled at a certain rate, possibly calculations performed on the data across samples, and the results returned at that rate or another rate. This technique makes use of the auto-invoke feature of the AMI C API.

- ami_sample5.c

  This sample shows how to create asynchronous AMI method for a asynchronous data source. The data source sends data using ami_SessionOnData API to send data asynchronously to every subscriber.

Executing Sample Programs

- To execute, for example, `ami_samplex` program, use the following command line:

  ```
  ami_samplex hawk_example.props
  ```

Using Sample Applications on IBM i5/OS

The names of the TIBCO Hawk AMI C API sample programs have been truncated to fit the IBM i5/OS name limit. On IBM i5/OS, the sample programs have the following names, with the numbers 1-4 substituted for the x:

```
ami_samplex.c
```

A sample CL program is provided to compile the sample programs. This sample CL program is included with the HAWKAMI library in QCSRC. The sample CL program assumes the source for the sample programs is present in `TIBHAWK/QCSRC`.

To compile the sample programs, execute the following command:

```
CALL TIBHAWK/AMICMP AMI_SAMPL1
```

You can execute the TIBCO HAWK AMI C sample programs as a job or from `qsh`.

To submit a job, execute the following command:

```
SBMJOB CMD(CALL PGM(TIBHAWK/AMI_SAMPL1)) JOBQ(QUSRNOMAX)
MSGQ(*USRPRF) ALWMLTTHD(*YES)
```

TIBCO Hawk AMI C++ API Sample Application

The TIBCO Hawk AMI C++ API sample directory contains one sample application, which has the following filename and AMI application names:

- AmiSample1.cpp

  Filename: `AmiSample1.cpp`

  Display name: `AmiSample1`

  Internal name: `COM.TIBCO.hawk.ami_api.cpp.AmiSample1`

  This sample creates an AMI session to support methods that demonstrate how to:

  — pass data to an instrumented application,

  — receive data from an instrumented application,

  — return tabular data,

  — return data asynchronously, and

  — shutdown an application.

  The sample application can be built by following the instructions in the `Makefile.sample` file, also located in the sample directory.

Executing the Sample Program

- To execute `AmiSample1` program, use the following command line:

  ```
  AmiSample1 hawk_example.props
  ```

TIBCO Hawk AMI Java API Sample Application

The TIBCO Hawk AMI Java API sample directory contains the following files:

- `AmiSample.Java:`a sample application designed to illustrate how to instrument a Java application using the TIBCO Hawk AMI API for Java.
- `Spot.java`: a simple GUI application using TIBCO Hawk AMI API for Java.
- `SpotAmi.java`: AMI Java API instrumentation for the Spot application.

This sample demonstrates how an existing application, `Spot.java`, can be instrumented with the AMI Java API.

The AMI sample `spot.java` has the following AMI application names:

- Display name: `Spot`
- Internal name: `COM.TIBCO.hawk.ami_api.java.Spot`

Executing the Sample Program

In order to compile and execute this sample, the following must be in your Java `CLASSPATH`:

- `ami.jar` and `util.jar` from the TIBCO Hawk `java` directory
- `tibrvj.jar` from the TIBCO Rendezvous `java` directory

It is recommended that you use Java 1.7 or higher.

The Spot application is executed with the following command:

```
java Spot -rvd_session <service> <network> <daemon> hawk_example.props
```

Using the Sample Programs

Each sample program represents a separate application. Compile and link them using your C/C++ development environment.

Do not use any class libraries (such as MFC) in your build; make the application a console application only.

Programmer's Checklist

C++ Library Files

TIBCO Hawk C++ programs must link the appropriate library files. Choose the appropriate files based on operating system platform.

See Transport Based Library Files on page 77 for appropriate library files based on the transport type.

The table below lists the appropriate library files for the various TIBCO Hawk platforms.

Table 2 Libraries to be Linked

OS platform	Compilation Style	Libraries to Link
Microsoft Windows	/MD compiled console application to be run with DLL	`tibhawkamicpp.lib` `tibhawkami.lib`
Solaris	Shared libraries	`libtibhawkamicpp.so` `libtibhawkami.so`
HP-UX	Shared libraries	`For IA64(Itanium):` `libtibhawkamicpp.so` `libtibhawkami.so`
AIX	Shared libraries	`libtibhawkami.a` `libtibhawkamicpp.a`
Linux	Shared libraries	`libtibhawkamicpp.so` `libtibhawkami.so`
Mac OS	Mac shared libraries	`libtibhawkamicpp.dylib` `libtibhawkami.dylib`

C Library Files

TIBCO Hawk AMI C API programs must link the appropriate library files.

The table below lists the appropriate library files for use on the different TIBCO Hawk platforms.

Table 3 C libraries to be linked

OS platform	Compilation Style	Libraries to link with
Microsoft Windows	/MD compiled console application to be run with DLL	`tibhawkami.lib`
Solaris	Shared libraries	`libtibhawkami.so`
HP-UX	Shared libraries	`For IA64(Itanium):` `libtibhawkami.so`
AIX	Shared libraries	`libtibhawkami.a`
Linux	Shared libraries	`libtibhawkami.so`
Mac OS	Mac shared libraries	`libtibhawkami.dylib`

Transport Based Library Files

TIBCO Rendezvous Based Libraries

The table below lists the appropriate library files based on TIBCO Rendezvous for use on different platforms:

Table 4 TIBCO Rendezvous based libraries to be linked

OS platform	Compilation Style	Libraries to link with
Microsoft Windows	/MD compiled console application to be run with DLL	`librv.lib` `tibhawkamirv.lib`
Linux/Solaris/HP-UX	Shared libraries	`libtibrv.so` `libtibhawkamirv.so`
AIX	Shared libraries	`libtibrv.a` `libtibhawkamirv.a`
Mac OS	Mac shared libraries	`libtibhawkamirv.dylib` `libtibrv.dylib`

Java AMI API Reference

The following link provides access to the TIBCO Hawk AMI API Javadocs.

- AMI API Javadocs

Chapter 5 C++ AMI API Reference

This chapter describes the TIBCO Hawk AMI API C++ class reference. It provides detailed descriptions of each class and method in the TIBCO Hawk.

Topics

AmiSession Class

Class

Declaration `class AmiSession;`

Purpose An instance of `AmiSession` represents an interface to the TIBCO Hawk agent and is treated as a microagent.

Remarks This class can be used as a base class for an application-specific AMI class. The user can inherit from this class and populate the AMI session with methods in the constructor.

The `AmiSession` class will establish point-to-point communication with a TIBCO Hawk agent.

Member Summary

Member	Description	Page
`AmiSession()`	Constructor. Independent of the transport being used.	80
`AmiSession()`	Constructor.	80
`AmiSession::open()`	Initializes the AMI API	86
`AmiSession::close()`	Terminates the AMI API	87
`AmiSession::versionName()`	Returns current version name	88
`AmiSession::version()`	Returns current version	89
`AmiSession::versionDate()`	Returns current version date	90
`AmiSession::banner()`	Returns product banner	91
`AmiSession::versionMajor()`	Returns major version	92
`AmiSession::versionMinor()`	returns minor version	93
`AmiSession::getTraceLevels()`	Returns the current AMI session trace level settings	94
`AmiSession::setTraceLevels()`	Resets AMI session trace level settings	95
`AmiSession::enableTraceLevels()`	Enables levels of race output.	96
`AmiSession::disableTraceLevels()`	Disables levels of trace output	97

Member (Cont'd)	Description (Cont'd)	Page
`AmiSession::announce()`	Announces existence of the microagent.	98
`AmiSession::get... Accessors()`	Returns AmiSession accessors.	99
`AmiSession::sendUnsolicitedMsg()`	Sends unsolicited message to the monitoring agent.	100
`AmiSession::stop()`	Stops the AMI session	101

AmiSession()

Constructor

Declaration

```
AmiSession(ami_TraceCode traceLevel,
           AmiProperty amiProperty,
           const char * name,
           const char * display,
           const char * help,
           ami_TraceHandler traceHandler,
           const void * userData );
```

Purpose This constructor creates an instance of `AmiSession`. Each instance corresponds to an independent microagent.

Parameters

Parameter	Description
`traceLevel`	AMI trace levels for this AMI session. Levels can be: **`AMI_ALL`**. Turns on all trace code levels. **`AMI_AMI`**. Indicates AMI level trace message. **`AMI_DEBUG`**. Logging statement are written into the trace file if, and only if, the trace level set in the current `ami_Session` object has the `AMI_DEBUG` bit mask turned on. **`AMI_ERROR`**. Logging statement are written into the trace file, regardless of whether the `AMI_ERROR` bit mask is turned on in the current `ami_Session` object. **`AMI_INFO`**. Logging statements are written into the trace file if, and only if, the trace level set in the current `ami_Session` object has the `AMI_INFO` bit mask turned on. **`AMI_STAMP`**. Adds source file name and line number to all messages. **`AMI_WARNING`**. Logging statement are written into the trace file if, and only if, the trace level set in the current `ami_Session` object has the `AMI_WARNING` bit mask turned on.
`amiProperty`	Object of AmiProperty Class class.
`name`	Internal name of the microagent.
`display`	User friendly name for the microagent. This name appears in the TIBCO Hawk WebConsole.
`help`	Help text for describing the functions of this microagent.
`traceHandler`	Error callback function used for this AMI session.

Parameter (Cont'd)	Description (Cont'd)
`userData`	User data for this AMI session.

AmiSession()

Constructor

Declaration

```
AmiSession(ami_TraceCode traceLevel,
           const char * service,
           const char * network,
           const char * daemon,
           unsigned int rvTransport,
           unsigned int rvQueue,
           const char * name,
           const char * display,
           const char * help,
           ami_TraceHandler traceHandler,
           const void * userData );
```

Purpose This constructor creates an instance of `AmiSession`. Each instance corresponds to an independent microagent.

Parameters

Parameter	Description
`traceLevel`	AMI trace levels for this AMI session. Levels can be: **`AMI_ALL`**. Turns on all trace code levels. **`AMI_AMI`**. Indicates AMI level trace message. **`AMI_DEBUG`**. Logging statement are written into the trace file if, and only if, the trace level set in the current `ami_Session` object has the `AMI_DEBUG` bit mask turned on. **`AMI_ERROR`**. Logging statement are written into the trace file, regardless of whether the `AMI_ERROR` bit mask is turned on in the current `ami_Session` object. **`AMI_INFO`**. Logging statements are written into the trace file if, and only if, the trace level set in the current `ami_Session` object has the `AMI_INFO` bit mask turned on. **`AMI_STAMP`**. Adds source file name and line number to all messages. **`AMI_WARNING`**. Logging statement are written into the trace file if, and only if, the trace level set in the current `ami_Session` object has the `AMI_WARNING` bit mask turned on.
service network daemon	TIBCO Rendezvous service, network and daemon parameters. For information about setting these parameters, see your TIBCO Rendezvous documentation.
`rvTransport`	C handle for TIBCO Rendezvous `tibrvTransport`.
`rvQueue`	C handle for TIBCO Rendezvous `tibrvQueue` handle.

Parameter (Cont’d)	Description (Cont’d)
`name`	Internal name of the microagent.
`display`	User friendly name for the microagent. This name appears in the TIBCO Hawk WebConsole.
`help`	Help text for describing the functions of this microagent.
`traceHandler`	Error callback function used for this AMI session.
`userData`	User data for this AMI session.

AmiSession::open()

Method

Declaration `static AmiStatus open();`

Purpose Initializes the AMI API.

AmiSession::close()

Method

Declaration	static AmiStatus close();
Purpose	Terminates the AMI C++ API and releases associated resources.

AmiSession::versionName()

Method

Declaration `static const char * versionName();`

Purpose Returns the release name of the application.

AmiSession::version()

Method

Declaration

```
static const char * version();
```

Purpose Returns the release version of the application.

AmiSession::versionDate()

Method

Declaration `static const char * versionDate();`

Purpose Returns the version date of the application.

AmiSession::banner()

Method

Declaration

```
static const char * banner();
```

Purpose Returns the application banner.

AmiSession::versionMajor()

Method

Declaration `static int versionMajor();`

Purpose Returns the major version.

AmiSession::versionMinor()

Method

Declaration `static int versionMinor();`

Purpose Returns the minor version.

AmiSession::getTraceLevels()

Method

Declaration `AmiStatus getTraceLevels(ami_TraceCode * inpTraceLevel) const;`

Purpose Returns the current AMI session trace level settings.

AmiSession::setTraceLevels()

Method

Declaration

```
AmiStatus setTraceLevels(amiTraceCode inTraceLevel);
```

Purpose Resets all AMI Session trace level settings to the specified settings.

AmiSession::enableTraceLevels()

Method

Declaration `AmiStatus enableTraceLevels(amiTraceCode inTraceLevel);`

Purpose Enables the specified level(s) of trace output. All other trace levels settings are unaffected.

AmiSession::disableTraceLevels()

Method

Declaration `AmiStatus disableTraceLevels(amiTraceCode inTraceLevel);`

Purpose Disables the specified level(s) of trace output. All other trace level settings are unaffected.

AmiSession::announce()

Method

Declaration `AmiStatus announce(void) const;`

Purpose Announces the existence of the microagent to the TIBCO Hawk agent.

AmiSession::get... Accessors

Method

Declarations

```
AmiStatus getName(const char ** name) const;

AmiStatus getDisplayName(const char ** displayName) const;

AmiStatus getHelp(const char ** help) const;

AmiStatus getUserData(void ** userData) const;

AmiStatus& getStatus();
```

Purpose Accessors for `AmiSession` objects.

Methods The following table lists the get accessors for `AmiSession` objects.

Method	Description
getName()	Gets the name of this microagent. This name can be different from the display name, as this is the actual name by which the microagent is identified in the TIBCO Hawk system.
getDisplayName()	Gets the display name of the microagent. This is the name as it appears and can be different from the actual name of the microagent. The display name is the user-friendly name by which this interface is to be known, as opposed to the internal interface identifier. The TIBCO Hawk WebConsole shows the display name in the list of discovered microagents/AMI applications.
getHelp()	Gets the help text of this microagent. The returned help text is the optional help text that is displayed to the user.
getUserData()	Returns the user data in the specified AMI session.
getStatus()	Used to check that the session has been created correctly.

AmiSession::sendUnsolicitedMsg()

Method

Declaration

```
AmiStatus sendUnsolicitedMsg(
    ami_AlertType type,
    const char * text,
    int id) const;
```

Purpose Sends an unsolicited message to the monitoring agent.

