TTM
Total Transaction Management

ActiveMQ Monitor

A Monitoring System for ActiveMQ

User Guide

Total Transaction Management, LLC
An Open Source Solutions Company
570 Rancheros Drive, Suite 140
San Marcos, CA 92069
760-591-0273
www.ttmsolutions.com

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

Table of Contents

1 INTRODUCTION 1

2 ACTIVEMQ'S DEFAULT ADVISORY SYSTEM 1

3 THE AMON MONITORING SYSTEM 2

3.1 MONLET FRAMEWORK 2

3.1.1 The AMon Plugin Module 3

3.1.2 The Monitoring Expression Language (MEL) 5

3.1.3 Apache Camel 7

3.2 SNMP AGENT AND MIB 8

4 THE ANATOMY OF A MONLET 8

5 THE MEL 23

5.1 EVENT TYPES AS CONSTANTS 24

5.2 VARIABLES 25

5.3 METHODS 34

5.4 ASYNCHRONOUS PROCESSING 35

6 THE INTERVAL TIMER 35

7 SNMP CAMEL ENDPOINT 36

7.1 SNMP LOGGINGNOTIFICATION 36

7.2 SNMP MONITORNOTIFICATION 40

8 INSTALLING AMON 42

8.1 INSTALLING THE AMON FILES 42

8.2 LOADING AMON INTO THE ACTIVEMQ MESSAGE BROKER 43

8.3 LOADING YOUR MONLETS INTO THE ACTIVEMQ MESSAGE BROKER 45

9 CONFIGURING AMON 47

9.1 PROPERTIES FILE 47

9.2 JMX CLIENT 51

9.3 LOG4J MONITORING 52

9.4 SNMP AGENT 53

9.4.1 SNMP Notifications 55

10 THE MANAGEMENT CONSOLE 56

11 VIRTUALMACHINE OBJECT 56

1 Introduction

It is important that enterprise-class infrastructure software systems, like the ActiveMQ message broker, provide system administrators with a mechanism through which they can monitor the overall health and performance of that system. Without a monitoring system in-place, the reliability, high-availability, and performance of the system, as a whole, are compromised. This guide describes the feature-functionality of such a monitoring system for ActiveMQ. The monitoring system is referred to as the "ActiveMQ Monitor" or "AMon" for short.

2 ActiveMQ's Default Advisory System

The ActiveMQ message broker does provide an out-of-the-box monitoring system that comprises a series of "Advisory" topics and the advisory messages that the message broker publishes to those topics. However, this advisory system is static in nature, because the system's set of advisory topics and event messages cannot be customized and/or extended by the end-user without having to make changes to ActiveMQ's core components.

Advisory messages can be thought of as events that inform your management clients of certain conditions that have already occurred within the ActiveMQ message broker. ActiveMQ management clients can thus monitor the message broker by subscribing to the Advisory topics and processing the messages that are read from those topics. However, the advisory system does not allow you to monitor the ActiveMQ message broker such that you can detect if a particular runtime state is beginning to reach a critical point. For example, the advisory system does not allow you to set certain thresholds against memory, threads, message queues, etc.

Starting with version 5.2 of ActiveMQ, this advisory system covers some 26 different events. More information on this system can be found here. Here is a sampling of the types of events that are generated by the advisory system.

- Consumers, producers, and connections starting and stopping
- Temporary destinations being created and destroyed
- Messages expiring on topics and queues
- Brokers sending messages to destinations with no consumers.
- Connections starting and stopping
- No consumer is available to process messages being sent on a Topic or Queue
- Slow queue or topic consumer
- Fast queue or topic producer
- Discarded message (e.g., expired message) from queue or topic
- Message consumed from queue or topic
- Message delivered to broker
- A usage resource is full
- A slave broker has taken over for a master.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

3 The AMon Monitoring System

AMon is a monitoring system that is specifically designed for the ActiveMQ message broker. The system comprises two primary modules: Monlet Framework and SNMP Agent.

3.1 Monlet Framework

AMon does not perform the actual monitoring of an ActiveMQ message broker. What AMon provides is a framework that facilitates the rapid development and deployment of one or more user-defined ActiveMQ monitoring agents, or “Monlets” as they are referred to in this document. The AMon framework and the Monlets that it hosts are embedded within an ActiveMQ message broker process. The framework and its Monlets combine to provide a powerful and flexible monitoring system for ActiveMQ.

A Monlet is an end-user defined component that comprises one or more conditional expressions and their corresponding actions; the Monlet invokes the actions if and when the conditional expressions are TRUE. You can also view the conditional expressions as predicate functions that accept one or more variables with the variables representing the runtime state of the message broker.

In general, the conditions are designed to test the runtime or operational state of those objects that comprise the ActiveMQ message broker. Unlike ActiveMQ’s default advisory system, the AMon framework gives its Monlets direct access to all of the message broker’s runtime object hierarchies. The broker comprises many objects and their properties; therefore, there are also many conditions that can be defined and monitored. Here are just a couple of examples of the types of Monlets and the objects that they can monitor.

1. Resource Monlet – one that tests whether a particular object, which represents a resource (e.g., memory, queue, etc.), has breached a threshold and invokes an action if the threshold is breached. The action can be invoking a script, firing off an email, or raising an SNMP notification[1].
2. Audit Monlet – one that tests whether a message that is being routed through the broker meets some sort of condition (e.g., includes some user-defined property that has been assigned a particular value), and if the condition is met the contents of the message can be written to an audit log.

AMon also allows Monlets to test conditions associated with the broker’s hosting Java Virtual Machine (JVM). For example, a Monlet can test whether the number of active threads within the JVM or its memory heap size has breached a certain threshold.

So unlike ActiveMQ’s default Advisory system, AMon is a framework that:

[1] AMon provides the facilities that allow Monlets to generate SNMP notifications (a.k.a., traps)

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

- Allows end-users to define their own monitoring agents (Monlets)
- Gives the monitoring agents the ability to monitor all aspects of the ActiveMQ message broker's runtime state
- Gives them the capability to perform a myriad of different actions in response to one or more conditional expressions returning TRUE.

AMon maintains a clean separation between the Monlets and the broker objects that they are monitoring. This separation is provided by a combination of the following:

- Event messages that are published by the AMon plugin module, which is the AMon core component or engine.
- AMon's default monitoring expression language (MEL)
- The Apache Camel integration framework's routing Domain Specific Language (DSL).

The above items all combine to make developing Monlets relatively straightforward and allows the AMon system as a whole to be extensible and flexible. The following three sections describe the above three items in more detail.

Along with the Java programming language, the Monlet developer will need to have a basic understanding of Apache Camel and the Unified Expression Language (UEL).

The AMon product package includes example "quick start" Monlets, which are designed to serve as examples and help developers ramp-up quickly on Monlet development.

3.1.1 The AMon Plugin Module

AMon's core component or engine is installed as an ActiveMQ message broker plugin; therefore, the AMon plugin is included in the message broker's main processing event chain. Inclusion in the processing event chain allows the AMon plugin to participate in the processing of events that occur within the ActiveMQ message broker. The AMon plugin's responsibility is not to process these events, but to publish event messages that correspond to each of these processing events. Other portions of the AMon framework deliver these event messages to the end-users' Monlets (much more on this later).

The event messages that are published by the AMon plugin are similar to the ActiveMQ advisory system's event messages. That is, both message types are published in response to a processing event (e.g., connection created, connection removed, message arrived, message expired, etc.) that has occurred within the message broker. All together, there are 33 AMon broker processing events. Through an external properties file, the AMon plugin is configured to publish messages that correspond to all, none, or any subset of these events. Refer to Event Types as Constants for a complete listing of all the AMon event types.

The AMon core also introduces two events that are not associated with broker processing events. The first event is generated whenever the ActiveMQ message broker writes a message to its log file (ACTIVEMQ_HOME/data/activemq.log) with a severity or priority of WARN or higher.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

The second event is generated via a configurable interval timer with a one millisecond resolution. These two events and the timer are further described in later sections of this guide.

One important distinction between the AMon event message and ActiveMQ's advisory event message is that an AMon event message is an object whose reference is communicated to a Monlet via direct method invocation (DMI), whereas an advisory event message is designed to be transmitted as a JMS message across the network to remote ActiveMQ advisory clients. This difference gives the AMon event message a distinct advantage, because it can and does provide direct references to the message broker's runtime object hierarchy. Having direct references to the broker's object hierarchy provides an invaluable source of information from which many different conditions and thus types of Monlets can be defined.

So, an AMon event message should be viewed as a portal or vehicle through which Monlets test the operational state of the broker at the time a processing event occurs.

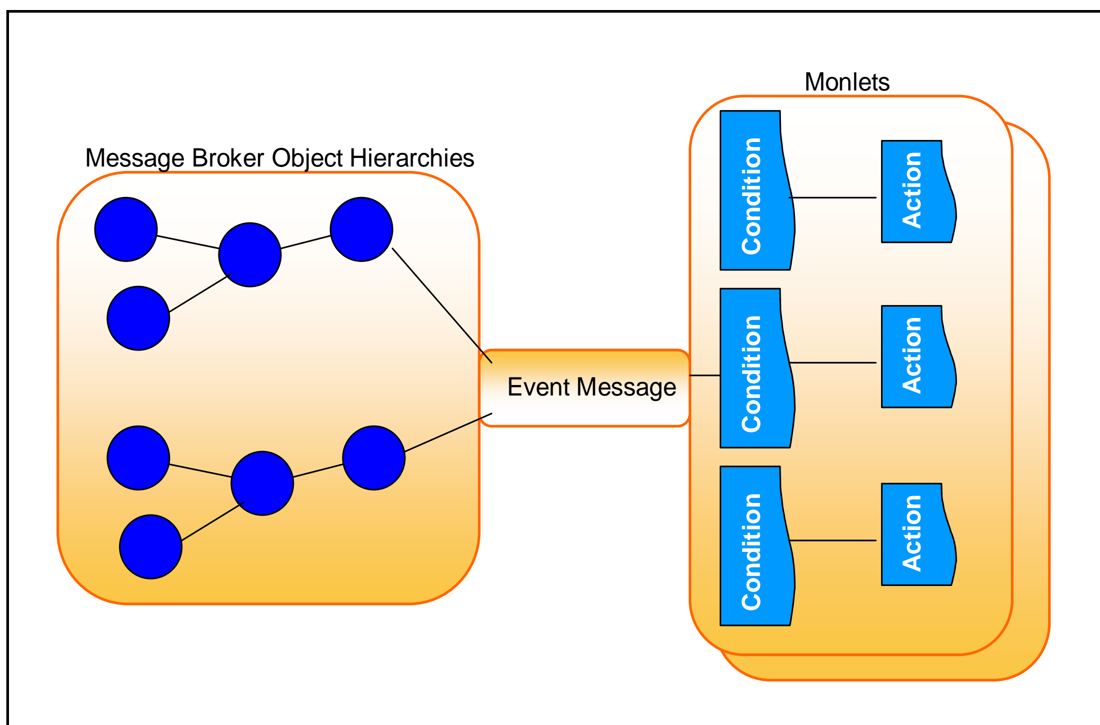


Figure 1 AMon Monlets and Event Message

A Monlet is not precluded from sending JMS messages in response to one or more of its conditions returning TRUE; therefore, a Monlet can also act as a source of advisory messages destined to remote advisory or management clients. In other words, one action associated with a condition can be to send a JMS message to a remote management client. This is one way in which AMon complements ActiveMQ's advisory system.

Copyright 2009, Total Transaction Management

3.1.2 The Monitoring Expression Language (MEL)

The MEL is an extension of the Unified Expression Language (UEL), which is the expression language used within Java Server Pages (JSPs). The MEL, which was developed for the AMon project, is used by AMon end users to define their Monlets' conditional statements or predicates.

AMon depends on a Java implementation of the UEL that is called the Java Unified Expression Language or JUEL for short. JUEL provides a lightweight and efficient Java implementation of the UEL. The JUEL leverages the JavaBeans architecture/standard to access objects and their properties. Most if not all of ActiveMQ's runtime objects adhere to the JavaBeans standard, which makes them conducive to accessing from the JUEL.

The JUEL extension that defines MEL is specifically designed to facilitate access to the message broker's objects. The MEL includes variables that are references to message broker objects, properties that can be used to dynamically configure Monlets, and utility methods.

Recall that a Monlet subscribes to AMon event messages. By the time an event message has been delivered to a Monlet, the AMon framework will have created a MEL context for that event message. This MEL context includes the binding of MEL variables to the broker objects that are referenced via the event message. The context also includes MEL-specific methods/functions and properties.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

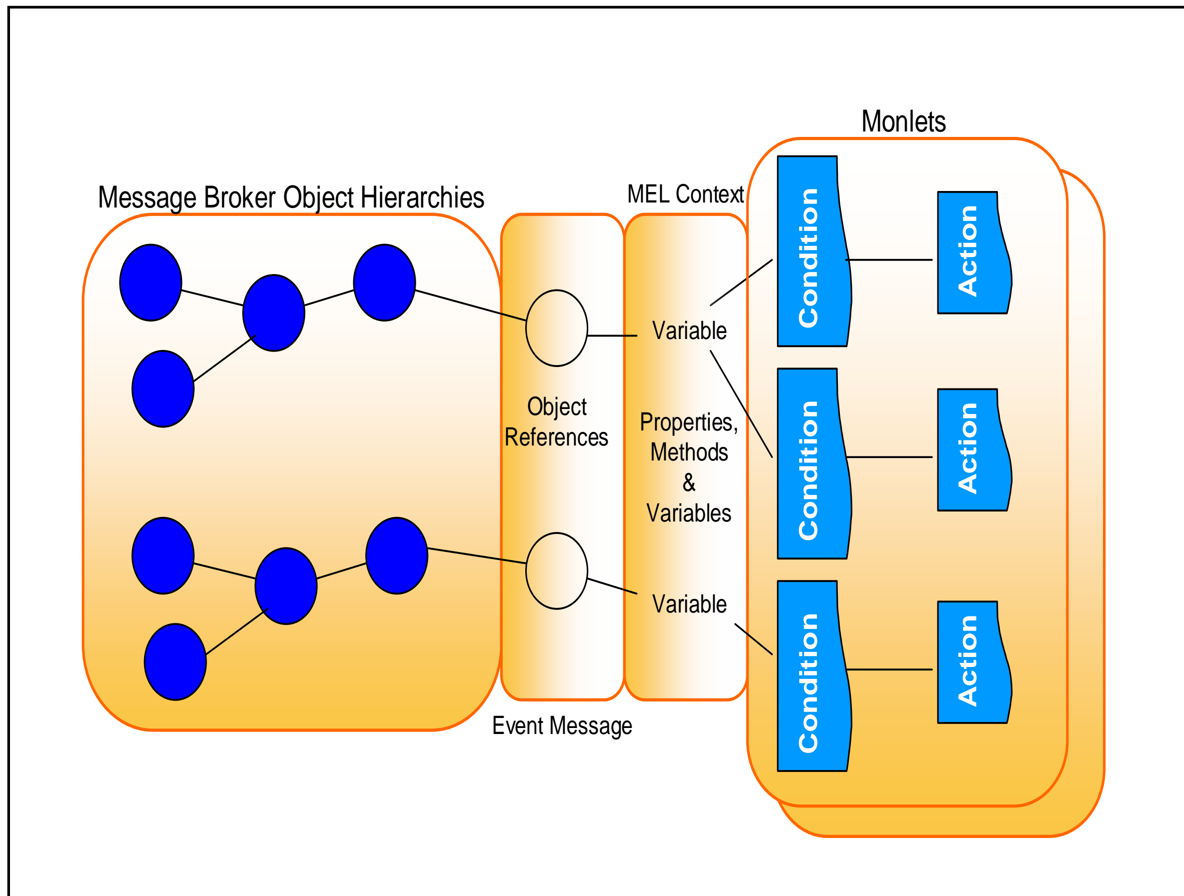


Figure 2 The MEL Context and AMQ Object Hierarchies

This is a very simple example of a MEL conditional expression (predicate) that is found within a Monlet.

