Abstract
This tool manual describes a software library for gathering performance data from software systems implemented in the Java programming language. The manual provides overview of the key concepts, basic library API, and instrumentation methods that can be used to instrument applications in order to obtain performance data during application execution.

Keywords: performance measurement, application instrumentation
Revision history

<table>
<thead>
<tr>
<th>Version</th>
<th>Change date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>01.12.2011</td>
<td>Lubomír Bulej</td>
<td>Initial version</td>
</tr>
<tr>
<td>0.2</td>
<td>17.12.2011</td>
<td>Lukáš Marek</td>
<td>Automatic instrumentation, simple data analysis</td>
</tr>
<tr>
<td>0.9</td>
<td>17.12.2011</td>
<td>Lubomír Bulej</td>
<td>Release candidate</td>
</tr>
</tbody>
</table>
1 Tool Description

1.1 Purpose of the tool
The main purpose of the JPMF library is to simplify the process of obtaining performance data from running applications. Instead of having to develop a performance measurement harness for each application, the library provides a generic interface that allows the user to define performance event sources that emit performance events related to application execution. The user then configures what performance are to be collected with particular performance events and the JPMF library takes care of the actual collection and storage of the data in an efficient fashion. The user of the library thus only needs to concern herself with application instrumentation, which can be performed either manually at the source code level, or automatically at the Java byte code level, using tools provided by the library.

1.2 Tool relationship with the Q-ImPrEss workflow
Technically, the JPMF library is a standalone tool, independent of other tools found in the Q-ImPrEss tool chain in the sense that it neither processes input from other tools, nor provides directly consumable output to other tools. However, it is meant to be used for measurement purposes in the Q-ImPrEss Overall Process shown in Figure 1. The need to conduct performance measurements can arise either during the model validation phase, when a model needs to be validated against implementation, or during performance model design..
and performance prediction phases, when some model parameters need to be determined through measurement or when a system usage profile needs to be determined.

1.3  Key concepts
The JPMF library aims to provide a generic framework for collecting performance data from Java applications, without the need to tediously create a measurement infrastructure from scratch whenever there is a need to obtain performance data from some application. To this end, the JPMF library operates with several generic concepts that should be understood prior to using the library. However, since the purpose of this manual is to provide basic overview of the operation and usage of the library, many technical details are omitted for clarity and can be found in the library reference documentation.

1.3.1  Performance Events
The principle of the JPMF library operation is based on receiving named performance events from an application under test (AUT) and attaching performance data collected upon performance event reception to records describing the received events. This allows the library
to be used in any context, as long as performance events can be delivered to the library.

User of the library must ensure that the AUT registers performance events needed to collect performance data relevant to the performance metrics that need to be determined. Each performance event must have a unique name, which should be somehow related to the origin of the event in the application. For component-based applications, the architecture of the application provides a natural structure for naming performance events.

To provide support for hierarchical event naming schemes without prescribing a specific naming structure, the library utilizes the `ObjectName` format defined in the Java JMX specification. An `ObjectName` is a string that comprises the following elements, in order:

- The domain.
- A colon (:).
- A list of key/value pairs, separated by a comma.

The boundaries of key/value pair are determined solely by the initial colon and intermediate commas. Whitespace characters have no special meaning, i.e. they are considered to be part of a key or its value. The following example represents an `ObjectName` with two keys, where each of the keys and the value associated with “key1” begin and end with a space:

\[
\text{domain: key1 = value1 , key2 = value2}
\]

This allows grouping related performance events together and simplifies navigation and runtime event activity management (e.g., a group of events can be enabled or disabled using a single command).

Each performance event also has a type, which determines the basic data payload provided to the JPMF runtime and allows the user to use appropriate even type in different contexts. At the moment, the library supports the following performance event types:

- **AtomicEvent**, which is intended for generic standalone events,
- **IntervalEvent**, which is intended for pairs of events delimiting the start and the end of some generic action,
- **LoopEvent**, which is intended for pairs of events delimiting the start and the end of a generic loop in the code, including the number of loop iterations,
- **MethodEvent**, which is intended for pairs of events delimiting the start and the end of a method invocation, including (optionally) the cause for method return.