Remarks An unsolicited message is an application information, warning, or error message that is sent from the managed application directly to the manager (TIBCO Hawk agent).

Parameters

Parameter	Description
`type`	Alert message type (information, warning, or error).
`text`	A text message describing the alert condition.
`id`	An arbitrary identification number defined by the application.

AmiSession::stop()

Method

Declaration `AmiStatus stop(void) const;`

Purpose Stops the AMI session.

AMI Property Class

An instance of AmiProperty class has to be created while creating AMI session to add transport specific properties to AMI session.

- AmiProperty Class, page 103

AmiProperty Class

Class

Declaration `class AmiProperty;`

Purpose This class implements the transport properties to be created. These properties are name value pairs and need to be passed while creating AmiSession.

Member Summary

Member	Description	Page
`AmiProperty()`	Constructor	104
`AmiProperty::()`	Sets the transport properties	105

AmiProperty()

Constructor

Declaration `AmiProperty::AmiProperty();`

Purpose This constructor creates an instance of `AmiProperty`.

AmiProperty::()

Method

Declaration

```
AmiProperty::AmiProperty(
    const char * name,
    void * value);
```

Purpose Sets the transport properties specified as name value pairs.

The following table provides the supported property names and their default values:

Property Name	Mandatory	Default	Description
`hawk_domain`	No	`Default`	The hawk domain.
`hawk_transport`	No	`tibas`	Choice of transport. The available options are: • `tibrv` • `tibtcp`

Properties for TIBCO Rendezvous Transport

Table 5 Properties List for hawk_transport = tibrv

Property Name	Mandatory	Default	Description
`rv_service`	No	`7474`	TIBCO Rendezvous service property.
`rv_network`	No	`;`	TIBCO Rendezvous network property.
`rv_daemon`	No	`tcp:7474`	TIBCO Rendezvous daemon property.
`rv_queue`	No	`Internal-queue`	TIBCO Rendezvous queue for AMI session.
`rv_transport`	No	`Internal-rv-transports`	TIBCO Rendezvous transport for AMI session.

When `hawk_transport = tibrv` and you need to use your own `rv_queue` and `rv_transport`, perform the following steps:

- Explicitly call `tibrv_open()` method before providing a user queue (`rv_queue`).
- Explicitly call `tibrv_close()` method before application exit.

Refer to TIBCO Rendezvous® documentation for more information.

Properties for TCP Transport for TIBCO Hawk

Table 6 Properties list for hawk_transport=tibtcp

Property Name	Mandatory	Default	Description
`tcp_session`	No	`localhost:2591 localhost:2571`	The TCP session parameter for Hawk AMI API.

AMI Method Classes

The methods define the application interface to the TIBCO Hawk agent. When `AmiSession` announces itself to the TIBCO Hawk agent, the agent queries for a description of the available methods. `AmiSession` creates a description of available AMI method objects based on implementations of `AmiMethod`.

`AmiMethod` class provides a foundation for classes that describe synchronous, and asynchronous AMI methods.

The `AmiAsyncMethod` class extends the `AmiMethod` class to send data whenever it becomes available. This allows the AMI-instrumented application to actively publish data whenever data becomes available.

The `AmiSyncMethod` class extends the `AmiMethod` class to return synchronous data. With `AmiSyncMethod` class, the method returns data only upon request.

The `AmiSubcription` class encapsulates an asynchronous method subscription.

AmiMethod Class

Class

Declaration `class AmiMethod`

Purpose This class implements methods.

Remarks Classes derived from `AmiMethod` can be registered with `AmiSession`. The `AmiSyncMethod` class extends the `AmiMethod` class to implement synchronous methods. The `AmiAsyncMethod` class extends the `AmiMethod` class to implement asynchronous methods.

Member Summary

Member	Description	Page
`AmiMethod::setIndexName()`	Sets the index name for this `AmiMethod`.	109

`AmiMethod::get...() Accessors`	Retrieves information from this `AmiMethod` instance.
	`getStatus()`
	`getName()`
	`getHelp()`
	`getSession()`
		110

`AmiMethod::onInvoke()`	Callback on arrival of a method invocation method from the monitoring agent. See AmiParameter Class, page 128.	111

AmiMethod::setIndexName()

Method

Declaration `AmiStatus setIndexName(const char * index);`

Purpose Sets the index field when this `AmiMethod` is to return tabular data. The method can be invoked multiple times to establish the composite index.

Parameters

Parameter	Description
`index`	The index name to be set.

AmiMethod::get...() Accessors

Accessors

Declaration

```
AmiStatus getHelp(const char ** help) const;

AmiStatus getName(const char ** name) const;

AmiSession * getSession(void) const;

AmiStatus& getStatus();
```

Purpose Accessors for retrieving information from an `AmiMethod` instance.

Method	Description
`getHelp()`	Gets the help text for the method.
`getName()`	Gets the name of the method.
`getSession()`	Gets the `AmiSession` object this method belongs to.
`getStatus()`	Used to check if the method object has been created correctly.

See Also `AmiSession Class`, page 80

AmiMethod::onInvoke()

Method

Declaration

```
virtual AmiStatus onInvoke(AmiSubscription * context,
                           AmiParameterListIn * argsIn
                           AmiParameterListOut * argsOut) = 0;
```

Purpose This is a callback method invoked by the C++ API whenever a method invocation message arrives from the monitoring agent.

Remarks Programmers must implement this method to implement the actions to be performed by the application.

Parameters

Parameter	Description
`context`	AMI context associated with invocation.
`argsIn`	Input parameters list from the TIBCO Hawk agent.
`argsOut`	Output parameters list

See Also `AmiParameterList Class`, page 136

AmiAsyncMethod Class

Class

Declaration `class AmiAsyncMethod : public AmiMethod Class`

Purpose The `AmiAsyncMethod` class extends the `AmiMethod` class to implement asynchronous methods.

Member Summary

Member	Description	Page
`AmiAsyncMethod()`	Constructor.	113
`AmiAsyncMethod::onStart()`	Optional virtual method to initiate data flow.	114
`AmiAsyncMethod::onStop()`	Optional virtual method to halt data flow.	114
`AmiAsyncMethod::onData()`	Sends data asynchronously when an event occurs.	116
`AmiAsyncMethod::sendData()`	Sends data asynchronously from an asynchronous AMI method.	117
`AmiAsyncMethod::sendError()`	Reports an error condition for the specified asynchronous method subscription.	119

AmiAsyncMethod()

Constructor

Declaration

```
AmiAsyncMethod(
   AmiSession * session,
   const char * name,
   const char * help,
   ami_MethodType type,
   int inTimeout);
```

Purpose Constructs an instance of `AmiAsyncMethod class.`

Parameters

Parameter	Description
`session`	AMI session object.
name	Name of the method.
`help`	Help text for the method.
`type`	Type of method. Must be one of **`AMI_METHOD_INFO`**, **`AMI_METHOD_ACTION`** or **`AMI_METHOD_ACTION_INFO`**. **`AMI_METHOD_INFO`** methods collect data. Data sources in rulebases and method subscriptions in the TIBCO Hawk WebConsole use this type of method only. **`AMI_METHOD_ACTION`** methods affect the application's behavior in some way. They can be invoked in the TIBCO Hawk WebConsole through interacting with one agent or through a network action. Action methods can also be invoked as an action in a rulebase. **`AMI_METHOD_ACTION_INFO`** methods both make a change to the application and return data. They can be invoked in the TIBCO Hawk WebConsole through interacting with one agent or through a network action.
`inTimeout`	The timeout interval of this AMI method. The default is 10000 milliseconds(10 seconds).

AmiAsyncMethod::onStart()

Method

Declaration

```
virtual AmiStatus onStart(AmiSubscription * context,
                          AmiParameterListIn * args);
```

Purpose This method is invoked by the AMI C++ API whenever an asynchronous method subscription request is made on this method. This method implements the start actions to be performed by the application on such an event

Remarks This method is optional. The default is noop if the application chooses not to implement it. In this case, the AMI session will track the pertinent context for the purpose of sending asynchronous data.

Parameters

Parameter	Description
`context`	AMI context associated with the invocation. The context is specific to a subscription request. The lifetime of this context starts at the moment this method is invoked and stops after the `AmiAsyncMethod::onStop` method returns.
`args`	Input parameters list from the TIBCO Hawk agent.

See Also `AmiParameterList Class`, page 136

AmiAsyncMethod::onStop()

AmiAsyncMethod::onStop()

Method

Declaration

```
virtual AmiStatus onStop(AmiSubscription * content);
```

Purpose

This method is invoked by the AMI C++ API whenever cancellation of asynchronous method subscription arrives for this method. The method implements the stop actions to be performed by the application on such an event. this method is optional.

Parameters

context	Method context

AmiAsyncMethod::onData()

Method

Declaration `void onData();`

Purpose This method sends data asynchronously when an event occurs.

Remarks This method goes through the session context list and sends the data returned by `AmiMethod::onInvoke()` to the appropriate subscription based on the varying input parameters calculated in `AmiMethod::onInvoke()`. This method tracks the context in a way that is transparent to the users.

If not interested in the context or subscription, you must not return any data or call to `AmiParamterListOut::newRow()` in `AmiMethod::onInvoke()`. However, the method should not be suppressed.

It is the user's responsibility to invoke this method when the event occurs.

AmiAsyncMethod::sendData()

Method

Declaration

```
AmiStatus sendData(AmiSubscription * context,
                   AmiParameterListOut * data);
```

Purpose Sends data asynchronously from an asynchronous AMI method.

Parameters

Parameter	Description
`context`	Asynchronous subscription context.
`data`	Reply data to be sent to the subscriber of the asynchronous AMI method.

AmiAsyncMethod::sendError()

Method

Declaration

```
AmiStatus sendError(AmiSubscription * context,
                    AmiStatus& status);
```

Purpose Reports an error condition for the specified asynchronous method subscription.

Parameters

Parameter	Description
`context`	Asynchronous subscription context.
`status`	Reported error condition.

AmiSyncMethod Class

Class

Declaration `class AmiSyncMethod : public AmiMethod Class`

Purpose The `AmiSyncMethod` class extends the `AmiMethod` class to implement synchronous methods.

Member Summary

Member	Description	Page
`AmiSyncMethod()`	Constructor.	120

AmiSyncMethod()

Constructor

Declaration

```
AmiSyncMethod(
   AmiSession * session,
   const char * name,
   const char * help,
   ami_MethodType type,
   int inTimeout);
```

Purpose Constructs an instance of `AmiSyncMethod class`.

Parameters

Parameter	Description
`session`	AMI session object.
name	Name of the method.
`help`	Help text for the method.
`type`	Type of method. Must be one of **`AMI_METHOD_INFO`**, **`AMI_METHOD_ACTION`** or **`AMI_METHOD_ACTION_INFO`**. **`AMI_METHOD_INFO`** methods collect data. Data sources in rulebases and method subscriptions in the TIBCO Hawk WebConsole use this type of method only. **`AMI_METHOD_ACTION`** methods affect the application's behavior in some way. They can be invoked in the TIBCO Hawk WebConsole through interacting with one agent or through a network action. Action methods can also be invoked as an action in a rulebase. **`AMI_METHOD_ACTION_INFO`** methods both make a change to the application and return data. They can be invoked in the TIBCO Hawk WebConsole through interacting with one agent or through a network action.
`inTimeout`	The timeout interval of this AMI method. The default is 10000 milliseconds(10 seconds).

AmiSubscription Class

class

Declaration `class AmiSubscription;`

Purpose The `AmiSubscription` class encapsulates an asynchronous method subscription.

Member Summary

`AmiSubscription::getUserData()`	Allows retrieval of application specific data from a particular asynchronous method subscription	122
`AmiSubscription::setCallbackInterval()`	Indicates that for this subscription the associated `AmiMethod::onInvoke()` callback should be invoked automatically at the specific interval.	123
`AmiSubscription::setUserData()`	Allows attachment of application specific data to a particular asynchronous method subscription.	124
`AmiSubscription::getMethod()`	Allows retrieval of the associated AMI method object.	125
`AmiSubscription::getArguments()`	Allows retrieval of the method argument values for a particular asynchronous method subscription.	126

AmiSubscription::getUserData()

Method

Declaration

```
void * getUserData();
```

Purpose Retrieves the application specific data attached to a particular asynchronous method subscription. This method is usually used in the `AmiMethod::onInvoke()` callback when processing asynchronous method invocations to obtain access to the application specific data associated with that invocation.

AmiSubscription::setCallbackInterval()

Method

Declaration

```
AmiStatus setCallbackInterval(int inInterval);
```

Purpose

Indicates that for this subscription the associated `AmiMethod::onInvoke` callback should be invoked automatically at the specified interval.

Remarks

This provides an asynchronous event to trigger what would normally be synchronous methods so that they can behave as asynchronous methods. A typical scenario is a method that must calculate (polled) data over a precise time interval and return the calculated result based on that interval. In this case the method returns data not based on a synchronous call but on a specified time interval.

Parameters

Parameter	Description
`inInterval`	Interval in seconds. Zero disables the interval.

AmiSubscription::setUserData()

Method

Declaration `AmiStatus setUserData(void * inpUserData);`

Purpose Allows you to attach application specific data to a particular asynchronous method subscription. This function is usually used in the `AmiAsyncMethod::onStart()` callback.

Parameters

Parameter	Description
`inpUserData`	User data.

AmiSubscription::getMethod()

Method

Declaration `AmiMethod * getMethod();`

Purpose Allows retrieval of the associated AMI method object for a particular asynchronous method object.

AmiSubscription::getArguments()

Method

Declaration `AmiParameterListIn * getArguments();`

Purpose Allows retrieval of the method argument values for a particular asynchronous method subscription.

AMI Parameter Classes

The `AmiParameter` class is used to implement the data types to be exchanged between the C++ AMI interface and the TIBCO Hawk agent.

The `AmiParameterIn` class extends the `AmiParameter` class to describe input parameters for the TIBCO Hawk manager's (agent) invocation.

The `AmiParameterOut` class extends the `AmiParameter` class to describe the result parameters returned from method invocation.

The `AmiParameterList` class is used to implement the data types to be exchanged between the C++ AMI interface and the TIBCO Hawk agent.

The `AmiParameterListIn` class lists the complete set of input parameters for an `AMI method`.

The `AmiParameterListOut` class lists the complete set of output parameters for an `AMI method`.