${destination.name = = 'TEST.Q'}

The above expression is a condition that tests whether an ActiveMQ destination object has been assigned the name of 'TEST.Q'. Underneath the covers, AMon has assigned the 'destination' variable a reference to an ActiveMQ Destination object. That object's getName() method is then invoked to return the name of the Destination object. Any of the Destination object's 'get' and 'is' methods can be invoked in this fashion. Some of the Destination object's 'get' methods can return other objects whose 'get' and 'is' methods also can be accessed. Thus the 'destination' variable gives your Monlet access to various object hierarchies within the ActiveMQ message broker.

As stated in the UEL web page, the UEL supports what is referred to as *immediate* and *deferred* evaluation of expressions. The MEL only supports immediate evaluation; therefore, only the "${}" syntax, and not the "#{}" syntax, can be used to define MEL conditional expressions.

Copyright 2009, Total Transaction Management

Here's another MEL example that illustrates how you can traverse an object hierarchy tree.

```
${destinationStatistics.messages.count >= 100}
```

In the above MEL conditional expression, the destinationStatistics.getMessages().getCount() method invocation chain is invoked, which returns the number of messages that are currently residing in the Destination. In the example, the condition tests positive if the number of messages is equal to or exceeds 100.

A MEL conditional expression is tested whenever the Monlet receives an AMon-published event message (more on this later).

So in some respects, a Monlet is somewhat analogous to a JEE servlet. A servlet adheres to a well-defined API through which it receives HTTP request packets; the packets are dispatched to the servlet by the JEE web container. In similar fashion, a Monlet adheres to a well-defined interface, which in this case is in the form of a combination of Java domain specific language (DSL) and expression language. And it is through this interface that a Monlet receives and processes AMon event packets.

3.1.3 Apache Camel

Apache Camel is a powerful, Spring-based integration framework that is primarily used to implement the enterprise integration patterns defined in Gregor Hohpe and Bobby Woolf's book titled, "Enterprise Integration Patterns". One of the unique features of Camel is that it offers a 'routing' Java Domain Specific Language (DSL) that is used by Camel end-users to:

- Quickly implement all sorts of messaging patterns (filters, content-based routers, splitters, recipient lists, message enrichers, message transformers, etc) or "Routes" as they are also referred to.
- Interface to a myriad of different endpoints/components that Camel supports. These components can also serve as potential Monlet 'actions'.

The other nice thing about Camel's Java DSL is that it supports many different scripting and "*expression*" languages that are used for implementing filters and content based routers. Camel also allows you to very easily introduce your own language, which is what the AMon project has done with MEL.

A Monlet is a Camel Route that utilizes a slightly extended version of Camel's routing Java DSL. A Monlet is a subclass of the com.ttm.activemq.monitor.Monlet abstract super class, which in turn is a subclass of the Camel Route super class. The reason a Monlet directly extends the Monlet class and not Camel's Route class will become apparent as you read through the guide.

Camel's Java DSL is combined with the MEL to facilitate the quick development and deployment of Monlets. Monlet developers will have to learn the basics of Camel's routing Java

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

DSL, as well as the JUEL and MEL. These are not difficult subjects. You are encouraged to visit the Apache Camel web site and/or download the PDF manual to learn more about Camel. This guide assumes that you have a basic understanding of Camel.

So the combination of Camel's Java DSL and the MEL facilitate the quick development of Monlets. By leveraging Camel, Monlet developers can also quickly implement many different event message design patterns (e.g., content-based routers, recipient lists, delayers, etc.).

Subsequent sections of this guide will show you how to combine the Camel DSL and the MEL to create Monlets.

3.2 SNMP Agent and MIB

AMon includes an embedded SNMP v2 Agent and accompanying Management Information Base (MIB). These components allow ActiveMQ to be monitored by SNMP-capable management systems. The MIB, which is included in the AMon product archive file, lists and describes all the ActiveMQ attributes that can be monitored via AMon's SNMP agent. Please refer to the MIB for a complete description of all the attributes.

AMon's SNMP agent also generates SNMP notifications (a.k.a., traps) and gives Monlets the capability to generate notifications via the Camel SNMP endpoint that AMon makes available to Monlets. See the SNMP Camel Endpoint section of the guide for examples of how Monlets can take advantage of this endpoint (component) to generate SNMP v2 notifications.

The SNMP Agent section of the guide describes how to configure the agent.

4 The Anatomy of a Monlet

In this section we'll take a closer look at the Monlet and its structure.

To recap, a Monlet is a monitoring agent or component that:

- Is defined and developed by the end-user
- Adheres to the AMon framework's MEL and Camel's Java DSL
- Subscribes to AMon event messages
- Monitors one or more ActiveMQ runtime objects
- Invokes one or more actions as a result of monitoring the runtime objects
- Is embedded within the ActiveMQ message broker
- Is a Camel Route

Copyright 2009, Total Transaction Management

A Monlet is a Java object class that must extend the com.ttm.activemq.monitor.Monlet abstract super class. The one abstract method within the Monlet super class is the configure()[2] method, and this method must be implemented by the Monlets that extend the super class.

A Monlet runs within the same JVM as the message broker; therefore, the Monlet developer should use caution and not design resource intensive Monlets. In other words, the Monlet should not include actions that will overly compete with the broker for valuable resources.

This guide provides a lot of example Monlet code and we'll start things off with the simplest one of all, which depicts what amounts to a dummy Monlet, because it does absolutely nothing. However, it is a good starting point from which we can begin to describe the overall structure of a Monlet.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

/**
 * A very basic Monlet that does nothing
 */
public class MyMonlet extends Monlet {

  public void configure() {
    // does nothing
  }

}
```

The first thing to note is the three import statements, which are the minimum required import statements. Next, note how you must extend the Monlet super class and implement the configure method, which in the above example does absolutely nothing. The Monlet's configure method is invoked when the Monlet first gets loaded by the corresponding Camel context's class loader. All of this occurs during the ActiveMQ message broker's initialization phase.

Okay, so we've built our first, albeit dummy, Monlet. Now let's start adding some intelligence one step at a time.

In this next block of code, we've added some intelligence to our dummy Monlet by adding this one line of Java DSL code, "monitor().to("seda:router")".

```
package com.ttm.activemq.monitor.monlets;
```

[2] The Monlet class extends Camel's RouteBuilder class, which is where the abstract configure() method is defined.

Copyright 2009, Total Transaction Management

```
import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

/**
 * A very basic Monlet that does nothing
 */
public class MyMonlet extends Monlet {

 public void configure() {

   // Subscribe to AMon events, then pipe the events to a seda queue with the name, "router".
   // All Monlets subscribe to AMon events via the 'monitor' DSL verb, which is an extension to
   // the Camel DSL's repertoire of verbs.
    monitor().to("seda:router");
 }
}
```

However, this one simple line of code does a lot so let's take it apart and describe everything it does for us. First, it introduces the Camel Java DSL and a couple of its verbs[3] ("monitor" and "to"). The "monitor" verb is the extension to the DSL for the AMon product. This verb is used by the Monlet to subscribe to and consume AMon event messages; therefore, you must have at least one Monlet in your Camel context invoke this verb in order to receive AMon event messages. Whether each and every Monlet subscribes to AMon event messages or only one subscribes, depends on how you design your Monlets within a Camel context (more on this later).

The AMon event messages are delivered *synchronously* to the Monlet. In essence, the AMon framework issues a call to the monitor verb's underlying logic to deliver the event message. It is important to also note that the AMon framework is an extension of the ActiveMQ message broker. This means that the AMon framework / message broker has given execution control to the Monlet; therefore, it is critically important that the Monlet quickly return control to the calling AMon framework. This is accomplished by the "to" verb, which redirects the event messages so they are *asynchronously* processed in a separate thread (or threads) of execution. In this case, the "to" verb places the message in a "seda:" Camel endpoint component, which maps to a Java BlockingQueue (not to be confused with a JMS or ActiveMQ queue). To learn why the term "seda" is used, click here. In the example above, the seda blocking queue is called "router".

The "monitor().to("seda:router")" DSL statement implements our first messaging pattern, which is a simple pipe that consumes event messages from the AMon framework and pipes them into a Java BlockingQueue called "router". You can assign any name to a seda queue (Camel endpoint) and the name "router" has been chosen, because as you'll soon see our next Monlet sample code will include a content-based router that processes the messages from the "seda:router" processing queue.

---

[3] A DSL 'verb' is a Java method that is statically imported by the Monlet super class.

Copyright 2009, Total Transaction Management

Camel's Java DSL is expressed using what is referred to as the "fluent builder syntax". With this syntax, you stack methods that are separated with the dot character. In the above example, the 'monitor' verb represents the consumer that consumes event messages and then hands the messages to the producer, which is represented by the 'to' verb. The producer places the messages in the seda blocking queue called, "router".

All the Monlets that you deploy to a particular Camel context can inter-communicate with one another via the seda queues. So for example, your Camel context may comprise a Monlet whose sole purpose is to consume AMon event messages and place those messages in a "seda:router" endpoint. A different Monlet within the same Camel context can then read messages from the "seda:router" endpoint for subsequent processing. So the seda queues are a form of inter-Monlet communication mechanism for those Monlets that reside within the same Camel context. Monlets cannot communicate across Camel contexts.

So you're now probably asking how we are going to process those messages that are being piped into the asynchronous blocking queue called, "router". To address that question, we have to define another messaging pattern that reads event messages from the "router" blocking queue. So in this next block of Monlet source code we introduce the MEL and use it in conjunction with the DSL to construct a simple content-based router messaging pattern for our Monlet. This router reads messages from the "router" queue and routes the messages based on their event type. Figure 1 below is a pictorial representation of our content based router.

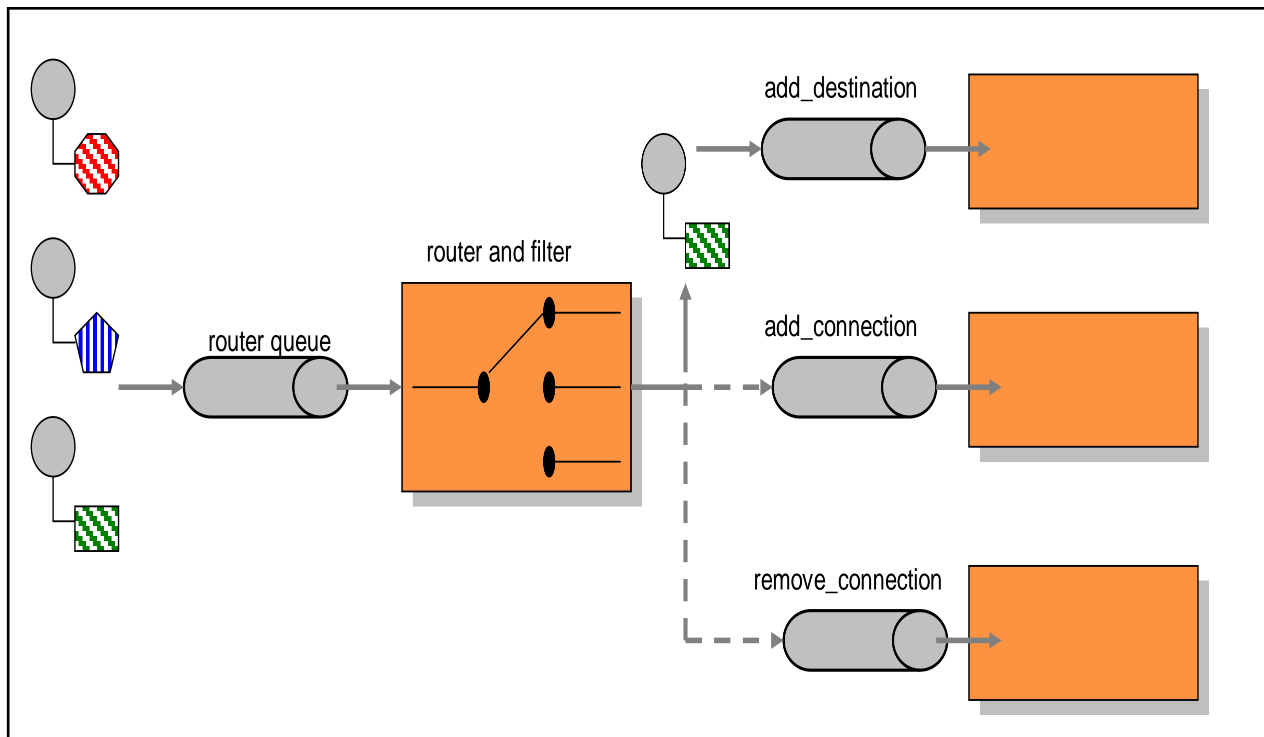


Figure 3 Content Based Router

The router is also acting as a message filter, because it discards all AMon event messages types that it is has no interest in.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

Below is the source code for the corresponding Monlet. Note how the MEL is used to define and test a condition, which in this case tests the event message type. The MEL must always be used within a Camel 'language' clause as follows.

language(MEL, "${*UEL expression*}")

This Monlet also introduces you to the 'from' verb, which is used by the Monlet to subscribe to and read from the seda queues. The 'from' verb can be used to subscribe to a myriad of different Camel endpoint components; click here to see the full list. The AMon-specific 'monitor' verb is an extension of the 'from' verb. The 'to' verb is used to send or produce messages to these same endpoint components. The code also introduces you to the 'choice', 'when', 'otherwise', and 'end' verbs, which allow you to construct a conditional statement much like an "if then else if" like conditional statement. To learn more about Camel and its Java DSL, visit the Apache Camel web site and and/or download Camel's PDF manual.

The comments within the example Monlet source code provide additional information, so please make sure to read the comments.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

/**
 *   This Monlet first consumes AMon events, then uses a content-based  router to filter
  *  out all but 3 AMon  event types.   The 'to', 'from', 'when', etc verbs are part of the Camel
  *  DSL.  All Monlets must extend  the Monlet super class and implement the configure
  *  method.
  */
public class MyMonlet extends Monlet {

   public void configure() {

   // Subscribe to AMon events, then pipe the events to the Monlets content-based  router.
   // All Monlets subscribe to AMon events via the 'monitor' DSL verb, which is an
   // extension to the Camel  DSL's repertoire of verbs.

   monitor().to("seda:router");

   // This is the Monlet's content based router that routes to processors based on AMon
   // event types. Note how this particular router only allows a subset of AMon event types to
   //  be passed on to the subsequent processors. The block of code that comprises the router
   //  illustrates the use of Camel's routing DSL in combination with the MEL.
   // The 'from', 'when', 'to', 'otherwise', 'language', and 'end' verbs pertain to the DSL,
   //  while the 'eventType == …' strings are MEL expressions.
```