All performance event types carry a high-resolution time stamp, which allows extracting basic information such as duration of actions, loops, or method invocations from the events alone.

### 1.3.2 Event Sources

Since multiple related performance events will often originate from the same runtime entity (object instance, component, or a class), the JPMF library defines an `EventSource` interface, which represents a management entity for a group of related events. An event source is an application-side entity that provides information on supported events and their types, and control over which performance events should be emitted by the instrumented application entity. During automatic instrumentation, the instrumentation tool will typically create a single `EventSource` instance for each instrumented application entity, such as interface
implementation, and register it with the JPMF library runtime. The JPMF runtime will then query the event source implementation for the supported performance events and their types and configure the event source to emit performance events specified by the user in an external configuration file.

This method of performance event registration is mainly intended for automatic instrumentation tools, because the cost of creating a custom EventSource implementation will be amortized by repeated use of the instrumentation tool.

### 1.3.3 Event Triggers

To simplify usage in situations where manual instrumentation is needed, the JPMF library provides an alternative method to register performance events. Instead of having to implement the EventSource interface, a user can instead request the library to implement the interface internally and provide the user with a simple trigger interface that can be used to send performance event notifications to the JPMF library.

Unlike event source, an event trigger only represents a single performance event and its interface depends on the represented event type. When requesting an event trigger from the JPMF library, a user must specify the desired event trigger interface in addition to the structured performance event name. At the moment, the library supports the following event triggers, directly corresponding to the performance event types mentioned above:

- AtomicEventTrigger
- IntervalEventTrigger
- LoopEventTrigger
- MethodEventTrigger

### 1.4 Basic API Usage

The JPMF library itself is application agnostic, therefore it only provides API for registering performance events that must be used directly by user, or by an automatic instrumentation tool. Figure 2 shows the API provided by the Jpmf class, which serves as an entry point to the library. The first three methods provide control over the initialization and shutdown of the JPMF runtime, while the other two methods allow registration of event sources and construction of event triggers. Using any of the first three methods is optional.

At runtime, the JPMF library provides a singleton implementation, which is initialized in a lazy manner when the first event source or event trigger is registered. Explicit initialization of the JPMF library is only required when the user wants to provide different initialization context represented by the Properties class. The flush() method can be called by the user to force flushing of buffers containing performance data to disk. And finally the shutdown() method can be called when the instrumented application is exiting. The JPMF library registers a shutdown hook with the JVM in order to ensure flushing of data from buffers to disk, but with manually instrumented applications (where the user is expected to be in control of application startup and shutdown), it is recommended to call the shutdown() method explicitly.

The registerEventSource() and createEventTrigger() methods serve for performance event registration. Each of those methods is targeted at different user. The
registerEventSource() method is intended for automatic instrumentation tools, which will automatically create instances of the EventSource interface and register them with the framework. The createEventTrigger() method is intended for manual instrumentation and provides an implementation of a simplified event source that only supports a single performance event. When creating an event trigger, the user has to specify the desired trigger interface that will be used to trigger performance event notifications.

### 1.4.1 Instrumentation using event triggers

When using the JPMF library for manual instrumentation using event triggers, the user is responsible for creating and storing the event trigger instances and invoking their methods that trigger performance event notifications. The semantics of a particular trigger depends entirely on the user and her use of the trigger. When defined as an instance variable, its name should be different for different instances and the events emitted by the trigger are related to a particular instance. When defined as a static class variable, there will only be a single trigger instance per class, and the events emitted by the trigger are related to the class (or all instances of the class).

Currently, the JPMF library supports four basic types of event triggers, with their interfaces shown in Figure 3. The library can be extended to support additional trigger types by implementing and registering trigger providers with the framework (this is currently beyond the scope of this manual, description of the interfaces can be found in the JPMF reference documentation.

The AtomicEventTrigger represents the simplest of performance event triggers. Atomic events are independent and in subsequent analysis, atomic events can be used to determine count, rate, or delay between individual events. The trigger interface only provides a single method, fire(), which delivers an event notification (along with a high-resolution time stamp) to JPMF runtime.