- AmiParameter Class, page 128
- AmiParameterIn Class, page 132
- AmiParameterOut Class, page 134
- AmiParameterList Class, page 136
- AmiParameterListIn Class, page 137
- AmiParameterListOut Class, page 139

AmiParameter Class

Class

Declaration `class AmiParameter;`

Purpose Implements the data types to be exchanged between the `AMI` application and the TIBCO Hawk agent. The parameter can be either `AmiParameterIn` for input parameters from the TIBCO Hawk agent or `AmiParameterOut` for result parameters from method invocations.

Remarks The methods use these boolean definitions:

`AMI_FALSE` = 0

`AMI_TRUE` = 1

Member Summary

Member	Description	Page
`AmiParameter::addChoice()`	Sets the value choices for this `AmiParameter.`	129
`AmiParameter::addLegal()`	Sets the legal choices for `AmiParameter.`	130
`AmiParameter::getStatus()`	Checks if the parameter object was created correctly.	131

AmiParameter::addChoice()

Method

Declaration `AmiStatus addChoice(void* value);`

Purpose Sets the value choices for this parameter.

Remarks Value choices can be displayed by the managing application. If value choices are specified for a parameter, other values are also permitted. For a specified `AmiParameterIn` object, set only one of either choice or legal values. If both are set, the legal value takes precedence.

Parameters

Parameter	Description
`value`	Choice value.

AmiParameter::addLegal()

Method

Declaration `AmiStatus addLegal(void* value);`

Purpose Sets the legal value choices for this parameter.

Remarks Legal value choices can be enforced and displayed by the managing application. If legal value choices are specified for a parameter, no other values are permitted. For a specified `AmiParameterIn` object, set only one of either choice or legal values. If both are set, the legal value takes precedence.

Parameters

Parameter	Description
`value`	Legal choice value.

AmiParameter::getStatus()

Method

Declaration `AmiStatus& getStatus();`

Purpose Checks if the parameter object has been created correctly.

See Also: `AmiStatus Class`, page 144

AmiParameterIn Class

Class

Declaration `class AmiParameterIn : public AmiParameter Class`

Purpose Describes the input parameter from the TIBCO Hawk agent.

Member Summary

Member	Description	Page
`AmiParameterIn()`	Constructor.	133

AmiParameterIn()

Constructor

Declaration

```
AmiParameterIn(AmiMethod * method,
               const char* name,
               ami_DataType type,
               const char* help);
```

Purpose Creates an input parameter for the given method.

Parameters

Parameter	Description
`method`	`AmiMethod` to which this parameter is set.
`name`	Establishes the name of the `AmiParameterIn` object.
`type`	Parameter type. One of: `AMI_I32`. 32-bit signed integer. `AMI_I64`. 64-bit signed integer. `AMI_U64`. 64-bit unsigned integer. `AMI_F64`. 64-bit floating-point number. `AMI_STRING`. Null-terminated character string (UTF8 encoding). `AMI_BOOLEAN`. Boolean.
`help`	Establishes the help text describing the purpose of the `AmiParameterIn` object. `NULL` or empty string values are acceptable. We strongly recommend you specify meaningful descriptions when describing `AmiMethod` input parameters.

AmiParameterOut Class

Class

Declaration `class AmiParameterOut : public AmiParameter Class`

Purpose Creates an output parameter to describe the data that the method returns to the agent.

Member Summary

Member	Description	Page
`AmiParameterOut()`	Constructor.	135

AmiParameterOut()

Constructor

Declaration

```
AmiParameterOut(AmiMethod * method,
                const char* name,
                ami_DataType type,
                const char* help);
```

Purpose Describes the result parameters returned by the method invocation.

Parameters

Parameter	Description
`method`	`AmiMethod` to which this parameter is set.
`name`	Establishes the name of the `AmiParameterOut` object. **Note**: The use of curly brackets { } in microagent method parameter names is not supported. Use of these characters results in an error.
`type`	Parameter type. One of: `AMI_I32`. 32-bit signed integer. `AMI_I64`. 64-bit signed integer. `AMI_U64`. 64-bit unsigned integer. `AMI_F64`. 64-bit floating-point number. `AMI_STRING`. Null-terminated character string (UTF8 encoding). `AMI_BOOLEAN`. Boolean.
`help`	Establishes the help text describing the purpose of the `AmiParameterOut` object. `NULL` or empty string values are acceptable. We strongly recommend you specify meaningful descriptions when describing `AmiMethod` output parameters.

AmiParameterList Class

Class

Declaration `class AmiParameterList;`

Purpose The `AmiParameterList` object contains a list of AMI parameters. It is the parent class for `AmiParameterListIn` and `AmiParamterListOut`.

AmiParameterListIn Class

class

Declaration `class AmiParameterListIn;`

Purpose Defines the input parameter list for an AMI method.

Member Summary

Parameter	Description	
`AmiParameterListIn::getValue()`	parameters for an AMI method	138

AmiParameterListIn::getValue()

Method

Declaration `AmiStatus getValue(const char* name, void* value);`

Purpose Gets the value associated with the input parameter name.

Parameters

Parameter	Description
`name`	Name of parameter being retrieved.
`value`	Target for the retrieved value.

AmiParameterListOut Class

class

Declaration `class AmiParameterListOut;`

Purpose Groups `AmiParameter` objects to define an `AMI method` complete set of output parameters.

Member Summary

Parameter	Description	
`AmiParameterListOut()`	Constructor	140
`AmiParameterListOut::newRow()`	Retrieves a list of returned values to use with `setValue` for method invocation.	141
`AmiParameterListOut::setValue()`	Sets the parameter of the given name with the value.	142

AmiParameterListOut()

Constructor

Declaration `AmiParameterListOut(AmiMethod * method);`

Purpose Creates an `AmiParameterListOut` object to store the returned data used in `AmiAsyncMethod::sendData()`.

AmiParameterListOut::newRow()

Method

Declaration `AmiStatus newRow();`

Purpose Gets a list of returned values for the method, the current `AmiParameterListOut` object, then can call to `AmiParameterListOut::setValue` to set the return values for a method invocation. This function can be called multiple times to return multiple rows of data.

AmiParameterListOut::setValue()

Method

Declaration `AmiStatus setValue(const char * name void * value);`

Purpose Sets the parameter of the given name with the value

Parameters

Parameter	Description
`name`	Name of parameter being set
`value`	Value to set

AMI Error Handling

The `AmiStatus` class encapsulates information about `AmiStatus` conditions to aid in error tracing.

- AmiStatus Class, page 144

AmiStatus Class

Class

Declaration `class AmiStatus;`

Purpose Objects of this class also are used to return an error condition to the monitoring TIBCO Hawk agent.

Member Summary

Member	Purpose	Page
`AmiStatus()`	Constructor	146
`operator ami_Error() const`	Operator to convert an `ami_Error` to `AmiStatus`.	—
`AmiStatus& operator=(const AmiStatus& status)`	Assignment operator.	—
`AmiStatus& operator=(const ami_Error status)`	Assignment operator.	—
`AmiStatus& operator=(const ami_ErrorCode errorCode)`	Assignment operator.	—
`ami_Boolean operator==(const AmiStatus& status)`	Comparison operator.	—
`ami_Boolean operator!=(const AmiStatus& status)`	Comparison operator.	—
`ami_Boolean operator==(const ami_Error status)`	Comparison operator.	—
`ami_Boolean operator!=(const ami_Error status)`	Comparison operator.	—
`ami_Boolean operator!() const`	Comparison operator.	—
`ami_Boolean ok(void) const`	Evaluates whether this status object— indicates an error state.	—
`AmiStatus::getAmiError()`	Returns the C API error handle of this AmiStatus object.	147
`AmiStatus::setStatus()`	Creates a new AMI error for the specified error code and descriptive text.	148
`AmiStatus::setStatusV()`		149
`AmiStatus::stamp()`	Stamps AMI error for location ID	150
`AmiStatus::getCode()`	returns the AMI C API error code	151

Member (Cont’d)	Purpose (Cont’d)	Page
`AmiStatus::getText()`	Returns the textual description of the error	152
`AmiStatus::getThread()`	Returns the thread ID of the thread which created the specified AMI error	153
`AmiStatus::getFile()`	Returns error’s source file name	154
`AmiStatus::getLine()`	Returns the line number of the error source	155

AmiStatus()

Constructor

Declaration

```
AmiStatus();

AmiStatus(ami_Error status);

AmiStatus(ami_ErrorCode code);
```

Purpose Creates an instance of `AmiStatus` object.

Parameters

Parameter	Description
`status`	Handle to `ami_Error`.
`code`	AMI C error code.

AmiStatus::getAmiError()

Method

Declaration `ami_Error getAmiError();`

Purpose Gets the AMI C API error handle of this status object.

AmiStatus::setStatus()

Method

Declaration

```
void setStatus(int errorCode, const char * format, ...);
```

Purpose

Creates a new AMI error for the specified error code and descriptive text. Descriptive text is specified as a template (`printf` format) and substitution arguments. If error creation fails then an error describing this failure is returned in place of the specified error.

Parameters

Parameter	Description
`errorCode`	AMI C error code.
`format`	Descriptive template.

AmiStatus::setStatusV()

Method

Declaration

```
void setStatusV(int errorCode, const char * format, va_list args);
```

Purpose

Creates a new AMI error for the specified error code and descriptive text. Descriptive text is specified as a template (`printf` format) and substitution arguments. If error creation fails then an error describing this failure is returned in place of the specified error.

Parameters

Parameter	Description
`errorCode`	Error code.
`format`	Error description template.
`args`	Error template substitution arguments.

AmiStatus::stamp()

Method

Declaration

```
void stamp(const char * inpFilename, in inLineNumber);
```

Purpose Stamps the specified AMI error with the specified file name and line number.

AmiStatus::getCode()

Method

Declaration `int getCode();`

Purpose Returns the error code of the specified AMI error.

AmiStatus::getText()

Method

Declaration `const char * getText();`

Purpose Returns the textual description of the specified AMI error.

AmiStatus::getThread()

Method

Declaration

```
int getThread();
```

Purpose Returns the thread ID of the thread which created the specified AMI error.

AmiStatus::getFile()

Method

Declaration

```
const char * getFile();
```

Purpose Returns the name of the source file which generated the specified AMI error.

AmiStatus::getLine()

Method

Declaration

```
int getLine();
```

Purpose Returns the line number of the source file which generated the specified AMI error.

Chapter 6 C AMI API Reference

This chapter describes the TIBCO Hawk AMI API C class references. It explains the constants, error codes, data types and functions related to error handling, tracing, callbacks and initializing the API.

Topics

Data Types Summary

This table lists the AMI C API data types. These types are described in the following sections.

Data Type	Description	Page
`ami_AlertType`	Defines the valid alert types for unsolicited messages.	160
`ami_Boolean`	Defines the valid boolean data types.	161
`AMI C API Constants`	General constants defined in the AMI C API.	162
`ami_DataType`	Defines valid AMI C API parameter data types.	163
`ami_Error`	Error object handle—encapsulates all the information required to define an AMI C API error.	164
`ami_Method`	Method object handle—Encapsulates all the information required to define and support an AMI C API method.	166
`ami_MethodType`	Defines the valid AMI C API method types.	167
`ami_Parameter`	Parameter object handle—encapsulates all the information required to define and support an AMI C API parameter—used to represent an individual method input or output parameter.	168
`ami_ParameterList`	Object handle for parameter list that encapsulates all the information required to define and support a list of AMI C API parameters—used to group the input or output parameters of a method.	169

Data Type (Cont'd)	Description (Cont'd)	Page
`ami_ParameterListList`	Object handle that generates a list of Parameter lists—encapsulates all the information required to define and support multiple AMI parameter lists—used to return multiple instances of output parameter values for a method (i.e. tabular data).	170
`ami_Session`	Session object handle—encapsulates all the information required to define and support an AMI session.	171
`ami_Subscription`	Encapsulates all the information required to define and support a subscription to an asynchronous AMI method.	172
`ami_Property`	Defines the transport properties specified by name value pairs and are added by the `ami_AddProperty`.	173

ami_AlertType

Type

Purpose Alert classifications for unsolicited messages, specifying the supported AMI unsolicited message types. An unsolicited message is an application information, warning, or error message that is sent from the managed application directly to the manager. These relate directly to the TIBCO Hawk alert classifications.

Alert Type

Type	Definition
`AMI_ALERT_INFO`	Specifies an informational alert.
`AMI_ALERT_WARNING`	Specifies a warning alert.
`AMI_ALERT_ERROR`	Specifies an error alert.

ami_Boolean

Type

Purpose This type specifies valid boolean values

Enumeration

Declaration	Description
`AMI_FALSE`	Boolean value is false
`AMI_TRUE`	Boolean value is true

AMI C API Constants

The next table lists the general AMI C API constants.

Constant	Description
`AMI_VERSION`	The version of the AMI specification implemented by this version of the AMI C API.
`AMI_METHOD_DEFAULT_TIMEOUT`	AMI method default timeout period specified in milliseconds. If the method takes longer than this timeout period the TIBCO Hawk agent will reflect a timeout error. For methods that require a longer timeout period this can be overridden by passing the timeout parameter through the `ami_MethodCreate` function The default is 10000 milliseconds.

ami_DataType

Type

Purpose Defines the valid AMI C API parameter data types.

Enumeration

Declaration	Description
`AMI_I32`	32-bit signed integer.
`AMI_I64`	64-bit signed integer.
`AMI_F64`	64-bit floating-point number.
`AMI_U64`	64-bit unsigned integer.
`AMI_STRING`	Null-terminated character string (UTF8 encoding).
`AMI_BOOLEAN`	Boolean.
`AMI_INTEGER`	Use for backwards compatibility with previous AMI C API versions. AMI_I32 is preferred.
`AMI_LONG`	Use for backwards compatibility with previous AMI C API versions. AMI_I64 is preferred.
`AMI_REAL`	Use for backwards compatibility with previous AMI C API versions. AMI_F64 is preferred.

ami_Error

Type

Purpose This is the AMI C API error object handle. It encapsulates all the information required to define and process errors generated by the AMI C API and for the user's application to pass errors to the AMI C API.

Remarks A null `ami_Error` handle indicates success (i.e. no error). The convenience define `AMI_OK` is provided representing an `ami_Error` indicating success. It is recommended that you use an expression like the following when testing an `ami_Error` for success:

```
ami_Error RC;
RC = ami_Function();
if ( RC != AMI_OK )
{
   handle error condition
}
```

A non-null `ami_Error` indicates an error. The unique error code identifying the error can be obtained using the `ami_ErrorGetCode()` function.