Copyright 2009, Total Transaction Management

```
    from("seda:router").
     choice().
       when().language(MEL,"${eventType == ADD_DESTINATION}").
           to("seda:add_destination").
       when().language(MEL,"${eventType == ADD_CONNECTION}").
           to("seda:add_connection").
       when().language(MEL,"${eventType == REMOVE_CONNECTION}").
           to("seda:remove_connection").
       otherwise().
           to("seda:ignore").
     end();

    // The 'otherwise' verb is optional. If you leave it out, all unmatched messages will be
    // dropped.

    // This processor will be given the add destination event types
    from("seda:add_destination").
      process(new Processor() {
       public void process(Exchange exchange) throws Exception {
          // do some sort of processing here
        }
    });

    // This processor will be given the add connection event types
    from("seda:add_connection").
      process(new Processor() {
       public void process(Exchange exchange) throws Exception {
          // do some sort of processing here
       }
    });

    // This processor will be given the remove connection event types
    from("seda:remove_connection").
      process(new Processor() {
       public void process(Exchange exchange) throws Exception {
           // do some sort of processing here
       }
    });

    // All other event types can be put in the bit bucket.
    from("seda:ignore").end();

  }
}
```

Copyright 2009, Total Transaction Management

In the above example, all the logic is contained within one Monlet. However, you can split the logic across multiple Monlets. For example, one Monlet can be responsible for implementing the content-based router logic while other Monlets can be responsible for subscribing to their respective seda queues. The following snippet of code depicts a Monlet whose sole purpose is to process messages that arrive at its seda queue.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

/*
 *  This is a Monlet that reads from a seda queue and relies on another Monlet w/in the
 *  same Camel context to  subscribe to AMon event messages and place those messages
  *  in its seda queue.
 */
public class AddDestinationMonlet extends Monlet {

    public void configure() {

     from("seda:mySedaQueue ").
       process(new Processor() {
        public void process(Exchange exchange) throws Exception {
           // do some sort of processing here
        }
    });
  }
}
```

Take special note that a 'from' verb that reads from a seda queue will run within its own dedicated thread of execution; therefore, to prevent too much thread proliferation, make wise use of statements like 'from(seda:…)'.

One technique for cutting down on the 'from(seda:…)' verb clause is to use a 'choice' without adding an 'otherwise'; therefore, any unmatched exchanges will be dropped by default. If you elect to not use an 'otherwise', you may notice the following warning message being written to the console when the broker starts.

```
WARN  ChoiceType  - No otherwise clause was specified for a choice block -- any unmatched
exchanges will be dropped
```

Another technique is to place the processing logic within the 'when' clause. For example,

```
…

from("seda:mySedaQueue").
```

```
choice().

    // The processor within this 'when' clause will be invoked if the event message type
    // is of type MSG_DELIVERED or MSG_CONSUMED
    when().language(MEL,"${eventType == MSG_DELIVERED || " +
    "eventType == MSG_CONSUMED }").
      process(new Processor() {
        public void process(Exchange exchange) throws Exception {
        // do some processing here…
     }}).

    // The processor within this 'when' clause will be invoked if the event message type
    // is of type ADD_CONNECTION
     when().language(MEL,"${eventType == ADD_CONNECTION}").
      process(new Processor() {
        public void process(Exchange exchange) throws Exception {
        // do some processing here…
     }}).

end();

…
```

If you prefer to have an exception for an unmatched exchange, you can add a throwFault to the otherwise. For example

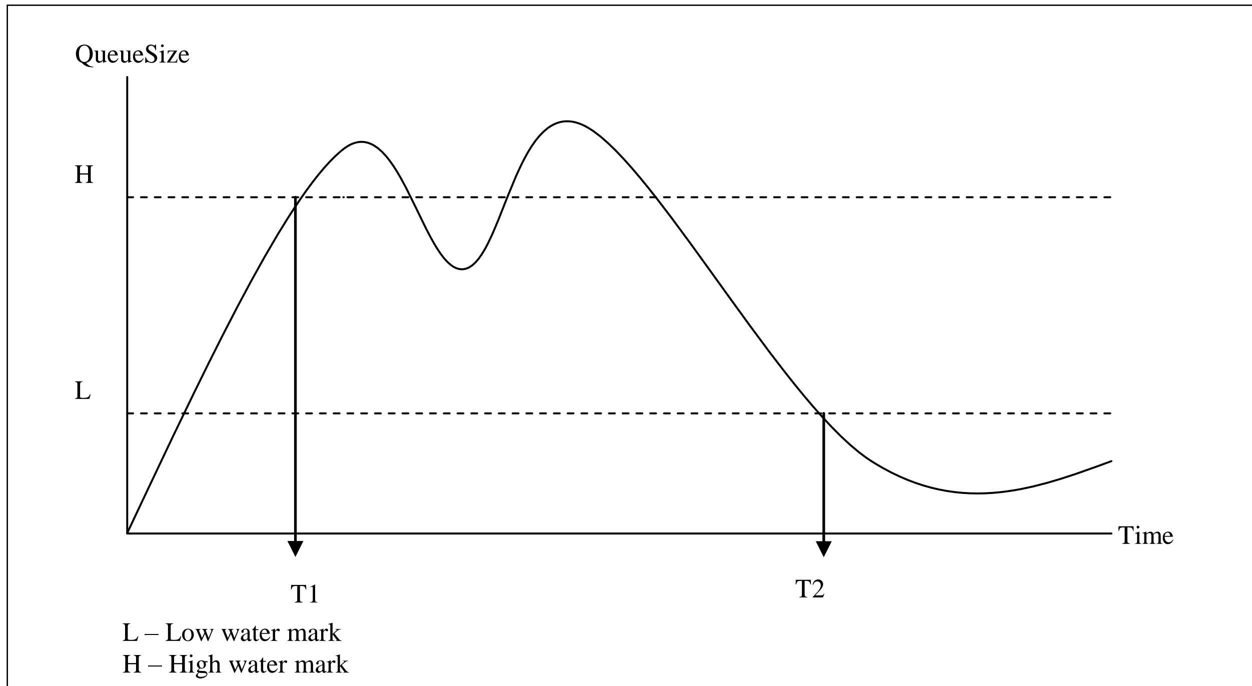
```
...otherwise().throwFault("No matching when clause found on choice block");
```

This next Monlet takes things a lot further by actually monitoring a broker resource, but let's first describe how the Monlet works and what it monitors.

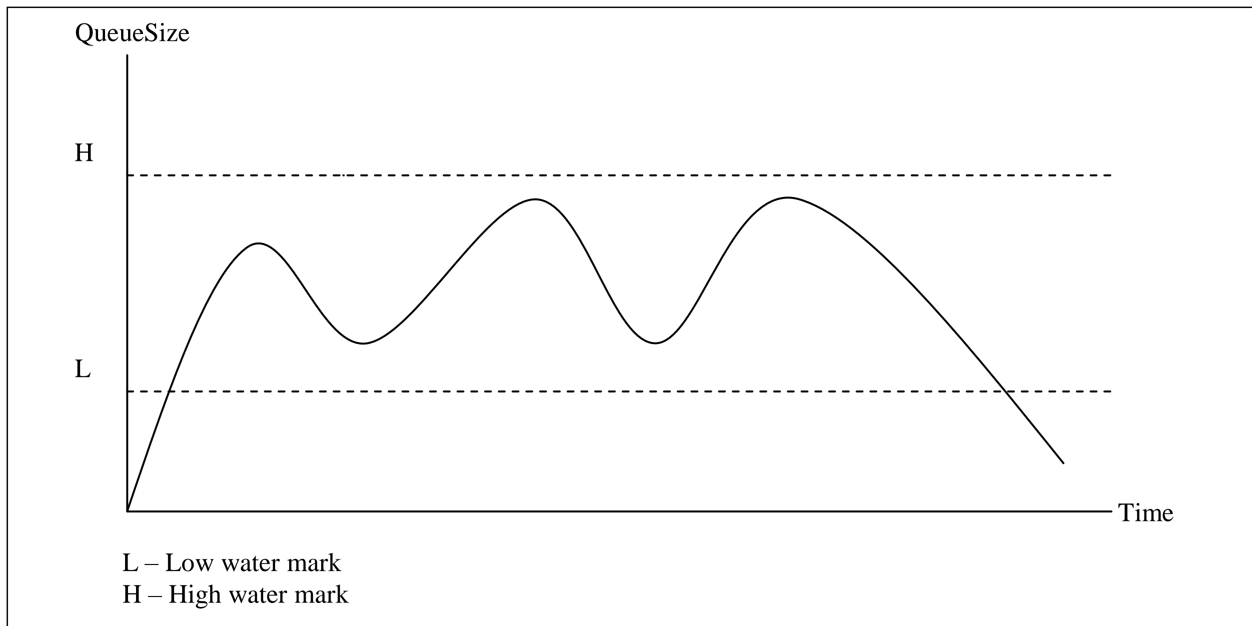
1. It will monitor the 'size' attribute for a queue called, "TEST.Q".
2. It monitors the queue size each time a message is added to or removed from the queue.
3. A high and low water mark have been specified for the queue size attribute.
4. It will invoke an action (processor) only if:
    a. the high-water mark is breached w/out a previous breach occurring at the high-water mark
    b. the lower-water mark is breached AND there has been a previous breach of the high-water mark. When this condition occurs the monitor is reset.

In the time diagram below, the monitor will invoke the processor (action) at T1 and T2. The monitor's state will also be reset at T2.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com



In the time diagram below, the monitor does not invoke any functions, because the high-water mark was never breached.



What follows are a couple of source code examples for the previously described Monlet. The first example illustrates how the Monlet utilizes a series of filters and content-based routers. However, this series of filters and routers results in several ‘from’ verbs, so the second example illustrates how you can collapse everything into one ‘from’ verb.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
/**
 * This Monlet monitors a queue called "TEST.Q" and invokes actions whenever the number of
 * pending messages in the queue enters a high or low water mark.
 */
public class Monlet4 extends Monlet {

  // Used to maintain state
  private boolean crossedIntoHighWaterMark = false;

  public void configure() {

    // Capture all AMon event message types. All Monlets subscribe to AMon events via the
    // 'monitor' DSL  verb, which is an extension to the Camel DSL's repertoire of verbs.

    monitor().to("seda:check_event_type");

    // This Monlet is only interested in two events. Those being when the broker has
    //  received a message from a producer or when the broker has dispatched a message to a
    // consumer. All other event messages will not contain the information that this Monlet
    // is interested in.

    from("seda:check_event_type").choice().
      when().language(MEL,"${eventType == MSG_DELIVERED || " +
                                    "eventType == MSG_CONSUMED}").
         to("seda:check_q_name").
      end(); // the end for the when clause


    //  Okay, we've now filtered out all message types except the two we're interested in.
    //  Now lets filter out all message types that are not associated with the queue that
    //  we're interested in. In other words, lets capture event messages associated only with
    //  the queue called 'TEST.Q'. Note that only event types of the type that we're interested
    //  in has the 'destination' variable mapped to the broker's destination HashMap.
    from("seda:check_q_name").choice().
      when().language(MEL,"${destination.name == 'TEST.Q'}").
        to("seda:check_queue_size").
      end(); // the end for the when clause

    //  Lets now check the queue's size and determine if the queue is in the high or low water
    //  mark range. We don't care if it is in between the two ranges.
```

```
  from("seda:check_queue_size"). choice().
    // if we're at or above the high water mark, route to the 'high' seda q
    when().language(MEL,"${destination.destinationStatistics.messages.count >= 100}").
      to("seda:high").
    // if we're in the low water mark range, route to the 'low' seda queue
    when().language(MEL,"${destination.destinationStatistics.messages.count <= 50}").
      to("seda:low").
     // else it is in between the two and we don't care
    end(); // the end for the when clause

  // The queue size is in the high water mark - if we had not previously crossed it send an
  //  alert and set the high water mark flag
  from("seda:high").process(new Processor() {
     public void process(Exchange exchange) throws Exception {
       System.out.println("In the high water mark");
       if(!crossedIntoHighWaterMark){
         crossedIntoHighWaterMark = true;
         // we've crossed into the high water so invoke an action, which in this case
         // simply prints out a message
         System.out.println("WARNING: high water mark has been breached");
       }
   }}); //end this Monlet route

   // we're in the low water mark
   from("seda:low").process(new Processor() {
     public void process(Exchange exchange) throws Exception {
       // reset the highwater flag since we've crossed back into low water mark and
       // invoke some action, , which in this case is to simply print out a message
       if(crossedIntoHighWaterMark){
         crossedIntoHighWaterMark = false;
         System.out.println("Low water mark has been breached");
       }

   }}); //end this Monlet route

 }

}
```

This is the shortened version of Monlet4, note how all the logic is now contained within one from verb or clause.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
```

```
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

public class Monlet4 extends Monlet {

  private boolean crossedIntoHighWaterMark = false;

  @Override
  public void configure() {

    // Capture AMon event message types
    monitor().to("seda:Monlet4");

    // This Monlet is only interested when a message has arrived from a
    // producer or when the broker has dispatched a message to a consumer.
    from("seda:Monlet4").

      choice().

        when().language(MEL,"${(eventType == MSG_CONSUMED || " +
                "eventType == MSG_DELIVERED) && destination.name == 'TEST.Q' &&" +
                "destination.destinationStatistics.messages.count >= 100}").
          process(new Processor() {
            // we're in the high water mark - if we had not previously crossed it
            // send an alert and set the high water mark flag
            public void process(Exchange exchange) throws Exception {
               System.out.println("In the high water mark area");
               if (!crossedIntoHighWaterMark) {
                  crossedIntoHighWaterMark = true;
                  // we've crossed into the high water so invoke an action, which in this case
                  // simply prints out a message
                  System.out.println("WARNING - High water mark has been breached");
               }
            }
          }).

        when().language(MEL,"${(eventType == MSG_CONSUMED || " +
                "eventType == MSG_DELIVERED) && destination.name == 'TEST.Q' &&" +
                "destination.destinationStatistics.messages.count <= 50}").
          process(new Processor() {
            // we're in the low water mark
            public void process(Exchange exchange) throws Exception {
                // reset the highwater flag since we've crossed back into low water mark and
                // invoke some action, , which in this case is to simply print out a message
                if (crossedIntoHighWaterMark) {
                    crossedIntoHighWaterMark = false;
                    System.out.println("Low water mark has been breached");
```

```
            }
        }
      }).

    end();

  }
}
```

In the previous two examples, note the use of the Processor object's 'process' method. It is in this method that some form of message processing takes place like transforming the event message to an email message, setting a header for a subsequent filter, or setting an Exchange property, etc. Here's a snippet from a simple Monlet that serves as a further example; please refer to the comments for additional information.