The IntervalEventTrigger represents a pair of events delimiting the start and the end of a generic action or time interval. The interface provides two methods, start() and stop(), which should be used to enclose code representing the action. Both methods deliver an event notification to the JPMF runtime. In addition to time stamp, these two events are also marked as related, so that the duration of the interval can be easily determined.

The MethodEventTrigger also represents a pair of events, this time delimiting the start and the end of method invocation. It is basically a variant of the interval event trigger
intended specifically for instrumenting method invocations. In addition to the enterMethod() and leaveMethod() methods, which should be used to delimit the method invocation, there is an additional method, leaveMethod(MethodExitCause cause), which can be used to report abnormal return from a method, i.e. exception. Like in the case of interval events, the method invocation events are marked as related, so that the duration of method invocation can be easily determined.

The LoopEventTrigger also represents a pair of events, this time delimiting the start and the end of a loop. It is also a specific type of an interval event, intended specifically for instrumenting loops. To this end, the interface provides the enterLoop() method, which marks the start of a loop, and leaveLoop(long loopCount), which marks the end of a loop and allows to associate the number of loop iterations with the event data. Like in the case of interval events, the loop start and end events are marked as related, so that the duration of a loop can be easily determined.

Figure 3: Event trigger interfaces

```java
interface AtomicEventTrigger extends EventTrigger {
    void fire();
}
interface IntervalEventTrigger extends EventTrigger {
    void start();
    void stop();
}
interface MethodEventTrigger extends EventTrigger {
    void enterMethod();
    void leaveMethod();
    void leaveMethod (MethodExitCause cause);
}
interface LoopEventTrigger extends EventTrigger {
    void enterLoop();
    void leaveLoop (long loopCount);
}
```

The LoopEventTrigger also represents a pair of events, this time delimiting the start and the end of a loop. It is also a specific type of an interval event, intended specifically for instrumenting loops. To this end, the interface provides the enterLoop() method, which marks the start of a loop, and leaveLoop(long loopCount), which marks the end of a loop and allows to associate the number of loop iterations with the event data. Like in the case of interval events, the loop start and end events are marked as related, so that the duration of a loop can be easily determined.

Figure 4 shows the usage of atomic event triggers to instrument Java code. There are two triggers, one global for the whole class named __classTrigger, and one local to a particular instance named __instanceTrigger. The global trigger is fired whenever a new instance of the class is created, the instance-local trigger is fired whenever the execution reaches a certain place in the bar() method code. When creating the triggers, the class of the desired trigger type along with a unique trigger name is passed to JPMF, which creates an internal event source implementation (so that the event source can be controlled from outside) for each trigger and returns a façade implementing the trigger interface. The __getInstanceId() method should be an internal method that is able to assign an identifier to the instance-specific trigger. The other trigger types are used in a similar way, the only difference is the interface type passed as an argument to the createEventTrigger() method and the methods invoked on the trigger instance.
1.4.2 Instrumentation using event sources

For bulk instrumentation of multiple methods in a class, the JPMF library provides the `EventSource` service provider interface, which needs to be implemented by the user and be registered within the JPMF runtime. The interface allows JPMF to control multiple related events originating from a single place (e.g. a class).

Figure 5 lists the methods defined by the `EventSource` interface. The first set of methods comprises the `id()` and `events()` methods, which provide information about an event source and the events it supports. Each event has a local identifier which, combined with the global event source identifier, makes up a fully qualified event name. The fully qualified name must be unique and should be related to application architecture. Each event also has an event type, which determines the event delegate interface required by a particular event in order to provide JPMF with event notifications.

The second set of methods comprises the `setTimerCounter(TimerCounter)` and `setEventDelegate(int, EventDelegate)` methods, which the JPMF uses to provide the event source with implementations of interfaces required from the framework. The framework may provide the event source with a common time source in form of a `TimerCounter` interface, which allows the event source to obtain timing information that is compatible with other event sources and the rest of the framework. However, more important
is to provide the event source with an `EventDelegate` interface reference for each event it supports. The `EventDelegate` references will be used to submit event notifications to JPMF.