The `ami_Error` is an object handle representing allocated resources and must be destroyed using the `ami_ErrorDestroy()` function or memory will be leaked. The AMI C API application must destroy any `ami_Error` instances returned by an AMI C API function call. It must also destroy any `ami_Error` instance it explicitly creates with the exception of `ami_Error` instances returned to the AMI C API from a method invocation callback function.

The unique error codes returned by the `ami_ErrorGetCode()` function are documented in the following table.

Error Code	Description
`AMI_AMIERROR_FAILURE`	Unable to create AMI error due to memory allocation failure
`AMI_INSUFFICIENT_MEMORY`	Insufficient memory to process request
`AMI_INVALID_ERROR`	Specified AMI error handle is invalid
`AMI_CORRUPT_ERROR`	Specified AMI error handle is invalid or corrupted
`AMI_MISSING_ARGUMENT`	Required argument not specified (null)
`AMI_INVALID_ARGUMENT`	Invalid argument specified
`AMI_INVALID_SESSION`	Specified AMI session handle (bad handle) is invalid
`AMI_CORRUPT_SESSION`	Specified AMI session handle (bad handle) is invalid or corrupted

Error Code (Cont’d)	Description (Cont’d)
`AMI_INVALID_METHOD`	Specified AMI method handle (bad handle) is invalid
`AMI_CORRUPT_METHOD`	Specified AMI method handle (bad handle) is invalid or corrupted
`AMI_INVALID_SUBSCRIPTION`	Specified AMI subscription handle (bad handle) is invalid
`AMI_CORRUPT_SUBSCRIPTION`	Specified AMI subscription handle (bad handle) is invalid or corrupted
`AMI_INVALID_PARM_TABLE`	Specified AMI parameter list handle (bad handle) is invalid
`AMI_CORRUPT_PARM_TABLE`	Specified AMI parameter list handle (bad handle) is invalid or corrupted
`AMI_INVALID_PARM_LIST`	Specified AMI parameter list handle (bad handle) is invalid
`AMI_CORRUPT_PARM_LIST`	Specified AMI parameter list handle (bad handle) is invalid or corrupted
`AMI_INVALID_PARAMETER`	Specified AMI parameter handle (bad handle) is invalid
`AMI_CORRUPT_PARAMETER`	Specified AMI parameter handle (bad handle) is invalid or corrupted
`AMI_RV_ERROR`	TIBCO Rendezvous error (RV error number): (RV error text)
`AMI_UNKNOWN_INVOCATION`	Received invocation request for unknown AMI method (method name)
`AMI_UNKNOWN_PARAMETER`	Method (method name) does not have a parameter named (parameter name)
`AMI_LIST_ADD_FAILED`	Failed to add object to linked list
`AMI_ARGUMENT_GET_FAILED`	TIBCO Rendezvous error (RV error number) occurred attempting to get value for argument (argument name) of method (method name). (RV error text)
`AMI_UNKNOWN_SUBSCRIPTION`	(Method name) invocation received for unknown subscription with context (context ID) and reply subject (subject name)
`AMI_SESSION_ANNOUNCED`	Attempt made to announce an AMI session which is already announced.
`AMI_SESSION_STOPPED`	Attempt made to stop an AMI session which has not been announced

ami_Method

Type

Purpose This is the AMI C API method object handle. It encapsulates all the information required to define and support an AMI method. It is used to identify a specific AMI method in AMI C API functions.

Remarks All `ami_Method` instances are associated with a specific `ami_Session` instance and are destroyed along with the `ami_Session` when `ami_SessionDestroy()` is called. There is no need (or function) to destroy an `ami_Method`.

ami_MethodType

Type

Purpose Defines the valid AMI C API method types.

Enumeration

`AMI_METHOD_INFO`	Returns data
`AMI_METHOD_ACTION`	Performs an action
`AMI_METHOD_ACTION_INFO`	Performs an action and returns data

ami_Parameter

Type

Purpose This is the AMI C API parameter object handle. It encapsulates all the information required to define and support an AMI parameter. It is used to identify a specific method input or output parameter in AMI C API functions.

Remarks All `ami_Parameter` instances are associated with an `ami_ParameterList` and are destroyed along with the `ami_ParameterList` when it is destroyed. There is no need (or function) to destroy an `ami_Parameter`.

ami_ParameterList

Type

Purpose This is the AMI C API parameter list object handle. It encapsulates all the information required to define and support a list of AMI parameters. It is used to group parameters for the purpose of defining the input or output parameters of a method and for setting and retrieving the values of input and output parameters.

Remarks All `ami_ParameterList` instances are associated with an `ami_Method` or an `ami_ParameterListList` and are destroyed along with the `ami_Method` or `ami_ParameterListList` when it is destroyed. There is no need (or function) to destroy an `ami_ParameterList`.

ami_ParameterListList

Type

Purpose This is the AMI C API list of parameter lists object handle. It encapsulates all the information required to define and support multiple AMI parameter lists. It is used to return multiple instances of output parameter values for a method (i.e. tabular data).

Remarks An `ami_ParameterListList` instance is passed to the invocation callback function of a method to allow that method to return zero or more `ami_ParameterList` instances containing the return values for the method invocation. The AMI C API provides functions for adding `ami_ParameterList` instances to the `ami_ParameterListList` instance as required by the method. On return from the method invocation callback function the AMI C API sends the output parameter values (if any) associated with the `ami_ParameterListList` to the AMI manager and then destroys the `ami_ParameterListList`.

The AMI C API provides functions to create and destroy `ami_ParameterListList` instances for use in asynchronous methods where no method invocation callback is involved. The AMI C API application is responsible for destroying any `ami_ParameterListList` instances it explicitly creates using `ami_ParameterCreateOut()`.

ami_Session

Type

Purpose This is the AMI C API session object handle. It encapsulates all the information required to define and support an AMI session.

Remarks The `ami_Session` is required on all AMI C API function calls to identify the AMI session. An AMI C API application may have one or more `ami_Session` instances, each representing a different Hawk microagent.

ami_Subscription

Function

Purpose This is the AMI C API subscription object handle. It encapsulates all the information required to define and support a subscription to an asynchronous AMI method.

Remarks An asynchronous method sends data whenever it becomes available. An AMI manager informs (starts) the asynchronous method when it is interested in receiving this data, in other words, it subscribes to the asynchronous method. The asynchronous method will send data as long as the subscription is in effect and will no longer send data when that subscription is terminated (stopped). An AMI C API asynchronous method specifies callback functions that are called when a subscription is started and when it is stopped. An `ami_Subscription` instance is provided in these callbacks to be used by the application to identify the subscription in AMI C API functions.

ami_Property

Type

Declaration

```
typedef struct amiProperty{
      char* name;
      void* value;
      struct amiProperty * next;
}ami_Property;
```

Purpose This data-structure helps creating AMI Transport properties link-list using `ami_AddProperty` AMI API, which will be passed to `AMI Session create` API.

AMI C API Error Functions

This chapter describes the error, callback, trace control, initialization and termination handling functions.

Error Functions Summary

The following table summarizes the error handling functions. These functions are described in the following sections.

Function	Description	Page
`ami_ErrorCreate()`, `ami_errorCreateV()`	Creates a new AMI error for the specified error code and descriptive text.	176
`ami_ErrorDestroy()`	Destroys the specified AMI error.	177
`ami_ErrorStamp()`	Stamps the specified AMI error.	178
`ami_ErrorGet... Accessors`	Accessor methods	179

ami_ErrorCreate(), ami_errorCreateV()

Functions

Declaration

```
ami_Error ami_ErrorCreate(
   int            inErrorCode,
   const char *   inpTemplate,
   ... );

 ami_Error ami_ErrorCreateV(
   int            inErrorCode,
   const char *   inpTemplate,
   va_list        inArguments );
```

Purpose

Creates a new AMI error for the specified error code and descriptive text. Descriptive text is specified as a template (`printf` format) and substitution arguments. If error creation fails then an error describing this failure is returned in place of the specified error.

The `ami_Error` is an object handle representing allocated resources and must be destroyed using the `ami_ErrorDestroy()` function or memory will be leaked. The AMI C API application must destroy any `ami_Error` instances returned by an AMI C API function call. It must also destroy any `ami_Error` instance it explicitly creates with the exception of `ami_Error` instances returned to the AMI C API from a method invocation callback function.

Parameters

Parameter	Description
`inErrorCode`	Error code.
inpTemplate	Error description template.
inArguments	Error template substitution arguments.

ami_ErrorDestroy()

Function

Declaration

```
void ami_ErrorDestroy(
   ami_Error     inAmiError );
```

Purpose Destroys the specified AMI error. Can be called with a NULL handle that is ignored.

Parameters

Parameter	Description
`inAmiError`	Handle of AMI error.

ami_ErrorStamp()

Function

Declaration

```
void ami_ErrorStamp(
   ami_Error      inAmiError,
   const char *   inpFilename,
   int            inLineNumber );
```

Purpose Stamps the specified AMI error with the specified file name and line number to identify the location in the code where this error was generated.

Parameters

Parameter	Description
`inAmiError`	AMI error object to stamp.
`inpFilename`	Source file name in which error occurred.
`inLineNumber`	Source file line number where error occurred.

ami_ErrorGet... Accessors

Function

Declaration

```
int ami_ErrorGetCode(
   ami_Error     inAmiError );

const char * ami_ErrorGetText(
   ami_Error     inAmiError );

int ami_ErrorGetThread(
   ami_Error     inAmiError );

const char * ami_ErrorGetFile(
   ami_Error     inAmiError );

int ami_ErrorGetLine(
   ami_Error     inAmiError );
```

Accessor Functions

Function	Description
`ami_ErrorGetCode`	Returns the AMI C API error code of the specified AMI error handle.
`ami_ErrorGetText`	Returns the textual description of the specified AMI error. This function always returns a description (never NULL). If no description was specified in the create call then a default message is used which states that no description is available.
`ami_ErrorGetThread`	Returns the thread ID of the thread which created the specified AMI error.
`ami_ErrorGetFile`	Returns the name of the source file which generated the specified AMI error. This function can return a NULL pointer if the specified error was not file stamped.
`ami_ErrorGetLine`	Returns the line number of the source file which generated the specified AMI error. This function could return zero if the specified error was not file stamped.

Callback Function Types Summary

The following table summarizes the AMI C API callback function types. These types are described in the following sections.

Function	Description	Page
`ami_OnInvokeCallback`	Prototype for an AMI method callback function.	181
`ami_OnStartCallback`	Prototype for the optional AMI asynchronous method on start callback function.	182
`ami_OnStopCallback`	Prototype for the optional AMI asynchronous method on stop callback function.	183
`ami_TraceHandler`	Prototype for the optional AMI trace handler callback.	184

ami_OnInvokeCallback

Type

Declaration

```
typedef ami_Error (*ami_OnInvokeCallback)(
   ami_Session                inAmiSession,
   ami_Method                 inAmiMethod,
   ami_Subscription           inAmiSubscription,
   void *                     inpUserData,
   ami_ParameterList          inArguments,
   ami_ParameterListList *    inpReturns );
```

Purpose

This is the prototype for an AMI method callback function. These functions are invoked whenever the associated method is executed by TIBCO Hawk. The callback should return method return values or an `ami_Error` if the function call fails. When this callback is executed for an asynchronous method the `inAmiSubscription` argument is provided. For synchronous or synchronous invocations of asynchronous methods this argument is NULL.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inAmiSubscription`	Asynchronous method subscription.
`inpUserData`	User data associated with the method.
`inArguments`	AMI input parameter handle.
`inpReturns`	Target for method return values.

ami_OnStartCallback

Type

Declaration

```
typedef ami_Error (*ami_OnStartCallback)(
   ami_Session       inAmiSession,
   ami_Method        inAmiMethod,
   void *            inpUserData,
   ami_Subscription  inAmiSubscription,
   ami_ParameterList inArguments ); /
```

Purpose This is the prototype for the optional AMI asynchronous method on start callback function. These functions are called whenever a new subscription is started. The application should perform any necessary initialization required to process this new subscription. This callback should return AMI_OK if no error, otherwise an `ami_Error` describing the failure.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpUserData`	User data associated with the method.
`inAmiSubscription`	Asynchronous method subscription.
`inArguments`	AMI input parameter handle.

ami_OnStopCallback

Type

Declaration

```
typedef void (*ami_OnStopCallback)(
   ami_Session        inAmiSession,
   ami_Method         inAmiMethod,
   void *             inpUserData,
   ami_Subscription   inAmiSubscription );
```

Purpose This is the prototype for the optional AMI asynchronous method on stop callback function. These functions are called whenever a subscription is stopped. The application should perform any necessary clean-up required when terminating a subscription. This callback should return AMI_OK if no error, otherwise an `ami_Error` describing the failure.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpUserData`	User data associated with the method.
`inAmiSubscription`	Asynchronous method subscription.

ami_TraceHandler

Type

Declaration

```
typedef void (*ami_TraceHandler)(
   ami_Session     inAmiSession,
   ami_TraceCode   inTraceCode,
   int             inTraceID,
   const char *    inpText,
   void *          inpUserData );
```

Purpose

This is the prototype for the optional AMI trace handler callback. This callback is used by AMI API to report events to the application. These events are classified by `ami_TraceCode`. If no trace handler is provided then tracing is disabled. Tracing can be controlled (including turned off) using the trace control functions.

Parameters

Parameter	Description
`inAmiSession`	AMI session handle reporting the trace event.
`inTraceCode`	Category of trace event.
`inTraceID`	Unique ID of trace event.
`inpText`	Textual description of trace event
`inpUserData`	User data associated with the AMI session.

Trace Control Functions Summary

The following table summarizes the AMI C API trace control functions. These functions are described in the following sections.

Function	Description	Page
`ami_TraceCode`	AMI C API trace levels.	186
`ami_SessionGetTraceLevels()`	Returns the current AMI session trace level settings.	187
`ami_SessionSetTraceLevels()`	Resets all AMI session trace level settings to the specified settings.	188
`ami_SessionEnableTraceLevels()`	Enables the specified level(s) of trace output.	189
`ami_SessionDisableTraceLevels()`	Disables the specified level(s) of trace output.	190

ami_TraceCode

Type

Declaration

```
typedef enum
 { AMI_INFO    =  1,
   AMI_WARNING =  2,
   AMI_ERROR   =  4,
   AMI_DEBUG   =  8,
   AMI_AMI     = 16,
   AMI_STAMP   = 32,
   AMI_ALL     = 0x7FFFFFFF
 } ami_TraceCode;
```

Purpose

AMI C API trace levels. All trace messages output by the AMI C API are classified under one of the following trace levels. When a trace message is generated it is passed to the `ami_TraceHandler` of the associated AMI session only if the corresponding trace level is enabled. This allows for programmatic control of the level of tracing performed.