```
…

/**
    Let's assume that a condition above has tested positive and that the message has been sent
     to this seda queue for processing.
**/

from("seda:low").process(new Processor() {

        // This is where we do the message processing like transform the message (if necessary)
        // prior to sending it to the next Camel endpoint (e.g., smtp, snmp, file, etc).  The
        // Exchange object is always passed from one endpoint to the next in the processing
        // pipeline.
        public void process(Exchange exchange) throws Exception {

            // Save the AMon BaseEvent object (event message)  that arrived in case we want
            // to use it for processing
            BaseEvent baseEvent =  (BaseEvent) = exchange.getIn().getBody();

            // Do some processing here…

           // Transform the body prior to sending it to the next endpoint. In this case, we're
           // simply transforming the BaseEvent object to a simple String object.
           exchange.getIn().setBody("Transformed body");

           // You can also set message headers for the next endpoints to receive the Exchange
           exchange.getIn().setHeader("key", "value");

          // The Exchange  object also allows you to set and get Exchange properties.
          // These properties, as well as message headers, can then be referenced by the next
          // endpoint in the pipeline. Here are some examples.
```

Copyright 2009, Total Transaction Management

```
        Map exchangeProps = exchange.getProperties();
        String someProp      = exchange.getProperty("key");
        exchange.setProperty("key", "value");

      }

   // After processing and transforming the message, send it to the next
   // Camel endpoint. Note that this is optional and that the Monlet may have terminated
  //  with the previous process method.
   }).to("some other endpoint");

…
```

In the above example, the action is to process the message, transform the message, and then send it to the next endpoint in the processing pipeline. The next endpoint can be any one of Camel's multitude of supported endpoints (e.g., smtp, ibatis, file, etc.).

Of special note is the SNMP endpoint that AMon provides and is described in the SNMP Camel Endpoint section of the guide. This next, shortened version of Monlet4, illustrates how a message is sent to AMon's SNMP Camel endpoint whenever one of the queue's thresholds has been breached. When the SNMP endpoint is sent a message, it will generate a SNMP notification (trap). Also note the reliance on AMon's external Properties File to make the Monlet dynamically configurable. Please refer to the comments for additional information.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

public class Monlet4 extends Monlet {

   private boolean crossedIntoHighWaterMark = false;

   @Override
   public void configure() {

      // Capture AMon event message types and forward them to the seda queue called,
      // "Monlet4".
      monitor().to("seda:Monlet4");

      /* **********
       This Monlet is only interested when a message has arrived from a  producer or when the
       broker has dispatched a message to a consumer. It is also relying on the following
       external properties:
```

```
 monlet4QName – This property specifies the name of the queue to monitor
 maxThreshold – This property specifies the maximum threshold for the queue
 minThreshold – This property specifies the minimum threshold for the queue

 Values for the above properties can be dynamically assigned. In other words,
 you do not have to stop and start the broker to change their values. See the
 Properties File section for more information on AMon properties.

********** */
from("seda: Monlet4").

 choice().

   when().language(MEL, "${(eventType == MSG_CONSUMED || " +
        "eventType == MSG_DELIVERED) && destination.name == " +
        "props.monlet4QName && " +
        "destination.destinationStatistics.messages.count >= " +
        "props.maxThreshold}").
        process(new Processor() {
          // we're in the high water mark - if we had not previously crossed it
          // send a SNMP trap and set the high water mark flag
         public void process(Exchange exchange) throws Exception {
           if (!crossedIntoHighWaterMark) {
            crossedIntoHighWaterMark = true;
            // Let the SNMP producer know to send the trap
            exchange.getIn().setHeader("sendSnmpTrap", "true");
            Monlet.setSnmpTrapMonitorPropertyName(exchange,"messages.count");
            Monlet.setSnmpTrapMonitorMessage(exchange,
                "WARNING high water mark was breached");
           }
        }
       }).to("seda: Monlet4.check4trap").

   when().language(MEL, "${(eventType == MSG_CONSUMED || " +
        "eventType == MSG_DELIVERED) && destination.name == " +
        "props.monlet4QName && " +
        "destination.destinationStatistics.messages.count <= " +
        "props.minThreshold}").
        process(new Processor() {
         // we're in the low water mark
         public void process(Exchange exchange) throws Exception {
           // reset the highwater flag since we've crossed back into low water
           if (crossedIntoHighWaterMark) {
             crossedIntoHighWaterMark = false;
             // Let the SNMP producer know to send the trap
            exchange.getIn().setHeader("sendSnmpTrap", "true");
```

Copyright 2009, Total Transaction Management

```
                // These two statics methods are used setting notification attributes.
                // These methods are described in the SNMP Camel Endpoint section
                // of the guide.
                Monlet.setSnmpTrapMonitorPropertyName(exchange,"messages.count");
                Monlet.setSnmpTrapMonitorMessage(exchange,
                     "The low water mark was breached");
               }
             }
      }).to("seda: Monlet4.check4trap").

    end();

    //  Via the use of a 'filter',  this seda consumer is checking to see if the message is to
    //  be forwarded on to the SNMP endpoint, which raises a SNMP notification (trap)

    from("seda: Monlet4.check4trap").filter(header("sendSnmpTrap").
         isEqualTo("true")).to("snmp:trap");

   }
}
```

5 The MEL

The MEL is an AMon-specific extension of the UEL or JUEL. In this section, we describe the variables, constants, and methods that define the MEL.

By the time a Monlet receives an AMon event message, a MEL context has been created for that message. This MEL context binds variables to the event message's instance properties, supplies constants, and supplies methods. All of these variables, constants, and methods can then be referenced within the Monlet's MEL conditional expression or predicate. Let's go back to one of the previous example Monlets that implemented the following content-based router message pattern.

```
from("seda:router").
 choice().
    when().language(MEL,"${eventType == ADD_DESTINATION}").
         to("seda:add_destination").
    when().language(MEL,"${eventType == ADD_CONNECTION}").
         to("seda:add_connection").
    when().language(MEL,"${eventType == REMOVE_CONNECTION}").
         to("seda:remove_connection").
```

```
end();
```

And in particular, let's take a closer look at this statement.

```
language(MEL,"${eventType == ADD_DESTINATION}").
```

A Monlet uses the 'language' Camel DSL clause to define a conditional expression. The MEL string constant tells Camel to use the MEL language and everything inside the "${}" string defines the MEL conditional expression. In the above examples, the 'eventType' is a variable that is assigned the value of the message's eventType property and ADD_DESTINATION is a constant that has been mapped to the corresponding event message type.

5.1 Event Types as Constants

The table below lists and briefly describes all the events that can be reported to the Monlet via the eventType variable. The event's type (e.g., ADD_CONNECTION) is also a MEL constant that is used by Monlets to test for that particular event type. Examples of these constants and how they are referenced within a MEL expression were provided in the previous sections.

*Please note that, by default, AMon will **not** generate any event messages*. You must decide what event types you'd like your Monlets to be informed of and then enable the publishing of the corresponding event messages. Use caution when choosing what events to enable, because enabling too many or all events may result in an overall performance degradation. See Configuring AMon for details on how to enable and disable event publishing.

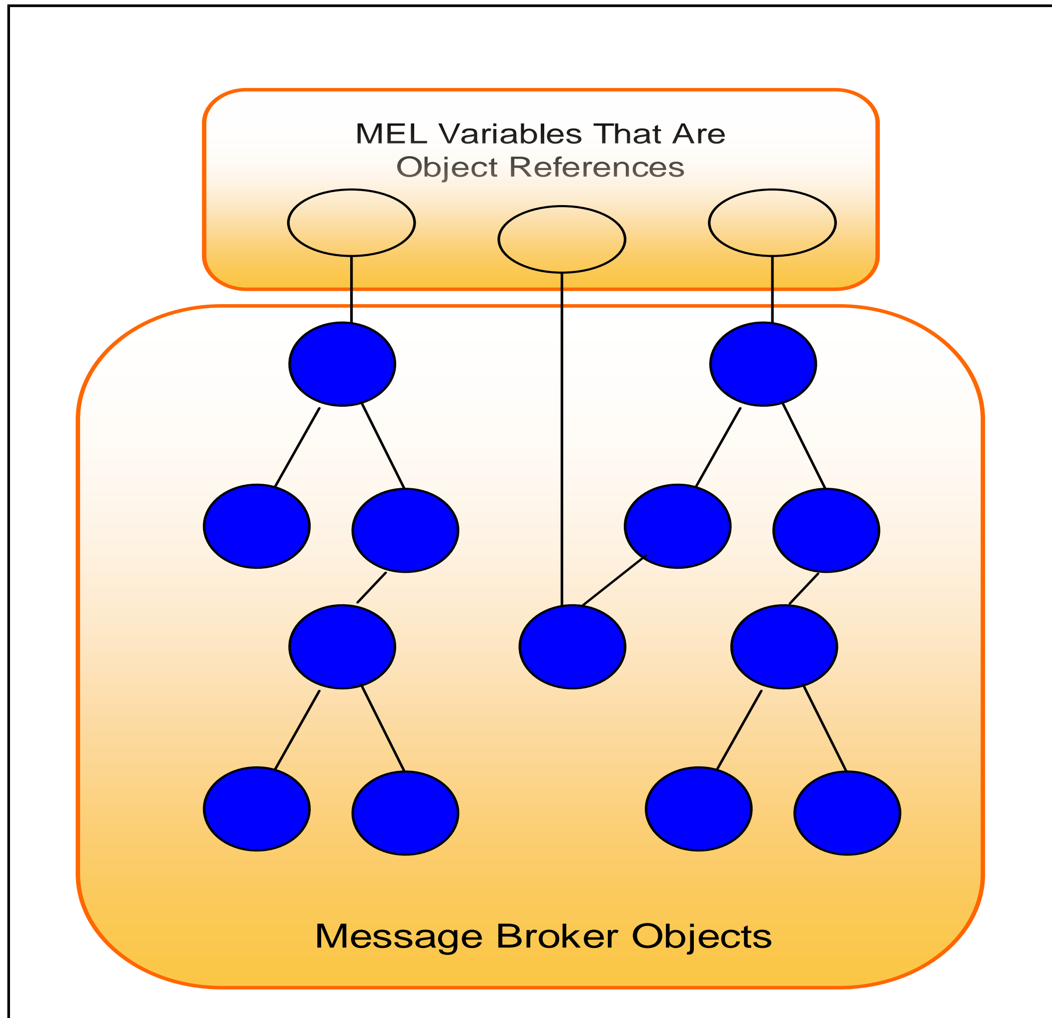
Event Type	Description
ADD_BROKER	A remote Broker connects to this broker
ADD_CONNECTION	An ActiveMQ client (consumer or producer) is establishing a connection with this broker.
ADD_CONSUMER	A consumer client is being added to this broker
ADD_DESTINATION	A destination is being added to this broker. Usually, this event occurs as a side-effect of sending a message to a destination that does not exist yet.
ADD_PRODUCER	A producer client is being added to this broker
ADD_SESSION	A JMS session has been established with this broker
BEGIN_TRX	A transaction is beginning
COMMIT_TRX	A transaction is being committed
FAST_PRODUCER	The broker has detected a fast producer.
FORGET_TRX	A transaction is being forgotten
LOG_ENTRY	A log4j message has been written to the ActiveMQ log file. Please see Log4J Monitoring for more details on how to set up and configure AMon to publish this event.

MASTER_BROKER	This broker, who was previously a slave, is now a master broker.
MSG_ACK	Acknowledge the receipt of a message by a client
MSG_CONSUMED	A message in a queue has been consumed by a consumer client. Note that consuming a message from a topic does not generate this event type.
MSG_DELIVERED	A message has been delivered to this broker.
MSG_DISCARDED	Called when a message is discarded (e.g. running low on memory). This will happen only if the policy is enabled (e.g. non durable topics)
MSG_EXPIRED	A message has expired
MSG_SEND	A message is being sent to a remote broker.
POST_DISPATCH	A message has been dispatched to a consumer
PRE_DISPATCH	A message is about to be dispatched to a consumer
PREPARE_TRX	A transaction is being prepared for commit
REMOVE_BROKER	A remote broker disconnects from this broker
REMOVE_CONNECTION	A client has disconnected from this broker.
REMOVE_CONSUMER	A consumer client is being removed from this broker
REMOVE_DESTINATION	A destination is being removed from this broker.
REMOVE_PRODUCER	A producer client is being removed from this broker.
REMOVE_SESSION	A JMS session has been removed from this broker
REMOVE_SUBSCRIPTION	A durable subscription is being removed from this broker.
ROLLBACK_TRX	A transaction is being rolled back
SEND_TO_DLQ	A message is being sent to the dead letter queue
SLOW_CONSUMER	The broker has detected a slow consumer
TIMER_INTERVAL	AMon's configurable interval event timer has expired (see The Interval Timer).
USAGE_FULL	Called when a broker resource reaches a limit

Table 1 AMon Event Types

5.2 Variables

In the previous examples, we saw how the value of the 'eventType' String variable is tested against a String constant. However, in some cases the MEL defines variables whose corresponding data types are ActiveMQ objects (i.e., the variables are object references). When a variable's data type is an ActiveMQ object, your Monlet can invoke all the standard JavaBean 'get' and 'is' methods associated with that particular object; this allows your Monlet to acquire the values associated with that object's instance properties. In some cases, one or more of the object's 'get' methods will return a reference to another ActiveMQ object, whose instance property values you then access through its 'get' and 'is' methods. This is how you can *walk* through the message broker's object hierarchy tree to acquire information regarding the broker's operational or runtime state.