Finally, the third set of methods comprises the `enable()`, `disable()`, `isEnabled()`, `enableEvent(int)`, `disableEvent(int)`, and `isEventEnabled(int)` methods, which the JPMF uses to control the event notifications emitted by an event source. The `enable()` and `disable()` methods serve as a master switch, which allows JPMF to enable or disable event source operation (and, consequently, allow the event source to e.g. remove instrumentation from the class to reduce passive-state overhead). The `enableEvent(int)` and `disableEvent(int)` methods serve for fine-grained control and allow JPMF to select events for which the event source should emit event notifications.

The `EventSource` interface is to be implemented by performance instrumentation code and allows creating a single event source entity for a set of related events, such as e.g. events corresponding to interface method invocations. However, because of the support for event groups and event source configuration, the interface is relatively complex for casual use, when only a few unrelated performance events need to be defined. In such cases, the manual instrumentation using event triggers should be used instead. For more information on the `EventSource` and related interfaces, please see the reference documentation.

```java
public interface EventSource {
    String id ();
    List <Event> events ();

    final class Event {
        public Event {
            final String localId,
            final Class <? extends EventDelegate> delegateClass
        );
        public String localId ();
        public Class <? extends EventDelegate> delegateClass ();
    }

    void setTimerCounter (TimerCounter timer);
    void setEventDelegate (int eventIndex, EventDelegate delegate);

    void enable ();
    void disable ();
    boolean isEnabled ();

    void enableEvent (int eventIndex);
    void disableEvent (int eventIndex);
    boolean isEventEnabled (int eventIndex);
}
```

Figure 5: Overview of the EventSource interface
2 Tool Usage

2.1 Tool prerequisites

The JPMF library itself requires a Java Runtime Environment version 1.5 or later, and (optionally) the log4j logging library version 1.2.15 or later. The instrumentation agent also requires ASM byte code manipulation library version 3.2 or later, which is however included in the distribution in a JPMF-specific package to avoid interference with other instances of the library that might be used by the instrumented application.

As for inputs, the JPMF library itself does not require any explicit input and by default, all the registered event sources are enabled. However, a user might wish to select a subset of performance events to record and the kind performance data to collect in addition to the data specific to performance event types. For this purpose, the library can be configured using a configuration file (see section 2.3.2), the path to which can be specified using the jpmf.conf system property.

The instrumentation agent will also require a complete list of classes to instrument. Such a list will be derived from the list of classes the user wants to instrument by scanning the class path and identifying all classes that need to be modified in order to perform the instrumentation (see section 2.3.1).

2.2 Tool activation

Being a software library rather than a standalone tool integrated in the Eclipse IDE, no special activation steps are required, apart from running the instrumented application. For the JPMF library to collect performance data, the AUT must be instrumented (manually, automatically, or both) and at least one of the registered performance events must be configured to emit notifications.

When using the automatic instrumentation agent, the Java Virtual Machine used for executing the application must be instructed to load the agent using the –javaagent command line option (see section 2.3.1).

2.3 Usage instructions and expected outputs

No special steps are required when using the library to collect application performance data. After executing the AUT, let the application perform the desired workload to generate workload-specific performance events and shut down the application. The JPMF library will produce output files containing the recorded performance events along with sampled performance data. These can be processed using the JpmfStat tool to obtain either basic statistical properties from selected events, or a text-based output suitable for further processing using heavy-duty statistical tools such as R.

2.3.1 Automatic load-time instrumentation

To simplify application instrumentation, the JPMF distribution provides an implementation of a Java agent that can be used to instrument generic Java applications. The user needs to provide a list of classes she wants to instrument and use a separate tool called ClassesToInstr to obtain a complete list of classes that will need to be modified by the instrumentation agent.
The input for the `ClassesToInstr` tool is a simple text file, with one fully qualified class name per line (an asterisk "*" can be used as a wildcard to match multiple classes). The `ClassesToInstr` tool will process all the classes reachable from the current class path and output a list of classes that will be instrumented by the agent, which then needs to be supplied to the agent.