These values may be `OR'ed` together when used as arguments in functions that take an `ami_TraceCode`.

Trace Levels

Level	Description
`AMI_ALL`	This is a convenience value for enabling or disabling all levels.
`AMI_AMI`	This trace level enables low level tracing of AMI operations. This level aids the investigation of problems related to AMI and should not be enabled under normal circumstances.
`AMI_DEBUG`	This level increases the detail of information in trace output to aid in investigation of problems. This level is for troubleshooting purposes only, and under normal circumstances should not be enabled.
`AMI_ERROR`	This level enables tracing of error messages. This level should be enabled at all times.
`AMI_INFO`	This level enables tracing of informational messages. This level can be enabled at all times.
`AMI_STAMP`	This level adds source file name and line number to trace messages to determine the exact source of trace messages. This level is for troubleshooting purposes only, and under normal circumstances should not be enabled.
`AMI_WARNING`	This level enables tracing of warning messages. This level can be enabled at all times.

ami_SessionGetTraceLevels()

Function

Declaration

```
ami_Error ami_SessionGetTraceLevels(
   ami_Session     inAmiSession,
   ami_TraceCode * inpTraceLevel );
```

Purpose Returns the current AMI session trace level settings.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inpTraceLevel`	Target for returned trace levels.

ami_SessionSetTraceLevels()

Declaration

```
ami_Error ami_SessionSetTraceLevels(
   ami_Session    inAmiSession,
   ami_TraceCode  inTraceLevel );
```

Purpose Resets all AMI session trace level settings to the specified settings. If a trace level is not specified, it is disabled.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inTraceLevel`	Trace levels to set.

ami_SessionEnableTraceLevels()

Function

Declaration

```
ami_Error ami_SessionEnableTraceLevels(
   ami_Session    inAmiSession,
   ami_TraceCode  inTraceLevel );
```

Purpose Enables the specified level(s) of trace output. All other trace level settings are not effected.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inTraceLevel`	Trace level(s) to enable.

ami_SessionDisableTraceLevels()

Function

Declaration

```
ami_Error ami_SessionDisableTraceLevels(
   ami_Session    inAmiSession,
   ami_TraceCode  inTraceLevel );
```

Purpose Disables the specified level(s) of trace output. All other trace level settings are not effected.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inTraceLevel`	Trace level(s) to disable.

Initialization and Termination Functions Summary

The following table summarizes the AMI C API functions for returning version information and starting and stopping the API. These functions are described in the following sections.

Function	Description	Page
`ami_Version... Accessors`	These functions return the AMI C API version information.	192
`ami_Open()`	Initializes the AMI C API.	193
`ami_Close()`	Terminates the AMI C API.	194

ami_Version... Accessors

Functions

Declaration

```
const char * ami_VersionName();
const char * ami_Version();
const char * ami_VersionDate();
int          ami_VersionMajor();
int          ami_VersionMinor();
int          ami_VersionUpdate();
```

Purpose These functions return the AMI C API version information. The version information consists of a major, minor, and update number formatted left to right, respectively like this 3.1.1.

Accessors

Function	Description
ami_VersionName()	Returns the product name.
ami_Version()	Returns the version string, for example, 3.1.1.
ami_VersionDate()	Returns the build date.
ami_VersionMajor()	Returns the major version number.
ami_VersionMinor()	Returns the minor version number.
ami_VersionUpdate()	Returns the update version number.

ami_Open()

Function

Declaration `ami_Error ami_Open();`

Purpose Initializes the AMI C API. Must be called prior to calling any other AMI C API functions.

ami_Close()

Function

Declaration `ami_Error ami_Close();`

Purpose Terminates the AMI C API and releases associated resources.

AMI C API Session Functions

This chapter describes the AMI C API session functions. Each AMI session manifests itself as a microagent in the associated AMI manager. The AMI C API session object (`ami_Session`) encapsulates an AMI session. The API provides handle-based functions to create, announce, stop, and destroy session objects.

- Session Functions Summary, page 197

Session Functions Summary

This table summarizes the AMI C API session functions. These functions are described in the following sections.

Function	Description	Page
`ami_SessionCreateUsingProperties()`	Creates a new AMI session using transport properties.	199
`ami_AddProperty()`	Adds transport properties.	200
`ami_SessionCreate()`	Creates a new AMI session.	202
`ami_SessionDestroy()`	Destroys the AMI session.	204
`ami_SessionAnnounce()`	Activates the AMI session.	205
`ami_SessionStop()`	Stops (deactivates) the AMI session.	206
`ami_SessionGetName()`	Gets the name string of AMI session (microagent).	207
`ami_SessionGetDisplayName()`	Get the user-friendly name string of AMI session (microagent).	208
`ami_SessionGetHelp()`	Gets the descriptive text string of AMI session (microagent).	209
`ami_SessionGetUserData()`	Returns the user data of the specified AMI session.	210
`ami_SessionSendData()`	Returns data for the specified asynchronous method subscription.	211
`ami_SessionSendError()`	Reports an error condition for the specified asynchronous method subscription.	212
`ami_SessionOnData()`	Calls the `ami_OnInvokeCallback` function of the specified AMI asynchronous method once for each currently active subscription.	213

Function	Description	Page
`ami_SessionSendUnsolicitedMsg()`	Send an unsolicited message to any interested subscribers.	214

ami_SessionCreateUsingProperties()

Function

Declaration

```
ami_Error ami_SessionCreateUsingProperties(
   ami_Session *     inpAmiSession,
   ami_TraceCode     inTraceLevel,
   ami_Property *    inpProperties,
   const char *      inpName,
   const char *      inpDisplayName,
   const char *      inpHelp,
   ami_TraceHandler inTraceHandler,
   const void *      inpUserData );
```

Purpose

Creates a new AMI session using transport properties. Each session represents a single TIBCO Hawk microagent.

Create property list using the ami_AddProperty() API and pass it to the `ami_SessionCreateUsingProperties()` API.

Parameters

Parameter	Description
`inpAmiSession`	Target for returned session handle.
`inTraceLevel`	AMI trace levels for this AMI session. See ami_TraceCode on page 186 for trace level descriptions.
`inpProperties`	AMI transport properties.
`inpName`	Unique name string for microagent.
`inpDisplayName`	User-friendly name string for microagent.
`inpHelp`	User-friendly microagent description.
`inTraceHandler`	AMI session trace callback function.
`inpUserData`	AMI session user data.

ami_AddProperty()

Function

Declaration

```
ami_Error ami_AddProperty(
   const char *    inpName,
   void *          inpValue,
   ami_Property **  inpProperties );
```

Purpose Creates a linked list defined as `ami_Property and` adds transport properties to it. The `ami_AddProperty()` API allocates memory and creates a linked list for every parameter added.

Create property list using the `ami_AddProperty()` API and pass it to ami_SessionCreateUsingProperties() API.

Parameters

Parameter	Description
`inpName`	Name of the parameter.
`inpValue`	Value of the parameter.
`ami_Property`	Reference to the property.

The following table provides the supported property names and their default values:

Property Name	Mandatory	Default	Description
`hawk_domain`	No	`Default`	The hawk domain.
`hawk_transport`	No	`tibas`	Choice of transport. The available options are: • `tibrv`

Table 7 Properties List for hawk_transport = tibrv

Property Name	Mandatory	Default	Description
`rv_service`	No	`7474`	TIBCO Rendezvous service property.
`rv_network`	No	`;`	TIBCO Rendezvous network property.

Property Name	Mandatory	Default	Description
`rv_daemon`	No	`tcp:7474`	TIBCO Rendezvous daemon property.
`rv_queue`	No	`Internal-queue`	TIBCO Rendezvous queue for AMI session.
`rv_transport`	No	`Internal-rv-transports`	TIBCO Rendezvous transport for AMI session.

ami_SessionCreate()

Function

Declaration

```
ami_Error ami_SessionCreate(
   ami_Session *     inpAmiSession,
   ami_TraceCode     inTraceLevel,
   const char *      inpRvService,
   const char *      inpRvNetwork,
   const char *      inpRvDaemon,
   unsigned int      inRvTransport,
   unsigned int      inRvQueue,
   const char *      inpName,
   const char *      inpDisplayName,
   const char *      inpHelp,
   ami_TraceHandler inTraceHandler,
   const void *      inpUserData );
```

Purpose Creates a new AMI session. Each session represents a single TIBCO Hawk microagent.

Parameters

Parameter	Description
`inpAmiSession`	Target for returned session handle.
`inTraceLevel`	AMI trace levels for this AMI session. See ami_TraceCode on page 186 for trace level descriptions.
`inpRvService`	TIBCO Rendezvous service parameter.
`inpRvNetwork`	TIBCO Rendezvous network parameter.
`inpRvDaemon`	TIBCO Rendezvous daemon parameter.
`inRvTransport`	TIBCO Rendezvous transport for AMI session.
`inRvQueue`	TIBCO Rendezvous queue for AMI session.
`inpName`	Unique name string for microagent.
`inpDisplayName`	User-friendly name string for microagent.
`inpHelp`	User-friendly microagent description.
`inTraceHandler`	AMI session trace callback function.
`inpUserData`	AMI session user data.

If the transport being used is `tibrv` , you need to perform the following steps:

- Explicitly call `tibrv_open()` method before providing a user queue (`rv_queue`).
- Explicitly call `tibrv_close()` method before application exit.

Refer to TIBCO Rendezvous® documentation for more information.

ami_SessionDestroy()

Function

Declaration

```
ami_Error ami_SessionDestroy(
   ami_Session   inAmiSession );
```

Purpose Destroys the AMI session. The AMI session (and associated handle) is no longer valid. If the AMI session is active it will be stopped prior to being destroyed.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.

ami_SessionAnnounce()

Function

Declaration

```
ami_Error ami_SessionAnnounce(
   ami_Session   inAmiSession );
```

Purpose

Activates the AMI session. All interested Hawk agents are notified that this AMI session is running and available. These agents will add the associated microagent to their microagent lists. This AMI session will be active until `ami_SessionStop` or `ami_SessionDestroy` is called.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.

ami_SessionStop()

Function

Declaration

```
ami_Error ami_SessionStop(
   ami_Session   inAmiSession );
```

Purpose Stops the AMI session. All associated Hawk agents are notified that this AMI session is no longer running or supported. These agents will remove the associated microagent from their microagent lists. This AMI session will be inactive until `ami_SessionAnnounce` is called to re-activate this session.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.

ami_SessionGetName()

Function

Declaration

```
ami_Error ami_SessionGetName(
   ami_Session   inAmiSession
   const char ** inpName );
```

Purpose Gets the name string of AMI session (microagent). This string should not be modified.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.
`inpName`	Target for the returned AMI session name.

ami_SessionGetDisplayName()

Function

Declaration

```
ami_Error ami_SessionGetDisplayName(
   ami_Session    inAmiSession,
   const char **  inpName);
```

Purpose Gets the user-friendly name string of AMI session (microagent). This string should not be modified.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.
`inpName`	Target for the returned display name.

ami_SessionGetHelp()

Function

Declaration

```
ami_Error ami_SessionGetHelp(
   ami_Session   inAmiSession,
   const char ** inpHelp );
```

Purpose

Gets the descriptive text string of AMI session (microagent). This string should not be modified.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.
`inpHelp`	Target for the returned AMI session description.

ami_SessionGetUserData()

Function

Declaration

```
ami_Error ami_SessionGetUserData(
   ami_Session   inAmiSession,
   void **       inpUserData );
```

Purpose Returns the user data of the specified AMI session.

Parameters

Parameter	Description
`inAmiSession`	AMI session parameter.
`inpUserData`	Target for the returned user data.

ami_SessionSendData()

Function

Declaration

```
ami_Error ami_SessionSendData(
   ami_Session            inAmiSession,
   ami_Subscription       inAmiSubscription,
   ami_ParameterListList inReturns );
```

Purpose Returns data for the specified asynchronous method subscription.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiSubscription`	Asynchronous method subscription.
`inReturns`	Data to be returned.

ami_SessionSendError()

Function

Declaration

```
ami_Error ami_SessionSendError(
   ami_Session            inAmiSession,
   ami_Subscription       inAmiSubscription,
   ami_Error              inAmiError );
```

Purpose Reports an error condition for the specified asynchronous method subscription.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiSubscription`	Asynchronous method subscription.
`inAmiError`	Error to be reported.

ami_SessionOnData()

Function

Declaration

```
ami_Error ami_SessionOnData(
   ami_Session  inAmiSession,
   ami_Method   inAmiMethod);
```

Purpose Calls the `ami_OnInvokeCallback` function of the specified AMI asynchronous method once for each currently active subscription. This function is typically invoked when new data becomes available for an asynchronous method. The `ami_OnInvokeCallback` is called with the subscriptions argument values allowing the application to properly send the new data to each subscription.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	AMI Asynchronous method handle.

ami_SessionSendUnsolicitedMsg()

Function

Declaration

```
ami_Error ami_SessionSendUnsolicitedMsg(
   ami_Session    inAmiSession,
   ami_AlertType  inType,
   const char *   inpText,
   int            inId );
```

Purpose Sends an unsolicited message to any interested subscribers.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inType`	Alert type of message.
`inpText`	Textual description of message.
`inId`	User defined ID of message.

AMI C API Method Functions

This chapter describes the AMI C API method functions. A method can return data and or perform a task. AMI methods can accept input parameters and return output parameters as required by the method. Handle-based functions are provided to define AMI methods for a specific AMI session. Through these AMI methods, the AMI manager monitors and manages your AMI application.

Topics

Method Functions Summary

This table summarizes the AMI C API method functions. These functions are described in the following sections.