For convenience, the MEL provides you with some variables that reference an ActiveMQ object that is more than one level deep within the object tree. As you'll see later in the section, one such example is the 'destinationStatistics' variable.

The table below lists all the variables that the MEL presents to the Monlet developer and also provides examples of how to reference the 'get' and 'is' object methods. You'll note that the broker, brokerService, brokerView, eventType, props, and vm variables are valid for all event types, while the others are valid only for a subset of all the event types. When a MEL variable is an object reference, you will be referred to that object's JavaDoc URL for a complete list of that object's 'get' and 'is' methods. An index to all ActiveMQ classes can be found here.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

Variable Name	Description
activeMQDestination	This variable is a reference to the ActiveMQDestination object and is only valid for the following event types: ADD_DESTINATION and REMOVE_DESTINATION. Usually, the ADD_DESTINATION event is invoked as a side-effect of sending a message to a destination that does not exist yet. Here are a couple of example use cases for this object. Is the destination a topic? language(MEL,"${amqDestination.topic}") Is the destination temporary? language(MEL,"${amqDestination.temporary}") Is it a composite destination? language(MEL,"${amqDestination.composite}")
broker	This variable is a reference to an AMon specific object that both implements and extends the ActiveMQ Broker interface; the variable is valid for all event types. Example usage: Is the broker's name set to 'mybroker'? language(MEL,"${broker.brokerName == 'mybroker'}") Note how the string literal 'mybroker' must be surrounded by single quotes.
brokerInfo	This variable is a reference to the BrokerInfo object, and it is only valid for the ADD_BROKER and REMOVE_BROKER event types.
brokerService	This variable is a reference to the BrokerService object, and it is valid for all event types.
brokerView	This variable is a reference to the BrokerView object and is valid for all event types. The BrokerView object implements the BrokerViewMBean interface. Through this variable, your Monlet can access all the 'get' and 'is' methods associated with this interface. Example usage: Has the total message dequeue count for this broker exceeded 60000?

Copyright 2009, Total Transaction Management

	language(MEL,"${brokerView.totalDequeueCount > 60000}") Is this broker's name set to 'mybroker' and is it a slave broker? language(MEL, "${brokerView.brokerName == 'mybroker' && brokerView.slave}")
connection	This variable is a reference to the Connection object, which is valid for the following event types: ADD_BROKER and REMOVE_BROKER
connectionContext	This variable is a reference to the ConnectionContext object, which is valid for all event types except the following: ADD_BROKER, REMOVE_BROKER, MASTER_BROKER, LOG_ENTRY, and TIMER_INTERVAL Here is an example use case. Does the connection pertain to an IP address of '123.45.67.89'? language(MEL,"${connectionContext.connection.remoteAddress == '123.45.67.89'}") From the connectionContext object, you can also get a reference to the Connection object and it's ConnectionStatistics (see next variable). From the connectionContext you can also get a reference to the Connector object and its ConnectorStatistics. So what's the difference between a Connection and Connector object? A Connector is an object that creates and manages client Connections that communicate to the Broker. One example of a Connector is the 'Transport Connector', which you may have seen referenced in the ActiveMQ message broker's XML configuration file as follows. <transportConnectors> <transportConnector name="openwire" uri="tcp://localhost:61616" discoveryUri="multicast://default"/> </transportConnectors> Note that a Connector is capable of listening for and accepting Connection requests from both clients and other brokers.
connectionInfo	This variable is a reference to the ConnectionInfo object and it is only valid for the following event types: ADD_CONNECTION and REMOVE_CONNECTION. Please note that the Connection that this ConnectionInfo object pertains to can be associated with either

Copyright 2009, Total Transaction Management

	a client or another broker. Here are some example use cases. Does the connection pertain to a user by the name of ‘joe’? language(MEL,"${connectionInfo.userName == ‘joe’}") Is the client id associated with this connection set to ‘client123’? language(MEL,"${connectionInfo.connectionId.value == ‘client123’}") Is the value of the corresponding connection id equal to ‘foo’? language(MEL,"${connectionInfo.connectionId.value == ‘foo’}")
connectionStatistics	This variable is a reference to the ConnectionStatistics object, which is valid for all event types except the LOG_ENTRY, PRE_DISPATCH, POST_DISPATCH, MASTER_BROKER, SLOW_CONSUMER, and TIMER_INTERVAL events. The reference to the ConnectionStatistics objects is acquired from the ConnectionContext object. Example usage: Have there been more than 1000 messages dequeued via this connection? language(MEL,"${connectionStatistics.dequeues.count > 1000}") This is a convenience variable, because it can be referred to instead of having to refer to the connection or connectionContext variables. This is an example of a variable that is a direct reference to a couple of layers down an object hierarchy. For example, if you used the connectionContext variable (see above) to acquire the connectionStatistics, you’d end up with the following MEL string, which is much longer than the one above. language(MEL, "${connectionContext.connection.statistics.dequeues.count > 1000}")
connectorStatistics	This variable is a reference to the ConnectorStatistics object, which is valid for all event types except the LOG_ENTRY, PRE_DISPATCH, POST_DISPATCH, MASTER_BROKER, SLOW_CONSUMER, and TIMER_INTERVAL events.
consumerBrokerExchange	This variable is a reference to the ConsumerBrokerExchange object, which is only valid for the MSG_ACK event type.
consumerInfo	This variable is a reference to the ConsumerInfo object.

Copyright 2009, Total Transaction Management

	This variable is only valid for the SLOW_CONSUMER, ADD_CONSUMER and REMOVE_CONSUMER event types.
destination	This variable is a reference to the Destination object associated with the event. It is only valid for the following event types: MSG_SEND, MSG_EXPIRED, MSG_CONSUMED, MSG_DELIVERED, MSG_DISCARDED, SEND_TO_DLQ, SLOW_CONSUMER and USAGE_FULL. Example usage: Is the destination associated with this event called 'TEST.Q' ? language(MEL,"${destination.name == 'TEST.Q'}")
destinationStatistics	This variable is a reference to the DestinationStatistics object and it is only valid for the following event types: MSG_SEND, MSG_EXPIRED, MSG_CONSUMED, MSG_DELIVERED, MSG_DISCARDED, SEND_TO_DLQ, SLOW_CONSUMER, and USAGE_FULL. From the DestinationStatistics object you can get statistics data for the event's corresponding destination. language(MEL,"${destinationStatistics.dequeues.count > 100}") language(MEL,"${destinationStatistics.enqueues.count > 100}") language(MEL,"${destinationStatistics.producers.count == 10}") language(MEL,"${destinationStatistics.consumers.count == 0}") This is a convenience variable, because it can be directly referred to instead of having to acquire it via the previously described 'destination' variable. This is an example of a variable that is a direct reference to an object that is a couple of layers into an object hierarchy. For example, if you used the destination variable (see above) to acquire the destinationStatistics, you'd end up with the following MEL string, which is a bit longer than the one above. language(MEL, "${destination.destinationStatistics.dequeues.count > 1000}")
eventType	This String variable will contain the value of the event message type. It is used by Monlets to determine the event message type. See the events table for a complete list of all event types. In the example below, the MEL conditional expression is checking for an event message of type of ADD_CONNECTION. language(MEL,"${eventType == ADD_CONNECTION }") The ADD_CONNECTION is a constant; therefore, it does not have

Copyright 2009, Total Transaction Management

	to be surrounded by single quotes. However, string literals will need to be surrounded by single quotes. This variable is valid for all event types.
loggingEvent	This variable is a reference to a log4j LoggingEvent object, which is only valid for LOG_ENTRY event type. Please see Log4J Monitoring for more details on how to set up AMon to publish LOG_ENTRY event. AMon will monitor only those messages that are generated by the "org.apache.activemq" package.
loggingEventMessage	This is a String object whose value is the message associated with a LoggingEvent object. This variable is only valid for LOG_ENTRY event types.
loggingEventPriority	This is a reference to a String object that is set to either "WARN', "ERROR, or "FATAL" and represents the priority of a LoggingEvent object. This variable is only valid for LOG_ENTRY event types.
messageAck	This variable is a reference to the MessageAck object and it is only valid for the MSG_ACK event type.
messageDispatch	This variable is a reference to the MessageDispatch object and it is only valid for the following event types: PRE_DISPATCH and POST_DISPATCH
messageReference	This variable is a reference to the MessageReference object and it is only valid for the following event types: MSG_EXPIRED, MSG_CONSUMED, MSG_DELIVERED, MSG_DISCARDED, and SEND_TO_DLQ.
producerInfo	This variable is a reference to the ProducerInfo object. This variable is only valid for the FAST_PRODUCER, ADD_PRODUCER and REMOVE_PRODUCER event types.
props	This variable, valid for all event types, is a reference to an AMon object that is used to contain properties that are specified via an external properties file. More information on the use of this variable can be found in section 9 "Configuring AMon". Example usage: Let's assume that there exists a property in the AMon properties object called 'vmThreadCount' and that the property's value is set to "100". The following then asks, have the total number of threads created and started since the Java virtual machine started reached 100 or more? language(MEL,

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

	"${vm.totalStartedThreadCount >= props.vmThreadCount}") The nice thing about using properties instead of constants is that you can dynamically change the properties' values without bouncing the message broker. The JUEL will coerce the 'vmThreadCount' property's String value to an integer. You can also use the bracket notation, instead of dot notation, to access the properties. This comes in handy if the property that you're accessing includes dots in its name, as this example illustrates. language(MEL,"${props['monitor.timer_interval'] >= 5000}")
removeSubscriptionInfo	This variable is a reference to the RemoveSubscriptionInfo object, and it is only valid for the REMOVE_SUBSCRIPTION event type.
sessionInfo	This variable is a reference to the SessionInfo object, which is only valid for the ADD_SESSION and REMOVE_SESSION event types. Here is an example use case. Is the session id set to 123456789? language(MEL,"${sessionInfo.sessionId.value == 123456789}")
subscription	This variable is a reference to the Subscription object, and is only valid for the SLOW_CONSUMER event type.
transactionId	This variable is a reference to the TransactionId object, and is only valid for the *_TRX (e.g., BEGIN_TRX) event types.
usage	This variable is a reference to the Usage object and it is only valid for the USAGE_FULL event. The USAGE_FULL event also provides references to the ConnectionContext and Destination objects.
vm	This variable is a reference to an AMon object that is called "VirtualMachine" and it is valid for all event types. The purpose of this variable is to give the Monlet developer access to real-time information for the hosting Java virtual machine. The VirtualMachine object essentially wraps the ManagementFactory object and also makes available some of the information that is acquired via the Runtime object. Please see VirtualMachine Object for a complete list and description of all the 'get' methods that are made available via this vm variable. Here is an example use case.

Copyright 2009, Total Transaction Management

	Have the total number of threads created and started since the Java virtual machine started reached 100 or more? language(MEL,"${vm.totalStartedThreadCount >= 100}")

Table 2. AMon Variables

The BaseEvent object is the body of the Camel 'in' message and can be acquired via the Camel Exchange object. Let's take the following sample source code, which is that of a simple Monlet that is interested only in LOG_ENTRY events. Note how after it has detected a LOG_ENTRY message, it sends or pipes the message to a Camel Processor object. The Processor objects must implement the process(Exchange) method, and it uses the Exchange object to acquire the 'in' message body, which is of type BaseEvent.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import com.ttm.activemq.monitor.event.BaseEvent;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

public class LogMonlet extends Monlet {

  @Override
  public void configure() {

   //  Capture all AMon event message types. All Monlets subscribe to AMon events via
   // the 'monitor' DSL   verb, which is an extension to the Camel DSL's repertoire of verbs.
  monitor().to("seda:logMonlet");

    // This Monlet is interested only in LOG_ENTRY event messages, which it will pipe
    //  to a Processor to extract some information regarding the logged message
    from("seda:logMonlet").choice().
       when().language(MEL,"${eventType == LOG_ENTRY}").
         process(new Processor() {
           public void process(Exchange exchange) throws Exception {
           // Get the message body, which is an AMon BaseEvent object
           BaseEvent baseEvent = (BaseEvent)exchange.getIn().getBody();

           // Now that we have the BaseEvent, we can get to all the AMon variables. So lets
           // print out some information regarding the LoggingEvent
           System.out.println("LogMonlet: Message message = " +
                   baseEvent.getLoggingEventMessage());
           System.out.println("LogMonlet: Message priority = " +
                   baseEvent.getLoggingEventPriority());
       }}).
     end();
  }
```

Copyright 2009, Total Transaction Management

```
}
```

So being able to access AMon variables via the BaseEvent comes in handy when creating a Camel Processor.

The Properties File section of this guide shows you how to use the BaseEvent object within a process() method to acquire a copy of AMon's dynamically configurable properties.

5.3 Methods

In previous sections, the guide describes the standard JavaBeans 'get' and 'is' accessor methods that are implicitly invoked via MEL conditional expressions. For example, the MEL expression, "${broker.brokerName}" invokes the broker object's getBrokerName() accessor method, which returns the broker's name. Note how you do not specify the 'get' portion of the method name and the first character after 'get' must be in lower case (this is in accordance with how the JUEL takes advantage of the JavaBeans standard). One disadvantage with this JUEL approach towards invoking the accessor methods is that you cannot pass parameters to the accessor methods. However, the MEL does provide direct access to a few objects' methods that do accept input parameters, but do not necessarily conform to the JavaBeans standard.

The first of these objects is the java.lang.String object and the following example illustrates how you can access, from within a MEL expression, the String object's methods that accept input parameters. In this example, we're invoking the startsWith() method, which is being used to determine if the broker name starts with "mybroker".