Function	Description	Page
`ami_MethodCreate()`	Creates a synchronous AMI method and adds it to the specified AMI session.	217
`ami_AsyncMethodCreate()`	Creates an asynchronous AMI method and adds it to the specified AMI session.	218
`ami_MethodGetName()`	Returns the name of the specified method.	219
`ami_MethodGetHelp()`	Returns the textual description of the specified method.	220
`ami_MethodGetUserData()`	Returns the user data of the specified method.	221
`ami_MethodSetIndex()`	Specifies which return parameters to use as the index(es) for methods that return tabular data.	222

ami_MethodCreate()

Function

Declaration

```
ami_Error ami_MethodCreate(
   ami_Session           inAmiSession,
   ami_Method *          inpAmiMethod,
   const char *          inpName,
   ami_MethodType        inType,
   const char *          inpHelp,
   int                   inTimeout,
   ami_OnInvokeCallback  inOnInvoke,
   const void *          inpUserData );
```

Purpose

Allocates and initializes an `ami_Method` object and returns the handle to the object. The `ami_Method` object belongs to the specified `ami_Session` object and will be destroyed when the `ami_Session` is destroyed.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inpAmiMethod`	Location to store new method handle.
`inpName`	Name of the method for AMI purpose.
`inType`	Type of method.
`inpHelp`	Textual description of method.
`inTimeout`	Timeout period in milliseconds.
`inOnInvoke`	Method invocation callback.
`inpUserData`	(Optional) AMI method user data.

ami_AsyncMethodCreate()

Function

Declaration

```
ami_Error ami_AsyncMethodCreate(
   ami_Session           inAmiSession,
   ami_Method *          inpAmiMethod,
   const char *          inpName,
   ami_MethodType        inType,
   const char *          inpHelp,
   int                   inTimeout,
   ami_OnInvokeCallback inOnInvoke,
   ami_OnStartCallback  inOnStart,
   ami_OnStopCallback   inOnStop,
   const void *          inpUserData );
```

Purpose Allocates and initializes an `ami_Method` object and returns the handle to the object. The `ami_Method` object belongs to the specified `ami_Session` object and will be destroyed when the `ami_Session` is destroyed.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inpAmiMethod`	Location to store new method handle.
`inpName`	Name of the method for AMI purpose.
`inType`	Type of method.
`inpHelp`	Textual description of method.
`inTimeout`	Timeout period in milliseconds.
`inOnInvoke`	Method invocation callback.
`inOnStart`	(Optional) Start subscription callback.
`inOnStop`	(Optional) Stop subscription callback.
`inpUserData`	(Optional) AMI method user data.

ami_MethodGetName()

Function

Declaration

```
ami_Error ami_MethodGetName(
   ami_Session       inAmiSession,
   ami_Method        inAmiMethod,
   const char **     inpMethodName );
```

Purpose Returns the name of the specified method in the specified AMI session. This string should not be modified.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpMethodName`	Target for returned method name.

ami_MethodGetHelp()

Function

Declaration

```
ami_Error ami_MethodGetHelp(
  ami_Session      inAmiSession,
  ami_Method       inAmiMethod,
  const char **    inpMethodHelp );
```

Purpose Returns the textual description of the specified method in the specified AMI session. This string should not be modified.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpMethodHelp`	Target for returned method help.

ami_MethodGetUserData()

Function

Declaration

```
ami_Error ami_MethodGetUserData(
   ami_Session     inAmiSession,
   ami_Method      inAmiMethod,
   void **         inpUserData );
```

Purpose Returns the user data of the specified method.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpUserData`	Target for returned user data.

ami_MethodSetIndex()

Function

Declaration

```
ami_Error ami_MethodSetIndex(
   ami_Session      inAmiSession,
   ami_Method       inAmiMethod,
   const char *     inpIndexName );
```

Purpose Specifies which return parameter to use as the primary key for methods that return tabular data. If you need to establish a composite index consisting of multiple parameters, this method can be called repeatedly, once for each index return parameter, in order of precedence with primary key first.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpIndexName`	Return parameter name.

AMI C API Subscription Functions

This section describes the AMI subscription functions.

- Subscription Functions Summary, page 224

Subscription Functions Summary

This table summarizes the AMI C API subscription functions. These functions are described in the following sections.

Function	Description	Page
`ami_SubscriptionSetUserData()`	Allows you to attach application specific data to a particular asynchronous method subscription.	225
`ami_SubscriptionGetUserData()`	Allows you to retrieve the application specific data attached to a particular asynchronous method subscription.	226
`ami_SubscriptionSetCallbackInterval()`	Indicates that for this subscription the associated `onInvoke` callback should be auto-invoked at the specified interval.	227
`ami_SubscriptionGetMethod()`	Allows user to retrieve the associated AMI method object for a particular asynchronous method subscription.	228
`ami_SubscriptionGetArguments()`	Allows user to retrieve the method argument values for a particular asynchronous method subscription.	229

ami_SubscriptionSetUserData()

Function

Declaration

```
ami_Error ami_SubscriptionSetUserData(
   ami_Session        inAmiSession,
   ami_Subscription   inAmiSubscription,
   void *             inpUserData );
```

Purpose Allows you to attach application specific data to a particular asynchronous method subscription. This function is usually used in the `onStart` callback.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiSubscription`	Asynchronous method subscription.
`inpUserData`	User data.

ami_SubscriptionGetUserData()

Function

Declaration

```
ami_Error ami_SubscriptionGetUserData(
   ami_Session        inAmiSession,
   ami_Subscription   inAmiSubscription,
   void **            inpUserData );/
```

Purpose

Allows you to retrieve the application specific data attached to a particular asynchronous method subscription. This function is usually used in the `onInvoke` callback when processing asynchronous method invocations to obtain access to the application specific data associated with that invocation.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiSubscription`	Asynchronous method subscription.
`inpUserData`	Target for returned user data.

ami_SubscriptionSetCallbackInterval()

Function

Declaration

```
ami_Error ami_SubscriptionSetCallbackInterval(
   ami_Session       inAmiSession,
   ami_Subscription  inAmiSubscription,
   int               inInterval );
```

Purpose

Indicates that for this subscription the associated `onInvoke` callback should be auto-invoked at the specified interval. This provides a pseudo-asynchronous event to trigger (what would normally be) synchronous methods so that they can behave as asynchronous methods.

A typical scenario is a method which must calculate (polled) data over a precise time interval and return the calculated result based on that interval. In this case the method returns data not based on a synchronous call but on a specified time interval.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiSubscription`	Asynchronous method subscription.
`inInterval`	Interval in seconds. Zero turns off the interval.

ami_SubscriptionGetMethod()

Function

Declaration

```
ami_Error ami_SubscriptionGetMethod(
   ami_Session         inAmiSession,
   ami_Subscription    inAmiSubscription,
   ami_Method *        inpAmiMethod );
```

Purpose Allows user to retrieve the associated AMI method object for a particular asynchronous method subscription.

ami_SubscriptionGetArguments()

Function

Declaration

```
ami_Error ami_SubscriptionGetArguments(
   ami_Session         inAmiSession,
   ami_Subscription    inAmiSubscription,
   ami_ParameterList * inpArguments );
```

Purpose Allows user to retrieve the method argument values for a particular asynchronous method subscription. This `ami_ParameterList` is only valid while the associated subscription is valid.

AMI C API Parameter Functions

This section describes the parameter functions. The functions are used to define and process method parameters and return values.

- Parameter Functions Summary, page 231

Parameter Functions Summary

This table summarizes the AMI C API parameter functions. These functions are described in the following sections.

Function	Description	Page
`ami_ParameterCreateIn()`	Adds the specified parameter to the list of input parameters for the method.	232
`ami_ParameterCreateOut()`	Adds the specified parameter to the description of return parameters for that method.	233
`ami_ParameterListOut()`	Returns a handle to a list of AMI parameter lists to be used to specify return values for a method invocation.	234
`ami_ParameterSetValue()`	Sets the value of an AMI parameter in the specified AMI parameter list.	235
`ami_ParameterGetValue()`	Retrieves the value of an AMI parameter from the specified AMI parameter list.	236
`ami_ParameterAddChoice()`	Adds a value choice for the specified parameter.	237
`ami_ParameterAddLegal()`	Adds a legal choice for the specified parameter.	238
`ami_ParameterListListDestroy()`	Destroys the specified list of parameter lists.	239

ami_ParameterCreateIn()

Function

Declaration

```
ami_Error ami_ParameterCreateIn(
   ami_Session       inAmiSession,
   ami_Method        inAmiMethod,
   ami_Parameter *   inpAmiParm,
   const char *      inpName,
   ami_DataType      inType,
   const char *      inpHelp );
```

Purpose Adds the specified parameter to the list of input parameters for the method.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpAmiParm`	Target for returned parameter handle.
`inpName`	Name of the parameter for AMI purposes.
`inType`	Type of the parameter.
`inpHelp`	Optional description for the parameter.

ami_ParameterCreateOut()

Function

Declaration

```
ami_Error ami_ParameterCreateOut(
   ami_Session      inAmiSession,
   ami_Method       inAmiMethod,
   ami_Parameter *  inpAmiParm,
   const char *     inpName,
   ami_DataType     inType,
   const char *     inpHelp );
```

Purpose Adds the specified parameter to the description of return parameters for that method.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpAmiParm`	Target for returned parameter handle.
`inpName`	Name of the parameter for AMI purposes. **Note**: The use of curly brackets { } in microagent method parameter names is not supported. Use of these characters results in an error.
`inType`	Type of the parameter.
`inpHelp`	Optional description for the parameter.

ami_ParameterListOut()

Function

Declaration

```
ami_Error ami_ParameterListOut(
   ami_Session               inAmiSession,
   ami_Method                inAmiMethod,
   ami_ParameterListList *   inpAmiParmListList,
   ami_ParameterList *       inpAmiParmList );
```

Purpose

Returns a handle to a list of AMI parameter lists to be used to specify return values for a method invocation. The first call to this method allocates and returns the list of AMI parameter lists (`ami_ParameterListList`) and one parameter list (`ami_ParameterList`) member. The application then uses the `ami_ParameterSetValue` to set the return values into this parameter list (`ami_ParameterList`).

To return more than one row of data (i.e. tabular data) this function can be called repeatedly using the same `ami_ParameterListList` handle. Each call will add an additional return row and return a parameter list (`ami_ParameterList`) to set the return values for that row.

If the `ami_ParameterListList` was created by the AMI API (for example, `ami_OnInvokeCallback`) then the API is responsible for destroying it. It it was created by the user's application, the application must destroy it using `ami_ParameterListListDestroy`.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpAmiParmListList`	List of AMI parameters list.
`inpAmiParmList`	Handle of AMI parameter list.

ami_ParameterSetValue()

Function

Declaration

```
ami_Error ami_ParameterSetValue(
   ami_Session       inAmiSession,
   ami_Method        inAmiMethod,
   ami_ParameterList inAmiParmList,
   const char *      inpName,
   const void *      inpValue );
```

Purpose Sets the value of an AMI parameter in the specified AMI parameter list.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inpAmiParmList`	Set parameter in this parameter list.
`inpName`	Name of parameter being set.
`inpValue`	Value being set.

ami_ParameterGetValue()

Function

Declaration

```
ami_Error ami_ParameterGetValue(
   ami_Session       inAmiSession,
   ami_Method        inAmiMethod,
   ami_ParameterList inAmiParmList,
   const char *      inpName,
   void *            inpValue );
```

Purpose Retrieves the value of an AMI parameter from the specified AMI parameter list.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiMethod`	Handle of AMI method.
`inAmiParmList`	Get parameter from this parameter list.
`inpName`	Name of parameter being retrieved.
`inpValue`	Target for retrieved value.

ami_ParameterAddChoice()

Function

Declaration

```
ami_Error ami_ParameterAddChoice(
   ami_Session      inAmiSession,
   ami_Parameter    inAmiParm,
   const void *     inpData );
```

Purpose Adds a value choice for the specified parameter.

Remarks Value choices can be displayed by the managing application. If value choices are specified for a parameter, other values are also permitted. For a specified object, set only one of either choice or legal values. If both are set, the legal value takes precedence.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiParm`	Handle of AMI parameter.
`inpData`	Choice value.

ami_ParameterAddLegal()

Function

Declaration

```
ami_Error ami_ParameterAddLegal(
   ami_Session       inAmiSession,
   ami_Parameter     inAmiParm,
   const void *      inpData );
```

Purpose Adds a legal choice for the specified parameter.

Remarks Legal value choices can be enforced and displayed by the managing application. If legal value choices are specified for a parameter, no other values are permitted. For a specified object, set only one of either choice or legal values. If both are set, the legal value takes precedence.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiParm`	Handle of AMI parameter.
`inpData`	Legal choice value.

ami_ParameterListListDestroy()

Function

Declaration

```
ami_Error ami_ParameterListListDestroy(
   ami_Session           inAmiSession,
   ami_ParameterListList inAmiParmList );
```

Purpose Destroys the specified list of parameter lists.

Parameters

Parameter	Description
`inAmiSession`	Handle of AMI session.
`inAmiParmList`	Handle of list of parameter lists.

Chapter 7 Security Framework

The TIBCO Hawk product currently supports the ability to “plug-in” an authorization module. The TIBCO Hawk WebConsole uses the plug-in module to create identification objects. The TIBCO Hawk agent guarantees that every request is authorized before execution by invoking the appropriate plug-in method.

The techniques used to address authentication, authorization, integrity, and privacy with regard to messaging and communication channels are varied. It is the goal of the framework to be completely independent of these techniques so that a plug-in is possible regardless of the tools and techniques chosen for implementation.

Implementation of the Certified Class and Trusted Class security models is discussed in *TIBCO Hawk Installation, Configuration, and Administration Guide*.

Topics

TIBCO Hawk Security Concepts

A secure environment addresses concerns of data authentication, authorization, privacy, and integrity.

Authentication

Data authentication is the practice of determining that an entity (such a person or system process) is who it claims to be. This verification can be performed through use of a shared secret system, such as requiring a password, or through certificates and digital signatures.

Authentication involves the following interactions.

1. The entity that is to identify itself to a verifying entity provides an identifier to that verifying entity. The identifier specifies that the originating entity has a particular identity.
2. The verifying entity then receives the identifier from the entity. It uses the verifier to check the authenticity of the entity's claim. In some instances, the verifying entity can be its own verifier.
3. The verifier makes sure that an entity is who it claims it is. This process may or may not involve communication with the entity making the claim.

The verification can involve different levels of authentication.

Identity Only

In identity-only authentication, the system does not verify that the entity is who it claims it is, but does pass the entity's identifier to other parts of the system. This is the lowest level of authentication, and is useful where costs of a more secure authentication system preclude higher degrees of security, but identity is still important. This sort of authentication is useful where non-sensitive data is involved.

Shared Secret

Shared secret authentication is where each entity has a secret, such as a password, that is shared with the authentication system. Proof that the entity holds the secret can take one of the following forms.