```
language(MEL,"${broker.brokerName.startsWith('mybroker')}")
```

For the above example, notice how the JUEL convention of invoking the methods is not being followed. That is, the method's entire name is being supplied. Some String methods are overloaded (i.e., there exists more than one method with the same name, but with different input parameters). ***Unfortunately, the JUEL does not inform the MEL as to the number of parameters associated with the method invocation. Therefore, the MEL always chooses the version of the method with the least number of parameters*; *the startsWith() method is one prime example***.

The MEL also exposes two methods that are provided by the 'broker' variable or object. One method is called getDestination(String name), which allows you to get a reference to a Destination object with the specified name. Here's an example where a reference to a Destination object with the name 'TEST.Q' is being acquired in order to get statistics on that destination.

```
language(MEL,"${broker.getDestination('TEST.Q').destinationStatistics.messages.count > 100}")
```

Copyright 2009, Total Transaction Management

Note how you must specify the method's full name.

The other 'broker' method that is exposed is called getDestinationStatistics(String name), which is a convenience method for accessing the Destination's statistics. Here's an example.

```
language(MEL,"${broker.getDestinationStatistics('TEST.Q').messages.count > 100}")
```

These two methods are handy to have if the Monlet has received an event message that is not associated with any particular Destination. In such a situation, the event message's 'destination' variable is not valid for the event. One such example is the timer event message, which is described next.

The two methods described above are provided by an AMon-specific object called MonitorBroker, which both implements and extends the ActiveMQ Broker interface. The two methods are an extension to the Broker interface.

5.4 Asynchronous Processing

Many of the example Monlets presented thus far take advantage of seda queues in order to asynchronously process the event messages that are published by the AMon framework. And as mentioned in chapter 4 (The Anatomy of a Monlet), the initial placement of an event message in a seda queue is a requirement, because the Monlet must quickly return control to the calling AMon framework.

When your Monlet design incorporates a pipeline of seda queues, you should note that as an AMon event message is forwarded across the pipeline, the operational run state of the ActiveMQ message broker can change. Therefore, the possibility does exist that the evaluation of a MEL conditional statement will return different values at different stages of the pipeline. For example, the queue size associated with an ActiveMQ queue may not have breached a certain threshold at stage 1 of the pipeline, but by the time the same event message is processed at stage 3, the queue size may have breached the threshold.

6 The Interval Timer

AMon includes an interval timer that is configured via the "monitor.timer_interval" property. This property, which is specified in the AMon Properties File, is used to enable or disable the timer and also to specify the timer's interval time period in milliseconds.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

By default, the timer is disabled; however, when it is enabled it will publish a TIMER_INTERVAL event message every interval time period. Refer to the EventTypesTable for a listing of all variables that are valid for timer event messages. To enable the timer, you simply assign the "monitor.timer_interval" property an integer value greater than zero. For example, to have your Monlets receive a TIMER_INTERVAL event message every 100 milliseconds, you add the following line to the AMon Properties File.

```
monitor.timer_interval=100
```

If you update the property, while the message broker is active, the new property value will take effect as soon as you save the file. To disable the timer, simply assign the property a value of zero.

```
monitor.timer_interval=0
```

If you want to create a Monlet that monitors a resource, but does not do so via potentially high-volume event messages (e.g., MSG_DELIVERED, MSG_CONSUMED, MSG_ACK), then it can rely on the TIMER_INTERVAL event messages.

7 SNMP Camel Endpoint

As mentioned in the SNMP Agent and MIB section of the guide, AMon includes an embedded SNMP v2 Agent and corresponding MIB. Together, the agent and MIB are used by SNMP-capable network management applications to monitor the agent's corresponding ActiveMQ message broker. By default, the agent automatically generates or produces SNMP notifications (a.k.a., traps) when the ActiveMQ message broker starts and stops (see SNMP Notifications). This section of the guide describes how Monlets can also generate SNMP notifications. To send or produce SNMP notifications, the Monlets utilize a Camel endpoint called, "snmp:trap", which is provided by the AMon product. This endpoint is essentially an interface that the SNMP agent provides, via Camel, that allows the Monlets to produce SNMP notifications.

There are two types of notifications that the Monlet can produce: *LoggingNotification* and *MonitorNotification*. The following sections describe their makeup and provide more information on how the Monlets produce the notification via the "snmp:trap" endpoint.

7.1 SNMP LoggingNotification

This notification is issued by Monlets to notify SNMP management applications that the ActiveMQ message broker has logged a message with a severity of WARN or higher. To achieve this, the Monlet subscribes to the LOG_ENTRY AMon event message and simply forwards, via the "snmp:trap" Camel endpoint, those event messages to the SNMP agent. See the Log4J Monitoring section of the guide for more information regarding the LOG_ENTRY event and how to set up AMon to generate these event types.

The following sample Monlet illustrates how simple it is to do this; please refer to the comments in the example for additional information.

```
package com.ttm.activemq.monitor.monlets;

import com.ttm.activemq.monitor.Monlet;
import com.ttm.activemq.monitor.event.BaseEvent;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
/**
 *
 *  This Monlet subscribes to LOG_ENTRY event messages and pipes the received
 *  messages to the SNMP agent via the "snmp:trap" endpoint/component.  Upon
 *  receiving a LOG_ENTRY message,  the SNMP agent will generate a
 *  LoggingNotification trap that includes information taken from the LOG_ENTRY
 *  event message.
 *
 */
public class LogMonlet extends Monlet {

  @Override
  public void configure() {

    // Capture AMon event message types and place them in the asynchronous seda
    // queue
    monitor().to("seda:LogMonlet");

    // Pipe the LOG_ENTRY event messages to the SNMP Agent who will extract
    // information from the LoggingEvent object for the correspoding LoggingNotification
    // notification
    from("seda:LogMonlet").
      choice().
        when().language(MEL, "${eventType == LOG_ENTRY}").
            to("snmp:trap").
      end();

  }
}
```

That's really all there is to a Monlet that produces a LoggingNotification. It just forwards or pipes the LOG_ENTRY event message to the SNMP agent via the "snmp:trap" Camel endpoint. Upon receiving the LOG_ENTRY event message, the agent uses the event message's reference to the LoggingEvent object to extract all relative information that is used to build the corresponding SNMP notification.

The following block of ASN.1 notation, which is taken from AMon's SNMP MIB, defines the SNMP LoggingNotification. It gives you an idea of the information that is carried in the LoggingNotification, and as you can see, a lot of the trap* attributes correlate back to the LoggingEvent object.

```
amqLoggingNotification NOTIFICATION-TYPE
    OBJECTS {
            trapDateTime,         -- date and time stamp of the notification
            trapBrokerName,       -- the broker's name
            trapBrokerId,         -- broker id
            trapLogClassName,     -- the class that generated the log entry
            trapLogFileName,      -- the classes file name
            trapLogLineNumber,    -- the line number w/in the file from which the log entry
                                  -- was generated
            trapLogMethodName,    -- the name of the method that generated the log entry
            trapLogThreadName,    -- the name of thread the generated the log entry
            trapLogTimeStamp,     -- the time stamp associated with the log entry
            trapLogMessage,       -- the log entry's message
            trapLogLevel,         -- the severity (i.e,. WARN, ERROR, or FATAL)
            trapMachineName       -- the name of the machine from which the log entry
                                  -- was generated
        }
    STATUS          current
    DESCRIPTION     "Used for notifying that a message has been logged
            to the AMQ log file"
    ::= { amqTraps 1 }
```

The SNMP agent assigns values to the trap* attributes prior to sending the notification. The only thing the Monlet has to do is forward the LOG_ENTRY event message to the agent.

The SNMP Notifications section of the guide shows you how to configure, via the AMon Properties File, the hosts that are to receive the notifications. However, your Monlet can specify its own list of hosts in addition to the ones listed in the properties file. Below is the LogMonlet again, but this time it is specifying, via a Map object, a community called "TTMLocations" that contains two hosts called, "DIEGO" and "TAMPA"[4]. When the SNMP agent receives the LOG_ENTRY event message, it will check if the Monlet has specified such a Map object in the Camel Exchange object. If the object exists in the Exchange, the agent will send the notification to the hosts that are specified in the Map object.

Note how the Monlet uses the *Monlet.setSNMPCommunityMap()* static method to nail the Map object to the Camel Exchange object prior to forwarding the LOG_ENTRY event message to the agent.

```
package com.ttm.activemq.monitor.monlets;
```

[4] IP addresses like "123.45.67.89" are also permitted

Copyright 2009, Total Transaction Management

```
import java.util.HashMap;
import java.util.List;
import java.util.Vector;

import com.ttm.activemq.monitor.Monlet;
import com.ttm.activemq.monitor.event.BaseEvent;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;
/**
 *
 *  This Monlet subscribes to LOG_ENTRY event messages and pipes the received
 *  messages to the SNMP agent via the "snmp:trap" endpoint/component.  Upon
 *  receiving a LOG_ENTRY message,  the SNMP agent will generate a
 *  LoggingNotification trap that includes information taken from the LOG_ENTRY
 *  event message.
 *
 */
public class LogMonlet extends Monlet {

  @Override
  public void configure() {

    // Capture AMon event message types and place them in the asynchronous seda
    // queue
    monitor().to("seda:LogMonlet");

    // Pipe the LOG_ENTRY event messages to the SNMP Agent who will extract
    // information from the LoggingEvent object for the correspoding LoggingNotification
    // notification
    from("seda:LogMonlet").
      choice().

        when().language(MEL, "${eventType == LOG_ENTRY}").
          process(new Processor() {
            public void process(Exchange exchange) throws Exception {

              // Create a Map containing the communities and host(s) at those communities
              // that we would like to send the trap to. This Map will be merged with the
              // community and host information (if any) that is located in the  AMon
             //  properties file.
             HashMap<String, List> hostMap = new HashMap<String,List>();
             Vector<String> hosts = new Vector<String>();
             hosts.add("DIEGO");
             hosts.add("TAMPA");
             hostMap.put("TTMLocations", hosts);
```

```
            //  Use this static method to pass the hostMap to the SNMP agent
            Monlet.setSnmpCommunityMap(exchange, hostMap);
          }
        }). to("snmp:trap").
      end();

  }
}
```

7.2 SNMP MonitorNotification

The MonitorNotification is the second type of SNMP notification that a Monlet can generate, and it is used to send general monitoring information to the SNMP management applications. For example, a Monlet can send this notification if a particular resource's threshold has been breached, a destination (queue or topic) no longer has consumers, or a particular client application has connected with the message broker.

Here is the ASN.1 notation for the MonitorNotification

```
amqMonitorNotification NOTIFICATION-TYPE
      OBJECTS {
                  trapBrokerName,
                  trapBrokerId,
                  trapDateTime,
                  trapMonitorPropertyName,
                  trapMonitorResourceName,
                  trapMonitorPropertyValue,
                  trapMonitorMessage,
                  trapMachineName
            }
      STATUS          current
      DESCRIPTION        "General purpose monitoring notification. Typical
                     in response to a gauge or counter breaching a
                     threshold"
      ::= { amqTraps 4 }
```

The SNMP agent will assign values to the trap* attributes. However, there are a handful of attributes whose values should be specified by the Monlet before it forwards the event message to the SNMP agent. These are the attributes with the "trapMonitor" prefix. The Monlet super class provides a handful of static convenience methods that the Monlets use to specify the values for the trapMonitor* attributes. The following lists and describes all the Monlet setSnmp* static methods.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

```
  /**
   *  The monlet uses this method to specify the name of the property that triggered the
   *  MonitorNotification.  For example, the name can be a threshold like
   *   'maxNumberOfMessages', 'maxNumberOfConnections', or 'minNumberofThreads', etc.
   */
  public static void setSnmpTrapMonitorPropertyName(Exchange exchange,
        String propertyName);

 /**
   *  The monlet uses this method to specify the the ASCII string representation
   *  of the value associated with the property name that triggered this
   *  notification. For example, this can be the value of a threshold that was
   *  breached and caused the triggering of the MonitorNotification.
   */
  public static void setSnmpTrapMonitorPropertyValue(Exchange exchange,
        String propertyValue);

  /**
   * The monlet uses this method to associate a message with a monitor
   * notification.
   */
  public static void setSnmpTrapMonitorMessage(Exchange exchange,
        String message);


  /**
   * The monlet uses this method to specify the name of the resource
   * associated with the triggering of the MonitorNotification (trap). For
   * example, the name of a queue or topic.
   */
  public static void setSnmpTrapMonitorResourceName(Exchange exchange,
        String resourceName);

/**
   * The monlet uses this static method to nail its snmp community-host
   *  map to the exchange; this will then allow the snmp agent to access
   *  the optional map
   */
  public static void setSnmpCommunityMap(Exchange exchange, Map communityMap);
```

The Monlet must set at least one trapMonitor* attribute. If not, the following exception is thrown.

```
com.ttm.activemq.snmp.camel.SnmpProducer.process(): Insufficient information to generate
meaningful snmp notification
```

Copyright 2009, Total Transaction Management

What follows is an example of a Monlet that sends a MonitorNotification whenever it receives any event message type. Prior to forwarding the message to the SNMP agent, it invokes the Monlet.setSnmp* static methods to set the corresponding MonitorNotification attributes. Please note that this is just an example of how to invoke these static methods.