- The secret is sent from the entity to the verifier. Web browsers using the basic authentication web paradigm use this method. It is not very secure, as it is possible to impersonate the entity. More security can be added by encrypting the conversation.

- The secret is used to encrypt a commonly-known piece of data. The encrypted data is then sent to the authentication system, which then verifies the identity of the sender by performing its own data encryption and comparing the result with the sender's data.
- A "challenge-response" protocol is used, wherein the verifier provides a piece of randomly-generated data, which the sending entity encrypts using the shared secret. The entity sends back the encrypted data, which the verifier then compares with its own version. If they match, the verifier accepts that the entity is who it claims it is.

Certificates

Digital certificates are a means whereby an entity has a public-private key pair, and registers the public key with a Certificate Authority. The infrastructure required for a public key system is referred to as a Public Key Infrastructure (PKI), of which the third-party Certificate Authority is a part. The Certificate Authority issues a certificate, containing information about the entity and the entity's public key, and signs it.

To provide authentication of identity, the authentication system challenges the entity in a similar manner to the challenge-response protocol. The entity signs the challenge using its private key, and the system verifies this signature by using the entity's public key.

Further information concerning security certificates can be found in *TIBCO Hawk Installation, Configuration, and Administration Guide*.

Authorization

Authorization is generally concerned with operations on which to grant permissions. Sometimes these permissions are determined by work groups or other concerns. Use of tickets, such as `tibrv.tkt`, is an example of authorization. A ticket is used for authentication and authorization in lieu of other credentials. In other cases, the issue is whether a certain operation can be performed on a specified system, or by a specified user.

Data Privacy and Integrity

Data privacy and integrity use encryption techniques to make sure unauthorized entities can't see or modify sensitive data. These techniques are also used when a principal needs to prove it originated a message. Encryption can either use the same key to encrypt and decrypt a message, or use a public-private key pair, where encrypted data using the public key can only be decrypted using the private key, and vice versa.

Data integrity is maintained by using one-way hash functions. These functions generate fixed-length output from input. When sending a message, the sender runs the one-way hash function on the message, encrypts the resulting hash value, and sends the resulting message identification code (MIC) along with the message. The recipient runs the same function on the message, decrypts the MIC, and sees if the results match. A match indicates that the message has not been tampered with.

Considerations for the TIBCO Hawk System

The security provisions of the TIBCO Hawk monitoring system are consistent with its scalable distributed architecture. While a user is not required to trade off scalability for security, the flexibility of the security framework allows choosing a loss of scalability in return for high degrees of security. It also provides a modular mechanism for addressing security, in which the TIBCO Hawk agent can delegate responsibility to the security module, through the interfaces of the security framework. Because every user has unique security needs, security is presented as an open framework. You can develop methods that grant or deny permissions to meet your requirements.

Implementing a Security Policy

The TIBCO Hawk software provides a security framework that you can adapt to your own security needs. To use a security policy, you create a Java class that implements the security interface.

Because every system has unique security needs, the security policy provides an open framework for security implementation, rather than a standardized security policy. You can develop methods to grant or deny permissions based on your needs.

Creating a Java Security Class

The TIBCO Hawk security system must be implemented as a Java class, though you may choose to make this class a simple wrapper that uses the Java Native Interface (JNI) to call other security methods in a C or C++ library. The Java class must implement the `HsConsoleInterface and HsAgentInterface`, which are included with the TIBCO Hawk distribution.

The name of the Java class file for security must be passed to the TIBCO Hawk WebConsole and the TIBCO Hawk agent as a command-line argument.

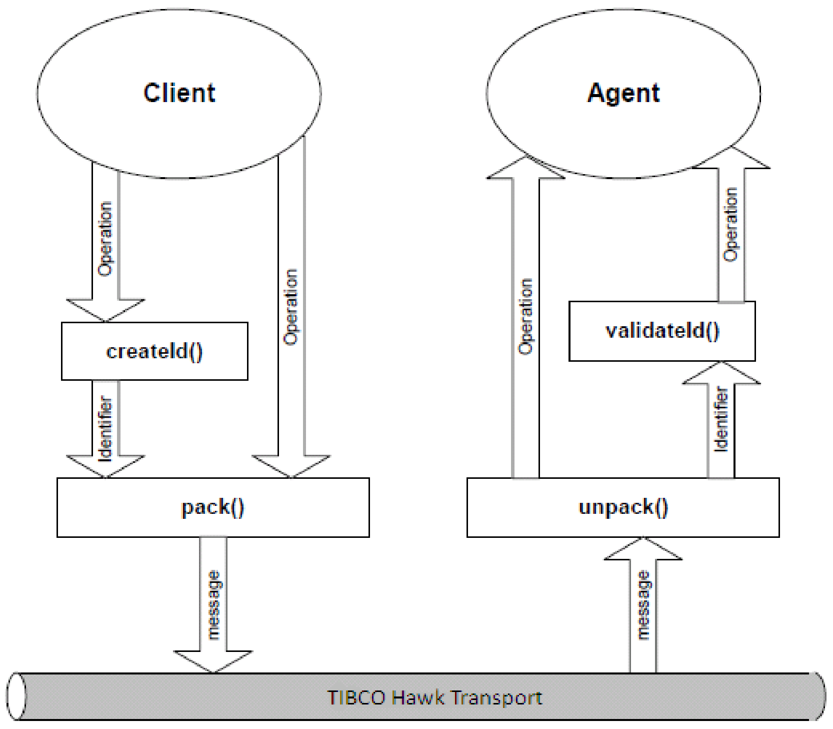
Once both of these processes have been started using this argument, the security policy is in force.

Framework Protocol

The security framework supports an agent and a client-side protocol, as shown below. The client side supports the creation of an identifying object (`createid()` in the diagram) and the transformation of the message (`pack()` in the diagram).

The agent side supports inverse operations for restoring the message's original format (`unpack()`) and validating the identifying object (`validateid()`).

Figure 15 Security Framework Model



Security Objects

While a sample security framework plug-in is provided later in this section, users may prefer to write their own security framework implementation. Plug-ins for the security framework are created using the classes listed here. The prefix `Hs` designates the object as part of the TIBCO Hawk Security Framework.

The following link provides access the detailed descriptions of security classes you can use to create plug-ins:

- Security API Javadocs

Sample Code

The following sample code shows an example of a security policy class file in Java.

```
/*
 * Copyright (c) 1997, 1998 TIBCO Software, Inc. All Rights
Reserved.
 *
 * This software is the confidential and proprietary information of
 * TIBCO Software Inc.
 */
package COM.TIBCO.hawk.security.test;

import java.lang.*;
import java.io.*;

import COM.TIBCO.hawk.console.security.*;

public class Test implements HsConsoleInterface, HsAgentInterface {

   public void Test() {
      System.out.println("PLUGIN: Test.constructor()");
   }

   public void initialize() throws HsException {
      System.out.println("PLUGIN: Test.initialize()");
   }

   public void shutdown() throws HsException {
      System.out.println("PLUGIN: Test.shutdown()");
   }

   public String initialize(int context) throws HsException {
      System.out.println("PLUGIN: Test.initialize(" + context +
")");

      return null;
   }

   public void shutdown(int context) throws HsException {
      System.out.println("PLUGIN: Test.shutdown(" + context +
")");
   }

   public HsIdentifier createId(HsOperation operation)
      throws HsException
   {
      if (operation instanceof HsNodeOperation)
         System.out.println("PLUGIN: createId(" +
                ((HsNodeOperation)operation).microagent() + ":" +
                ((HsNodeOperation)operation).method() + ")");
      else if (operation instanceof HsGroupOperation)
         System.out.println("PLUGIN: createId(" +
                 ((HsGroupOperation)operation).microagent() + ":"
+
                 ((HsGroupOperation)operation).method() + ")");
      else
         System.out.println("PLUGIN: Unknown request");

        HsIdentifier id = null;
        try {
         String name = new String("Test Plug-In");
            id = new HsIdentifier(name.getBytes());
        } catch (HsFrameworkException hsfe) {
            throw new HsException(hsfe.toString());
        }

        return(id);
   }

   public HsPackedOperation pack(HsIdentifier id, HsOperation
operation)
      throws HsException
   {
      if (operation instanceof HsNodeOperation)
         System.out.println(
                          "PLUGIN: pack("+ new
String(id.contents)+ "," +
                 ((HsNodeOperation)operation).microagent() + ":" +
                         ((HsNodeOperation)operation).method() +
")");
      else if (operation instanceof HsGroupOperation)
         System.out.println("PLUGIN: pack(" + new
String(id.contents)+ "," +
                 ((HsGroupOperation)operation).microagent() + ":"
+
                 ((HsGroupOperation)operation).method() + ")");
      else
         System.out.println("PLUGIN: Unknown request");

      TestOperation trustme =
         new TestOperation(id.contents, operation.contents);

      byte[] packed = null;
      try {
         ByteArrayOutputStream buffer = new
ByteArrayOutputStream();
         ObjectOutputStream out = new ObjectOutputStream(buffer);

         out.writeObject(trustme);
         out.flush();
         out.close();

         packed = buffer.toByteArray();

      } catch (IOException ioe) {
      }

        HsPackedOperation packedOperation = null;
        try {
            packedOperation = new HsPackedOperation(packed);
        } catch (HsFrameworkException hsfe) {
            throw new HsException(hsfe.toString());
        }

        return (packedOperation);
    }

    public HsUnpackedOperation unpack(HsPackedOperation operation)
        throws HsException
    {
        System.out.println("PLUGIN: unpack(operation)");

        TestOperation trustme = null;
        try {
            ByteArrayInputStream buffer =
                new ByteArrayInputStream(operation.contents);
            ObjectInputStream in = new ObjectInputStream(buffer);

            trustme = (TestOperation)in.readObject();
            in.close();
        } catch (ClassNotFoundException cnfe) {
            throw new HsException(cnfe.toString());
        } catch (IOException ioe) {
            throw new HsException(ioe.toString());
        }

        HsUnpackedOperation unpacked = null;
        try {
            unpacked = new HsUnpackedOperation
              (new HsIdentifier(trustme.id),new
HsOperation(trustme.operation));
        } catch (HsFrameworkException hsfe) {
            throw new HsException(hsfe.toString());
        }

        return unpacked;
    }

    public boolean validateId(HsIdentifier id, HsOperation
operation)
    {
        if (operation instanceof HsNodeOperation)
            System.out.println("PLUGIN: validateId("+new
String(id.contents)+","+
                 ((HsNodeOperation)operation).microagent() + ":" +
                 ((HsNodeOperation)operation).method() + ")");
        else if (operation instanceof HsGroupOperation)
            System.out.println("PLUGIN: validateId("+new
String(id.contents)+","+
                 ((HsGroupOperation)operation).microagent() + ":"
+
                 ((HsGroupOperation)operation).method() + ")");
        else
            System.out.println("PLUGIN: Unknown request");

        String name = new String(id.contents);
```

```
         if (name.equals("Test Plug-In"))
            return(true);
         else
            return(false);
      }

      public String describe() {
         System.out.println("PLUGIN: Test.describe()");

         return(new String("TIBCO Hawk Test security model."));
      }
   }
```

Index

Appendix A Common Configuration Object API Methods

This chapter describes the common RulebaseEngine and TIBCO Repository microagent methods that are used with the Configuration Object API.

Topics

Microagent and Method Invocation used in ConsequenceAction

A ConsequenceAction in a rulebase executes an action when the condition of the test is satisfied. The action taken is in the form of method invocation. The following are common actions performed by a ConsequenceAction of a rulebase:

- send a notification or an alert
- execute a custom command or script
- send an email

Since a ConsequenceAction invokes a method invocation on a specific microagent, any known method invocation with `ACTION` or `INFO` type can be specified. However, caution must be taken on methods that may potentially take a long time to execute.

When constructing a ConsequenceAction object, two arguments are needed: the name of a microagent and the method invocation that will be performed on that microagent. Hence, before constructing a ConsequenceAction, first construct a MethodInvocation. A MethodInvocation requires a method name. Depending on the method, it may also take arguments. For a complete list of built-in microagents and their open methods, please refer to the *TIBCO Hawk Microagent Reference*.

The follow methods show how the common ConsequenceActions can be created. Note that the `sendAlertMessage` method is a proprietary method of the RulebaseEngine microagent and is not listed in the *TIBCO Hawk Microagent Reference*.

```
/**
* Create an alert action.
*/
ConsequenceAction createAlertAction(String state, String alert)
throws RBEConfigObjectException
{
DataElement[] args = new DataElement[1];
if  ( state.equals("High") )
     args[0] = new DataElement("message",
                 new
COM.TIBCO.hawk.config.rbengine.rulebase.util.AlertHigh(alert));
else if ( state.equals("Medium") )
    args[0] = new DataElement("message",
         new
COM.TIBCO.hawk.config.rbengine.rulebase.util.AlertMedium(alert));
else if ( state.equals("Low") )
    args[0] = new DataElement("message",
         new
COM.TIBCO.hawk.config.rbengine.rulebase.util.AlertLow(alert));
MethodInvocation mi = new MethodInvocation( "sendAlertMessage",
args);
return new
ConsequenceAction("COM.TIBCO.hawk.microagent.RuleBaseEngine", mi);
}


/**
* Create an email action
*/
ConsequenceAction createEmailAction(String to, String from, String
cc, String subject, String server, String content) throws
RBEConfigObjectException
{
DataElement[] args = new DataElement[6];
args[0] = new DataElement("To", to);
args[1] = new DataElement("From", from);
args[2] = new DataElement("CC",cc);
args[3] = new DataElement("Subject", subject);
args[4] = new DataElement("Mail Server", server);
args[5] = new DataElement("Content", content);
MethodInvocation mi = new MethodInvocation( "sendMail", args);
return new
ConsequenceAction("COM.TIBCO.hawk.microagent.RuleBaseEngine", mi);
}


/**
 * Create a custom execute action.
 */
ConsequenceAction createCustomAction(String cmdStr)
throws RBEConfigObjectException
{
DataElement[] args = new DataElement[1];
args[0] = new DataElement("command", cmdStr);
MethodInvocation mi = new MethodInvocation( "execute", args);
return new ConsequenceAction("COM.TIBCO.hawk.microagent.Custom",
mi);

}
```

Interaction with Agent and Repository using the Console API

An application that uses the Configuration Object API often needs to retrieve, update, or replace such configuration objects. To retrieve and send the configuration objects from the application to the agent or repository, the application needs to use the Console API.