```
package com.ttm.activemq.monitor.monlets;

import java.util.HashMap;
import java.util.List;
import java.util.Vector;

import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

public class SnmpMonlet extends Monlet {

   public void configure() {

      // Subscribe to AMon events and forward to seda queue so that the event can be processed
      // asynchronously
      monitor().to("seda:snmp");

       from("seda:snmp").
            process(new Processor() {
              public void process(Exchange exchange) throws Exception {

              // do some sort of processing here
             Monlet.setSnmpTrapMonitorPropertyName(exchange,"somePropertyName");
             Monlet.setSnmpTrapMonitorPropertyValue(exchange,"1234");
             Monlet.setSnmpTrapMonitorResourceName(exchange,"someResourceName");
             Monlet.setSnmpTrapMonitorMessage(exchange,
                   "Hello World From The SnmpMonlet");
            }
        }).to("snmp:trap");
   }
}
```

8 Installing AMon

8.1 Installing the AMon Files

Installing AMon is a relatively easy process comprising these steps.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

1. Extract the three files called "monitor-<*version*>.jar", "juel-<*version*>.jar", and "jdmkrt.jar" from the "AMon-<*version*>.zip" or .gzip file, and copy them into the "ACTIVEMQ_HOME/lib/optional" directory.

2. After you have developed one or more Monlets, place the jar file(s) that contain your Monlet class files into the "ACTIVEMQ_HOME/lib/optional" directory.

3. Extract the file called, "monitor.properties" from the AMon<version>.zip or .gzip file into the "ACTIVEMQ_HOME/conf" directory.

4. When you receive your AMon license file, place it in the "ACTIVEMQ_HOME/conf" directory. The license file name is either "amonlicense" or "amonlicense.dat". If you are running an embedded broker (i.e., your application and broker are running w/in the same VM), make sure to place the license file in the broker's classpath.

That's it! Installation is now complete; now we need to configure the ActiveMQ message broker to load AMon and your Monlets on startup, which is all explained in the next section.

8.2 Loading AMon into the ActiveMQ Message Broker

To have the ActiveMQ message broker load the AMon framework on startup, you must define an AMon-specific plugin bean within the message broker's Spring XML configuration file (by default ACTIVEMQ_HOME/conf/activemq.xml). There are two options you can take, with the first being depicted in the XML listing below, which is a snippet of an ActiveMQ message broker's XML configuration file. Note how the AMon bean, with an id of "monitor", is defined outside the <broker> element and is referenced via the <broker> element's "plugins" attribute.

Option #1

```
<!--                    ActiveMQ Broker Configuration File   -->
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:amq="http://activemq.org/config/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
  http://activemq.apache.org/camel/schema/spring
  http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

  <bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

  <broker xmlns="http://activemq.org/config/1.0" brokerName="secprototype"
       dataDirectory="${activemq.base}/data" plugins="#monitor">
           …
       <transportConnectors>
```

Copyright 2009, Total Transaction Management

```
        <transportConnector name="openwire"  uri="tcp://localhost:61616" />
    </transportConnectors>
        …
 </broker>
 <bean id="monitor" class="com.ttm.activemq.monitor. MonitorPlugin"/>
</beans>
```

That's it! With those two minor additions to the message broker's XML configuration file, you've configured the message broker to load and start AMon. The Configuring AMon section of this document will show you how to configure AMon and the Monlets that it hosts.

The second option is depicted in the XML snippet below. With this approach, note how the "monitor" bean is defined within the <broker> element's <plugins> sub-element and that it is no longer referenced via the broker element's "plugins" attribute. Also note how you must specify a name space (i.e., xmlns="http://www.springframework.org/schema/beans") for the AMon or monitor bean.

Option #2

```
<!--                          ActiveMQ Broker Configuration File   -->

<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:amq="http://activemq.org/config/1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
 http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
 http://activemq.apache.org/camel/schema/spring
 http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

 <broker xmlns="http://activemq.org/config/1.0" brokerName="broker1"
    dataDirectory="${activemq.base}/data"">

    <transportConnectors>
        <transportConnector name="openwire"  uri="tcp://localhost:61616" />
    </transportConnectors>

    <plugins>
      <bean id="monitor"
         class=" com.ttm.activemq.monitor. MonitorPlugin"
         xmlns="http://www.springframework.org/schema/beans">
    </plugins>

 </broker>

</beans>
```

When the ActiveMQ message broker successfully loads and starts the ActiveMQ message broker, you will see the following informational messages on the broker's system console.

```
INFO  MonitorPlugin                - AMon License Key Is Valid
INFO  MonitorPlugin                - Starting SNMP
INFO  MonitorPlugin                - Started
```

You will not see the "Starting SNMP" message if you've disabled the SNMP agent. You will instead see the following message

```
INFO  MonitorPlugin                - SNMP has been disabled
```

If your AMon license file is not valid or has not been installed, then you will see a corresponding error message with the "ERROR" prefix. For example, if AMon could not find the license file it will put out this ERROR message to the console.

```
ERROR MonitorPlugin                - Unable to locate AMon license file
```

If the license file has expired, AMon will put the following ERROR message out to the console.

```
ERROR MonitorPlugin                - AMon License Key has expired
```

If the license key file could not be found or has expired, the ActiveMQ message broker will load AMon, but it will not be started.

8.3 Loading Your Monlets into the ActiveMQ Message Broker

The final step in the installation process is to configure the ActiveMQ message broker to load your Monlets when it starts up. To do this, all you have to do is define a Camel context within the ActiveMQ message broker's XML configuration file and assign it the one or more packages that contain your Monlets. As an example, take a look at the following ActiveMQ message broker's XML configuration file. Note in bold how a Camel context and its packages have been defined.

```
<!--                    ActiveMQ Broker Configuration File   -->
<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:amq="http://activemq.org/config/1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
 http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
 http://activemq.apache.org/camel/schema/spring
 http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">
```

```
<!-- Allows us to use system properties as variables in this configuration file -->
 <bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

<broker xmlns="http://activemq.org/config/1.0" brokerName="broker1"
    dataDirectory="${activemq.base}/data" plugins="#monitor">
        …
    <transportConnectors>
        <transportConnector name="openwire"  uri="tcp://localhost:61616" />
    </transportConnectors>
        …
</broker>

<bean id="monitor" class="com.ttm.activemq.monitor. MonitorPlugin"/>

 <camelContext id="AmonCamel" xmlns="http://activemq.apache.org/camel/schema/spring">
     <package>com.acme.monlets</package>
 </camelContext>

</beans>
```

That's all it takes! When the broker starts, your Monlet class files are loaded by Camel and your Monlets are ready to go.

You can also define more than one Camel context (just make sure to give them different bean ids), each with one or more Monlet packages. Just remember that Monlets cannot communicate with one another if they are in different Camel contexts.

If you also happen to have non-AMon related Camel contexts defined in your activemq.xml file, make sure the AMon Camel contexts are defined after all non-AMon Camel contexts (see the example below). Not doing so will result in spurious events being generated.

```
<!--                    ActiveMQ Broker Configuration File   -->
<beans
 xmlns="http://www.springframework.org/schema/beans"
 xmlns:amq="http://activemq.org/config/1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
 http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
 http://activemq.apache.org/camel/schema/spring
 http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

 <!-- Allows us to use system properties as variables in this configuration file -->
  <bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

 <broker xmlns="http://activemq.org/config/1.0" brokerName="broker1"
     dataDirectory="${activemq.base}/data" plugins="#monitor">
         …
```

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

```
    <transportConnectors>
        <transportConnector name="openwire"  uri="tcp://localhost:61616" />
    </transportConnectors>
        …
 </broker>

 <bean id="monitor" class="com.ttm.activemq.monitor. MonitorPlugin"/>

 <!-- Ensure that non-AMon camelContexts are always defined prior to AMon-related
      camelContexts -->

  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
      <package>org.foo.bar</package>
      <route>
         <from uri="activemq:example.A"/>
         <to uri="activemq:example.B"/>
      </route>
   </camelContext>

  <camelContext id=" AmonCamel " xmlns="http://activemq.apache.org/camel/schema/spring">
      <package>com.acme.monlets</package>
  </camelContext>

</beans>
```

9 Configuring AMon

So now that you've learned how to install/load AMon and the Monlets, it is time to discuss how to configure AMon in general. This section of the guide describes how to configure various AMon components.

9.1 Properties File

AMon and the Monlets that it hosts are primarily configured and managed through an external properties file. By default, this properties file is called "monitor.properties". You can place the file anywhere in the ActiveMQ broker's default class path (ACTIVEMQ_CLASSPATH), but it is best to place it in the ACTIVEMQ_HOME /conf directory. If you would like to place the file outside the ActiveMQ class path or change its name, use the MonitorPlugin's 'propertiesFile' bean property to specify the new location and/or name. Below is a sample message broker XML configuration file that illustrates the use of the 'propertiesFile' property (note the text in **bold**).

```
<!--                    ActiveMQ Broker Configuration File   -->
<beans
  xmlns="http://www.springframework.org/schema/beans"
```

Copyright 2009, Total Transaction Management

```
 xmlns:amq="http://activemq.org/config/1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
 http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
 http://activemq.apache.org/camel/schema/spring
 http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

  <bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

 <broker xmlns="http://activemq.org/config/1.0" brokerName="secprototype"
     dataDirectory="${activemq.base}/data" plugins="#monitor">
          …
 </broker>

 <bean id="monitor" class="com.ttm.activemq.monitor. MonitorPlugin">
    <property name="propertiesFile" value="c:\temp\myMonitor.properties"/>
 </bean>

  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
       <package>com.acme.monlets</package>
  </camelContext>

</beans>
```

Through the monitor properties file you:

- Control the event message types that AMon will publish. Recall that by default the publishing of all event messages is disabled.
- Define Monlet-specific properties.
- Define SNMP-specific properties

There are two types of properties that can be defined in the properties file: Monitor and Monlet-specific properties. Let’s first discuss Monitor-specific properties. A property name that follows the “monitor.<*event type*>” format is used to designate a Boolean property that enables/disables the publishing of the corresponding event message type. The <*event type*> portion of the property name must match one of the event types listed in the event types table (see Event Types as Constants), but it must be in lower case. For example, let’s suppose that you’ve developed one or more Monlets that would like to receive ADD_CONNECTION and REMOVE_CONNECTION event messages. To enable the publishing of these two event message types, define the two properties called “monitor.add_connection” and “monitor.remove_connection” and assign them both a value of “true” as this simple monitor properties file example illustrates.

```
# Enable the publishing of the ADD_CONNECTION and REMOVE_CONNECTION
# event message types.
```

```
monitor.add_connection=true
monitor.remove_connection=true
```

Please note that the entire property name must be in lower case.

Properties with the "monitor." prefix are reserved for the AMon product.

As mentioned in a previous section, by default AMon will not publish any event messages. You must decide what event types you'd like your Monlets to receive and then enable the publishing of the corresponding event messages. Use caution when choosing what events to enable, because enabling too many or all events will result in an overall performance degradation. In particular, use caution when enabling what can easily become very "high volume" events such as MSG_DELIVERED and MSG_CONSUMED.

Now let's discuss Monlet-specific properties, which can be assigned any name as long as they don't conflict with the previously described Monitor-specific properties. A Monlet-specific property is one that can be referenced within a MEL expression statement. For example, let's suppose that you've got the properties file defined below. It consists of two Monitor-specific properties and two Monlet-specific properties called, "myBrokerName" and "maxThreads".

```
# Enable the publishing of the ADD_CONNECTION and REMOVE_CONNECTION
# event message types.

monitor.add_connection=true
monitor.remove_connection=true

# Monlet-specific properties

myBrokerName=fred
maxThreads=100
```

The "myBrokerName" property can be referenced from within a MEL expression as follows.

```
language(MEL,"${broker.brokerName == props.myBrokerName}")
```

Let's take a closer look at what's happening within the above MEL expression. The expression is testing whether the ActiveMQ broker's name matches the value assigned to the "myBrokerName" property. The "broker.brokerName" portion of the expression is using the MEL mapped 'broker' variable to reference the ActiveMQ Broker object's getBrokerName() method and thus acquire the name of the hosting message broker. The "props.myBrokerName" portion of the expression is using the MEL mapped 'props' variable to reference a Map object that contains all of the properties defined in the properties file. And in this particular case, the "myBrokerName" property is being referenced.

Copyright 2009, Total Transaction Management

Let's take a look at another example.

```
language(MEL,"${vm.totalStartedThreadCount >= props.maxThreads}")
```

In the above example, the MEL expression is testing whether the hosting virtual machine has started more than 100 threads since it was first launched. In this case, the left hand portion of the expression is using the 'vm' MEL variable to reference the VirtualMachine Object and its getTotalStartedThreadCount() method. And as in the previous example, the right hand portion of the expression is once again using the MEL mapped 'props' variable to reference the Map object that contains all of the properties defined in the properties file. However, in this case the String value of the 'maxThreads' properties is being coerced to a type that matches the return value of the getTotalStartedThreadCount() method.

You can also use the UEL's bracket notation to access the properties. This comes in handy if you need to access a property name that includes the dot notation. For example, let's suppose you wanted to access one of the monitor-specific properties (e.g., monitor.timer_interval) and you used the dot notation, as this example illustrates.

```
language(MEL,"${props.monitor.timer_interval >= 5000}")
```

The above statement results in an error, because the MEL attempts to access the props object's 'monitor' property and then that object's 'timer_interval' property; both of which do not exist. As an alternative, you can use the bracket notation to access the property, as illustrated in this example.

```
language(MEL,"${props['monitor.timer_interval'] >= 5000}")
```

If you make any changes to the properties file, while the ActiveMQ message broker is running, AMon automatically detects that the file has been modified, reloads the file, and the new property values take effect immediately. In this way, AMon allows you to dynamically re-configure your Monlets.

Your Monlet can also access the properties from within its Processor object's 'process' method. Below is a snippet of Monlet code that illustrates how the Monlet is iterating through the properties from within the process method.

```
package com.ttm.activemq.monitor.monlets;

import java.util.Properties;
import java.util.Iterator;
import com.ttm.activemq.monitor.event.BaseEvent;
import com.ttm.activemq.monitor.Monlet;
import org.apache.camel.Processor;
import org.apache.camel.Exchange;

public class PropertiesMonlet extends Monlet {
```

Copyright 2009, Total Transaction Management

```
  public void configure() {

      // Subscribe to AMon events and forward to seda queue so that the
      // event can be processed asynchronously
      monitor().to("seda:PropertiesMonlet");

      from("seda:PropertiesMonlet").
            process(new Processor() {

         public void process(Exchange exchange) throws Exception {
            // Access the message's body, cast it to BaseEvent, and then use BaseEvent
            // to get the properties and iterate through them
            BaseEvent baseEvent = (BaseEvent) exchange.getIn().getBody();
            Properties props = baseEvent.getMonitorProperties();
            for (Iterator i = props.keySet().iterator(); i.hasNext();) {
               String key = (String) i.next();
               System.out.println("PropertiesMonlet: Key = " + key +
                     " Value = " + props.get(key).toString());
            }
         }
      });

  }
}
```

Note how the Monlet uses the Camel exchange object to get the message body. All the event messages that the AMon core plugin component publishes to the Monlets are of type com.ttm.activemq.monitor.event.BaseEvent. Therefore, your Monlet has access to the BaseEvent object and its 'get' methods. The BaseEvent's getMonitorProperties() method returns a copy of the AMon properties object. Another BaseEvent method that Monlet developers will find interesting is the getEventType(), which returns the String representation of the event type.

9.2 JMX Client

You can also update the properties through a JMX client like the JConsole. AMon includes JMX MBeans through which you can manage AMon (see The Management Console). The MBeans allow you to invoke the following properties-related operations:

- **View Properties**: this operation will display the current list of properties and their values.
- **Put Property**: this operation allows you to add a new property or change the value of an existing property
- **Remove Property**: this operation allows you to remove an existing property. Please note that removing a Monitor-specific property sets the value of that property to its default setting.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

- **Save Properties**: this operation will write out the current set of properties and their values to the external properties file. So if you've updated the properties, through the JMX client, you can use this operation to ensure that your updates will remain valid the next time the message broker is started.

9.3 Log4J Monitoring

AMon can be configured to monitor messages that are written to the ActiveMQ log file, which is typically called "activemq.log" and located in "ACTIVEMQ_HOME/data/<*broker name*>".

Only messages with the following log4j priorities or levels will be monitored: WARN, ERROR, and FATAL. Also, AMon will only monitor those messages that are generated by object instances that belong to the "org.apache.activemq" package.

To enable the monitoring of log4j messages, you must edit the log4j.properties file, which is located in "ACTIVEMQ_HOME/conf, as follows.

1. Append the ", monitor" string to this line, which should be towards the top of the file.