When an agent is running in the repository configuration mode, configuration objects (such as a rulebase or schedule) updated on the agent are not permanent and the rulebase map can only be updated on a repository.

The following steps illustrate one possible way that an application using the Console API and Configuration Object API can update a configuration object of an agent or repository.

1. Create a TIBHawkConsole.
2. Listen to AgentMonitorEvent of the AgentMonitor.
3. Wait until the AgentMonitorEvent from the desire agent is received by examining the AgentInstance of the AgentMonitorEvent.
4. Retrieve the MicroAgentID of the RulebaseEngine or Repository from the AgentInstance.
5. Retrieve the configuration object from the RulebaseEngine or Repository microagent using the AgentManager.
6. Update or modified the configuration object.
7. Update the configuration object on the RulebaseEngine or Repository microagent.

For convenience, the rest of this section lists the methods in RulebaseEngine and Repository microagent that are useful for retrieving and updating Rulebase, Schedule, and Rulebase Map. The complete list of methods can be found in the *TIBCO Hawk Methods Reference*.

For details and code samples, refer to the Java sample descriptions in Appendix B, Sample Programs, and the sample Java files described there.

For details on TIBHawkConsole, AgentMonitor, AgentMonitorEvent, AgentInstance, and other Console API classes, please refer to Chapter 2, Console API.

Methods Reference

The following `COM.TIBCO.hawk.microagent.RuleBaseEngine` methods are commonly use in ConsequenceActions:

- RuleBaseEngine:sendAlertMessage, page 260
- RuleBaseEngine:execute, page 261
- RuleBaseEngine:sendMail, page 262

The following `COM.TIBCO.hawk.microagent.RuleBaseEngine` methods are used for updating configuration objects in the agent:

- RuleBaseEngine:addRuleBase, page 263
- RuleBaseEngine:deleteRuleBase, page 264
- RuleBaseEngine:setSchedules, page 265
- RuleBaseEngine:getSchedules, page 266
- RuleBaseEngine:getRBMap, page 267

The following `COM.TIBCO.hawk.microagent.Repository` methods are used for updating configuration objects in the repository:

- Repository:addRuleBase, page 268
- Repository:deleteRuleBase, page 269
- Repository:setSchedules, page 270
- Repository:getSchedules, page 271
- Repository:setRBMap, page 272
- Repository:getRBMap, page 273

RuleBaseEngine:sendAlertMessage

Method

Purpose This method sends an alert.

Type Proprietary, Synchronous, `IMPACT_INFO`

Remarks This method can be invoked only from a rulebase action, because an alert is associated with the rulebase that triggers the alert.

Arguments

Name	Type	Description
command	COM.TIBCO.hawk.config.rbengine.rulebase.util.AlertHigh COM.TIBCO.hawk.config.rbengine.rulebase.util.AlertMedium COM.TIBCO.hawk.config.rbengine.rulebase.util.AlertLow COM.TIBCO.hawk.config.rbengine.rulebase.util.Notification	The alert message to be sent.

Returns None

RuleBaseEngine:execute

Method

Purpose This method executes a command and ignores the result.

Type Open, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
command	String	The command to execute. External and Internal variables can be used.

Returns None

RuleBaseEngine:sendMail

Method

Purpose This method (on all platforms) sends an email notification.

Remarks The `To` and `Subject` fields are only mandatory fields and all other fields are optional. If `From` is not specified, the current host ID is used. If the `Content` field is blank, the text of the `Subject` field is used. If the Mail Server is not specified, then SMTP server configured in the agent sends email.

Rulebases can send mail upon detecting a specified condition.

Type Open, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
To	String	Address of the receiver
CC	String	CC (carbon copy) recipients of email
BCC	String	BCC (blind carbon copy) recipients of email
Subject	String	Subject of email
Content	String	Content of email
Mail Server	String	SMTP mail server used to send message
From	String	Address of the sender

Returns None

RuleBaseEngine:addRuleBase

Method

Purpose This method adds a rulebase to the agent.

Remarks Timeout (milliseconds): 10000

Type Proprietary, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
RulebaseXML	COM.TIBCO.hawk.config.rbengine.rulebase.RulebaseXML	An Object that contains XML formatted string that represent the rulebase.

Returns None

RuleBaseEngine:deleteRuleBase

Method

Purpose This method deletes a rulebase from the agent.

Type Open, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
RuleBaseName	String	The name of the rulebase to be deleted.

Returns None

RuleBaseEngine:setSchedules

Method

Purpose This method replaces the schedules in the agent.

Type Proprietary, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
SchedulesXML	COM.TIBCO.hawk.config.rbengine.schedule.SchedulesXML	An Object that contains XML formatted string that represent the schedule.

Returns None

RuleBaseEngine:getSchedules

Method

Purpose This method returns the currently loaded Schedules.

Type Proprietary, Synchronous, `IMPACT_INFO`

Arguments None

Returns

Name	Type	Description
SchedulesXML	COM.TIBCO.hawk.config.rbengine.schedule.SchedulesXML	An Object that contains XML formatted string that represent the schedules.

RuleBaseEngine:getRBMap

Method

Purpose This method returns the currently loaded RBMap.

Remarks Timeout (milliseconds): 10000

Type Proprietary, Synchronous, `IMPACT_INFO`

Arguments None

Returns

Name	Type	Description
RBMapXML	COM.TIBCO.hawk.config.rbengine.rbmap.RBMapXML	An Object that contains XML formatted string that represent the rulebase map.

Repository:addRuleBase

Method

Purpose This method adds a rulebase to the repository.

Remarks Timeout (milliseconds): 10000

Type Proprietary, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
RulebaseXML	COM.TIBCO.hawk.config.rbengine.rulebase.RulebaseXML	An Object that contains XML formatted string that represent the rulebase.

Returns None

Repository:deleteRuleBase

Method

Purpose This method deletes a rulebase from the repository.

Type Open, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
RuleBaseName	String	The name of the rulebase to be deleted.

Returns None

Repository:setSchedules

Method

Purpose This method set the schedules in the repository.

Type Proprietary, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
SchedulesXML	COM.TIBCO.hawk.config.rbengine.schedule.SchedulesXML	An Object that contains XML formatted string that represent the schedule.

Returns None

Repository:getSchedules

Method

Purpose This method returns the schedules in the repository.

Type Proprietary, Synchronous, `IMPACT_INFO`

Arguments None

Returns

Name	Type	Description
SchedulesXML	COM.TIBCO.hawk.config.rbengine.schedule.SchedulesXML	An Object that contains XML formatted string that represent the schedule.

Repository:setRBMap

Method

Purpose This method set the rulebase map in the repository.

Type Proprietary, Synchronous, `IMPACT_ACTION`

Arguments

Name	Type	Description
RBMapXML	COM.TIBCO.hawk.config.rbengine.rbmap.RBMapXML	An Object that contains XML formatted string that represent the rulebase map.

Returns None

Repository:getRBMap

Method

Purpose This method returns the rulebase map in the repository.

Type Proprietary, Synchronous, `IMPACT_INFO`

Arguments None

Returns

Name	Type	Description
RBMapXML	COM.TIBCO.hawk.config.rbengine.rbmap.RBMapXML	An Object that contains XML formatted string that represent the rulebase map.

Appendix B Sample Programs

This appendix describes the sample programs provided in the `/examples/rulebase_api`, `/examples/schedule_api`, and `/examples/rbmap_api` directories.

Topics

Rulebase Samples

The example Java source files in `examples/rulebase_api` show how to use the Rulebase-related classes of the Configuration Object API.

For details of the code, refer to the Java source files.

RBIsample1.java

This sample shows how to create a simple rulebase and save it to a file. The rulebase uses the Spot microagent that is created using the provided sample application.

In the rulebase, the data source is the current color of the Spot microagent. The test in the rulebase checks the color of the Spot microagent. If the color is blue, it performs an action that changes the color to green.

RBIsample2.java

This sample extends `RBIsample1.java` to show how to use a compound test in a rulebase.

In this example, the rulebase created in `RBIsample1.java` is save to a file. After reading the rulebase from the file, `RBIsample2.java` replaces the test with a compound test. The compound test checks if the current color is either blue or red. If this condition is satisfied, it performs an action that changes the color to green.

RBIsample3.java

This sample demonstrates how an application can create, add, update, and delete a rulebase dynamically on an agent using the Configuration Object API and Console API. Refer to Appendix A, Common Configuration Object API Methods, for methods related to communication between a Console application and the TIBCO Hawk agent.

When running this sample, the Spot microagent should also be running. This allows you to see the effect of the action performed by the rulebase after being updated on the TIBCO Hawk agent.

The sample performs the following steps:

1. Creates a rulebase using the color of the SPOT microagent as the data source, which changes to green if the current color is blue.

2. Adds the created rulebase to the agent.
3. Changes the color of SPOT to blue. (At runtime, a few second after this call, the color on the Spot microagent will change to green due to the test in the rulebase.)
4. Retrieves the rulebase, modifies it to change the color of SPOT to green if the current color is either blue or red, and updates the rulebase on the agent.
5. Sets the color of SPOT on the agent to red. (At runtime, a few second later after this call, the color on the Spot microagent will change to green due to the test in the rulebase.)
6. Sets the color of SPOT on the agent to blue. (At runtime, a few second later after this call, the color on the Spot microagent should change to green due to the test in the rulebase.)
7. Deletes the rulebase from the agent.

Schedule Samples

The example Java source files in `examples/schedule_api` show how to use the Schedule-related classes of the Configuration Object API.

For details of the code, refer to the Java source files.

ScheduleCreateAndSave.java

This sample shows how to create a simple schedule and save it to a file.

The schedule created contains a period which is in-schedule from 8:00am to 5:59pm every Monday.

ScheduleUsingExclusion.java

This sample creates a schedule named BusinessHourInSummer. The purpose is to show the use of both inclusion and exclusion periods.

This schedule is in-schedule from Monday to Friday, 8:00 AM to 5:59 PM. The hours between 12:00 PM and 2:00 PM in June, July and August are excluded.

This schedule definition uses both inclusion and exclusion period even though the same schedule could be created without using the exclusion period but using a more specific inclusion period.

ScheduleWithPeriodGroup.java

This sample extends `ScheduleUsingExclusion` to demonstrate the use of PeriodGroup and PeriodGroupReference.

The exclusion period in `ScheduleWithPeriodGroup.java` is replaced by a period group that specifies the same time period. The resulting schedule contains an exclusion period equivalent to the one in `ScheduleUsingExclusion`.

ScheduleGetAndSet.java

This sample demonstrates how an application can get and set schedules dynamically on a repository using the Configuration Object API and Console API.

The sample performs the following steps:

1. Gets the schedules from the repository.

2. Creates and adds a schedule to the existing schedules.
3. Replaces the schedules in the repository.

Rulebase Map Samples

The example Java source files in `examples/rbmap_api` show how to use the Rulebase Map-related classes of the Configuration Object API.

For details of the code, refer to the Java source files.

RBMapCreateAndSave.java

This sample shows how to create a simple Rulebase Map and save it to a file. It also shows the use of the method `getAgentRulebases`() to retrieve rulebases maps to an agent in the Rulebase Map.

Based on the rulebase map, `getAgentRulebases()` returns a list of rulebases that an agent should load during startup if the agent is running in the repository configuration mode. It does not return a rulebase map.

RBMapUseCommand.java

This sample extends `RBMapCreateAndSave` to include an external command that generates a list of rulebases for an agent.

This example uses `RBMapUseCommand.exe` and will run on Microsoft Windows only.

RBMapUseCommand.java

This sample demonstrates how an application can get and set a Rulebase Map dynamically on a repository using the Configuration Object API and Console API.

The sample performs the following steps:

1. Gets the Rulebase Map from the repository.
2. Updates the Rulebase Map.
3. Replaces the Rulebase Map in the repository.

Appendix C Planning Your Instrumented Application

This appendix describes steps of planning your AMI interface.

Topics

Planning an AMI Interface

These steps can help you in planning your AMI interface:

1. Examine the samples of AMI code included with the TIBCO Hawk distribution to see how these requirements are carried out in code.
2. Copy one of the AMI code samples and amend it to add one or two methods.
3. When you are ready to create an interface to your application, consider what data and methods is to be accessed or changed through the application's management interface:
   - List the separate data items you want to be able to retrieve.
   - List the data items you want to be able to change.
   - List the actions you want to be able to carry out.

   Each of these items will become a supported method of your AMI interface.
4. For each of these methods, decide what arguments it will use and what results it will return. If a method uses arguments, consider what course to take if a default argument is supplied.
5. Using the collected information for each method, create an outline detailing the message structure to be returned to describe these methods to a manager.
6. Use this outline to write a describer method, which returns a nested message.
7. Set up the code to initialize the TIBCO Hawk AMI session.
8. Write methods that respond to method invocation messages from the manager. In creating each message, use an outline as you did with the describe method message, to lay out what information the message will include.

An Example of Planning AMI Methods

Let us suppose you have a transaction-processing application that needs to be monitored so that its message queue length doesn't grow too large. Instances of this application might make up a fault-tolerant group with primary/secondary status, which you want to autonomously monitor but also to interactively control.

Defining these needs, you could list two items:

- The manager should retrieve the application's queue length on a periodic basis.
- The manager should retrieve the application's primary/secondary status on a periodic basis.

You now create these methods:

- `getQueueLength`, which takes no arguments and returns the queue length.
- `getFTStatus`, which takes no arguments and returns the fault-tolerant status (primary or secondary) of the application.
- `makePrimary`, which takes no arguments and sets the fault-tolerant status of the application instance to primary.
- `makeSecondary`, which takes no arguments and sets the fault-tolerant status of the application instance to secondary.

Since you will be using a TIBCO Hawk agent as the manager, you build a rulebase with two rules, as follows:

- The first rule has as its data source the `getQueueLength` method. It raises an alert if the queue length is greater than 200.
- The second rule has as its data source the `getFTStatus` method. It sends a notification each time there is a fault-tolerant state transition.

From the TIBCO Hawk WebConsole, operators can control the fault-tolerant state of any instance of the application by invoking the `makePrimary` or `makeSecondary` methods.

Other possible AMI examples might include:

- Methods to dynamically control trace or debug levels.
- Methods to drop or re-request information from data streams.
- Methods to monitor client connections for server applications.
- Methods to control application backup procedures.
- Methods to extract internal state information used in debugging.

- Methods to change various application configuration parameters.
- Methods that instruct applications to write their new configuration parameters to configuration files or to the Microsoft Windows registry.