```
log4j.rootLogger=INFO, stdout, out
```

The above line must end up looking like this.

```
log4j.rootLogger=INFO, stdout, out, monitor
```

2. Add the following lines to the end of the file.

```
# Monitor appender
log4j.appender.monitor=com.ttm.activemq.monitor.Appender
log4j.appender.monitor.layout=org.apache.log4j.PatternLayout
log4j.appender.monitor.layout.ConversionPattern=%d [%-15.15t] %-5p %-30.30c{1} - %m%n
```

You must then make sure to set the monitor.log_entry property in the Properties File to true like so.

```
# Enable the publishing of the LOG_ENTRY event message type

monitor.log_entry=true
```

That's it! With those two simple edits, AMon will monitor ActiveMQ's log file and publish the LOG_ENTRY event message type.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

9.4 SNMP Agent

This section describes how to configure AMon's embedded SNMP v2 agent. The following table lists and describes all of the agent's configuration properties that may be specified through ActiveMQ's XML configuration file.

Variable Name	Default Value	Description
snmpCommunity	activemq	The community name used for sending SNMP notifications to SNMP network management applications.
snmpHostName	localhost or 127.0.0.1	This is the hostname or IP address that the agent uses for communicating with the SNMP management applications and sending notifications to the management applications. In other words, this is the IP address that the agent will listen for SNMP get requests from the management applications. Management applications must send SNMP requests to this hostname or IP address. This also comprises the default list of target hostnames or IP addresses that the agent uses for sending notifications (traps). Please see SNMP Notifications for details on how to add hostnames to the list.
snmpMaxActiveClientCount	10	The maximum number of network management applications that the SNMP agent can communicate with.
snmpPort	8085	This is the port number that the agent uses for communicating with the management applications. Management applications must send SNMP requests to this port.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

snmpTrapPort	snmpPort+1	This is the target port number that the agent uses for sending SNMP notifications (traps). SNMP management applications should listen on this port number for the agent's notifications.
useSnmp	true	Set to "false" to disable the agent.

Table 3 SNMP Properties

The above properties are *static,* because they cannot be modified while the message broker is running; however, the following section will show you how you can override two of these static properties with their *dynamic* counterparts.

The following is an example ActiveMQ broker configuration file that illustrates how to set the properties listed in the table above.

```
<!--                    ActiveMQ Broker Configuration File   -->
<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:amq="http://activemq.org/config/1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://activemq.org/config/1.0 http://activemq.apache.org/schema/activemq-core.xsd
  http://activemq.apache.org/camel/schema/spring
  http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

   <bean  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" />

  <broker xmlns="http://activemq.org/config/1.0" brokerName="secprototype"
       dataDirectory="${activemq.base}/data" plugins="#monitor">
           …
  </broker>

  <bean id="monitor" class="com.ttm.activemq.monitor. MonitorPlugin">
     <property name="propertiesFile" value="c:\temp\myMonitor.properties"/>
     <property name="snmpHostName" value="127.37.78.98"/>
     <property name="snmpPort" value="9000"/>
     <property name="snmpTrapPort" value="9050"/>
     <property name="snmpCommunity" value="myCommunity"/>
     <property name="snmpMaxActiveClientCount" value="50"/>
  </bean>
```

Copyright 2009, Total Transaction Management

```
  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
      <package>com.acme.monlets</package>
  </camelContext>

</beans>
```

By default, the SNMP agent is enabled.

9.4.1 SNMP Notifications

When enabled, AMon's SNMP agent generates a notification (trap) when the hosting ActiveMQ message broker starts and stops. These notifications include information such as the message broker's name, machine name, IP address, date and time the notification was generated, etc. Please refer to the AMon SNMP MIB for a full description of the notifications and all other SNMP-related attributes.

The following static properties, which were described in the previous section, are used to transmit the notifications: snmpHostName, snmpTrapPort, and snmpCommunity. So by default, the start and stop notifications will only be sent to the "activemq" community on the localhost's 8086 port number. See the SnmpPropertiesTable for details on how to change these static default settings. However, via the AMon Properties File, AMon allows you to override the snmpHostName and snmpCommunity static properties with their dynamic counterparts; the snmpTrapPort does not have a dynamic counterpart. Here's how it is done.

1. Add the "monitor.snmp.trap.communities" property to the properties file and assign it a comma-separated list of community names.
2. For each community name that is assigned to the "monitor.snmp.trap.communities" property, add a "monitor.snmp.trap.<*community name*>.hosts" property to the properties file and assign the property a list of host names or IP addresses.

So by adding those properties to the AMon properties file, you identify the communities and their host names or IP addresses that will be sent notifications.

Here's an example properties file

```
monitor.snmp.trap.communities= comm1, comm2, comm3
monitor.snmp.trap.comm1.hosts= host1, host2, host3
monitor.snmp.trap.comm2.hosts= host4, host5, host6
monitor.snmp.trap.comm3.hosts= 123.56.45.67, host8, host9
```

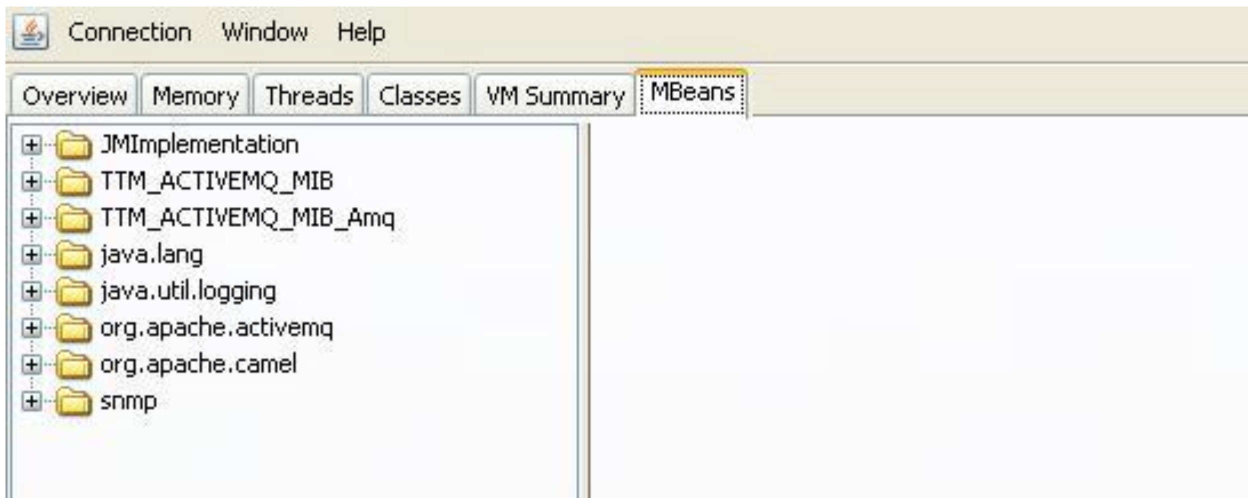
Please note that specifying the properties in the Properties File will override the default static values (i.e., community of "activemq" with a host of "localhost"); therefore, if you would like to continue to include the default values, you will need to specify them in the list.

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

10 The Management Console

AMon adheres to the JMX standard and as such, can be managed via any JMX client; the most popular of which is the JConsole. Along with managing properties, as described in the previous section, you can also 'stop' and 'start' AMon through the JMX client. Stopping AMon disables the publishing of all event message types. Starting AMon re-enables the publishing of AMon event messages types as dictated by the current set of Monitor-specific properties (see previous section).

If the SNMP agent has been enabled, you will see references to AMon's SNMP MIB MBeans in the JConsole's MBean tab screen (see picture below). These MBeans are called, "TTM_ACTIVEMQ_MIB" and "TTM_ACTIVEMQ_MIB_Amq". ***Please note that the 'operations' associated with these MBeans are not implemented and thus you will get an error screen if and when you invoke any of these MBeans' operations.***



11 VirtualMachine Object

This section lists and describes the get methods that are made available through the 'vm' MEL variable. This variable is a reference to an AMon object that is called VirtualMachine or VM for short. The purpose of this object is to give the Monlet developer access to real-time information for the hosting Java virtual machine.

```
// Returns the current number of live daemon threads.

public int getDaemonThreadCount()
```

```
// Returns the total number of threads created and also started since the
// Java virtual machine started.

public long getTotalStartedThreadCount()

// Returns the peak live thread count since the Java virtual machine started
// or peak was reset.

public int getPeakThreadCount()

// Returns the current number of live threads including both daemon and
// non-daemon threads.

  public int getThreadCount()

  /*************************************************************************
   * Returns the current memory usage of the heap that is used for object
   * allocation. The heap consists of one or more memory pools. The used and
   * committed size of the returned memory usage is the sum of those values of
   * all heap memory pools whereas the init and max size of the returned memory
   * usage represents the setting of the heap memory which may not be the sum
   * of those of all heap memory pools. The amount of used memory in the
   * returned memory usage is the amount of memory occupied by both live
   * objects and garbage objects that have not been collected, if any.
   *************************************************************************/
   // Returns the amount of used memory in bytes.

public long getUsedHeapMemory()

   // Returns the maximum amount of memory in bytes that can be used for memory
   // management. This method returns -1 if the maximum memory size is
   // undefined. This amount of memory is not guaranteed to be available for
   // memory management if it is greater than the amount of committed memory.
   // The Java virtual machine may fail to allocate memory even if the amount
   // of used memory does not exceed this maximum size.

   public long getMaxHeapMemory()

   // Returns the amount of memory in bytes that is committed for the Java
   // virtual machine to use. This amount of memory is guaranteed for the Java
   // virtual machine to use.

   public long getCommittedHeapMemory()

   // Returns the amount of memory in bytes that the Java virtual machine
   // initially requests from the operating system for memory management. This
   // method returns -1 if the initial memory size is undefined.
```

```
  public long getInitHeapMemory()


  /*****************************************************************************
  * Same as above, but for non-heap usage.

*****************************************************************************/
  // Returns the amount of used memory in bytes.

  public long getUsedNonHeapMemory()

  // Returns the maximum amount of memory in bytes that can be used for memory
  // management. This method returns -1 if the maximum memory size is
  // undefined. This amount of memory is not guaranteed to be available for
  // memory management if it is greater than the amount of committed memory.
  // The Java virtual machine may fail to allocate memory even if the amount
  // of used memory does not exceed this maximum size.

  public long getMaxNonHeapMemory()

  // Returns the amount of memory in bytes that is committed for the Java
  // virtual machine to use. This amount of memory is guaranteed for the Java
  // virtual machine to use.

  public long getCommittedNonHeapMemory()

  // Returns the amount of memory in bytes that the Java virtual machine
  // initially requests from the operating system for memory management. This
  // method returns -1 if the initial memory size is undefined.

public long getInitNonHeapMemory()

  // Returns the uptime of the Java virtual machine in milliseconds.

  public long getUptime()

  // Returns the start time of the Java virtual machine in milliseconds. This
  // method returns the approximate time when the Java virtual machine started.
  public long getStartTime()

  // Returns the amount of free memory in the Java Virtual Machine.
  // Calling the gc method may result in increasing the value returned by
  // freeMemory.

 public long getFreeMemory()
```

Copyright 2009, Total Transaction Management

```
// Returns the total amount of memory in the Java virtual machine. The value
// returned by this method may vary over time, depending on the host
// environment. Note that the amount of memory required to hold an object
// of any given type may be implementation-dependent.

public long getTotalMemory()

// Returns the maximum amount of memory that the Java virtual machine will
// attempt to use. If there is no inherent limit then the value
// Long.MAX_VALUE will be returned.

public long getMaxMemory()

// Return the version of the spec that the current VM adheres to.
// For example, it may return the string, “1.5”

public double getSpecificationVersion()

// Returns the Java virtual machine specification name. This method is
// equivalent to System.getProperty("java.vm.specification.name").

public String getSpecName()

// Returns the Java virtual machine specification vendor. This method is
// equivalent to System.getProperty("java.vm.specification.vendor").

public String getSpecVendor()

// Returns the Java virtual machine specification version. This method is
// equivalent to System.getProperty("java.vm.specification.version").

public String getSpecVersion()

// Returns the operating system name. This method is equivalent to
// System.getProperty("os.name").

public String getOSName()

// Returns the operating system architecture. This method is equivalent to
// System.getProperty("os.arch").

public String getOSArch()

// Returns the operating system version. This method is equivalent to
// System.getProperty("os.version").

public String getOSVersion()
```

Copyright 2009, Total Transaction Management
www.ttmsolutions.com

```
// Returns the Java virtual machine implementation name. This method is
// equivalent to System.getProperty("java.vm.name").

public String getVmName()

// Returns the Java virtual machine implementation vendor. This method is
// equivalent to System.getProperty("java.vm.vendor").

public String getVmVendor()

// Returns the Java virtual machine implementation version. This method is
// equivalent to System.getProperty("java.vm.version").

public String getVmVersion()

// Returns the system load average for the last minute. The system load
// average is the sum of the number of runnable entities queued to the
// available processors and the number of runnable entities running on the
// available processors averaged over a period of time. The way in which the
// load average is calculated is operating system specific but is typically
// a damped time-dependent average.
//
// If the load average is not available, a negative value is returned.
//
// This method is designed to provide a hint about the system load and may
// be queried frequently. The load average may be unavailable on some
// platforms where it is expensive to implement this method.

public double getSystemLoadAverage()
```

Copyright 2009, Total Transaction Management