When launching the AUT, the JVM needs to be instructed to load the JPMF instrumentation agent (packaged in a single jar file) using the `–javaagent` command line switch and providing a complete path to the agent jar file. Also, the name of the file containing the complete list of classes to instrument needs to be passed to the agent. The following example will instruct the JVM to load the JPMF agent from `/opt/jpmf/jpmf-agent.jar` file and pass `/tmp/instrument.cfg` as the name of the class list file:

```
-javaagent:/opt/jpmf/jpmf-agent.jar=/tmp/instrument.cfg
```

When the JVM is stared and the agent is activated, it intercepts all attempts to load a class and redefines (instruments) it if necessary. The instrumentation adds several fields and methods to the redefined class and also adds code at the beginning and at the end of each public method that will send performance event notifications to the JPMF runtime.

When an instrumented class is instantiated, the instrumentation code first creates an `EventSource` implementation for this particular instance, which will provide performance events for each instrumented method. The event source is then registered within JPMF.

At the moment, the downside of this approach is a limited ability to select the targets for instrumentation (only whole classes can be instrumented) and possibly slightly increased overhead compared to the manual instrumentation approach.

### 2.3.2 Configuration

The JPMF library can be configured from an XML configuration file, the location of which can be supplied through a Java system property specified on the command line:

```
-Djpmf.conf=/etc/jpmfconf.xml
```

If the `jpmf.conf` property is not set, the JPMF library by default attempts to load the configuration from a file named `jpmfconf.xml` for which it looks in the class path.

Three basic entities configurable via configuration file are events, event sources and transports. A sample configuration file named `jpmfconf.xml.sample` can be found in the JPMF `doc/` directory.

### 2.3.3 Simple data analysis

Output files produced by the JPMF are in binary format. To process these files, JPMF provides a simple tool called `JpmfStat`, which can either convert the binary files to human or machine readable text output, or calculate simple statistics. The statistics are calculated for each named event (see example below) and contains computed values such as mean, median, variance, median absolute deviation (MAD) and selected percentiles.

To execute the `JpmfStat` tool, the JVM must be instructed to execute the `org.ow2.dsrg.jpmf.JpmfStat` main class. On Linux, a simple wrapper script called `jpmfstat.sh` can be used. The `JpmfStat` tool accepts various command line arguments.
shown in Figure 6.

Figure 7 shows sample output of from the JpmfStat tool. The first part of the table contains statistical values for various types of sensors such as time in CPU clocks or application CPU time in seconds. The second part contains the computed percentiles.

### 2.4 Caveats

When used with the automatic instrumentation agent, the JPMF library provides performance data per instrumented object instance, instead of class, which is common with profiling tools. This is intentional, because the JPMF library was intended for collecting performance data from component-based applications, where component instances are usually named and even though they may be instances of the same class, their performance characteristics may be different and need to be captured separately. When aggregate performance data are required for the whole class, they can be obtained either by aggregating the instance-specific data provided by the library, or by manually instrumenting the class using static triggers, or by using a custom instrumentation tool.

At the moment, the automatic instrumentation agent does not support fine-grained selection of entities to instrument. The configuration file can only list classes only lists classes that need to be instrumented and the agent will always instrument all public methods of given classes. When other (private or protected) methods need to be instrumented, triggers should be used to manually instrument the required code parts.

The JPMF library currently supports neither network-based performance data transports, nor multiple output files.

---

**Figure 6: JpmfStat command line arguments**

- `-d "data files"` name of the data files without extension
  (default: "data/eventdata")

- `-e "event name"` selects only particular event
  (default: include all)
  (supports multiple occurrences)

- `-f "field name"` selects only particular field
  (default: include all)
  (supports multiple occurrences)

- `-o "output file"`
  (default: console)

- `-s "skip count"` skip first x results
  (default: 100)

- `-t "table output mod"` (separate/headers)
  (default: none)
  prints unprocessed data in a textual form.
  separate - prints every event into a separate file
  headers - events with the same header are printed into the same file
<table>
<thead>
<tr>
<th>type</th>
<th>mean</th>
<th>median</th>
<th>variance</th>
<th>MAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>40012664.643</td>
<td>33473648.500</td>
<td>2428576578625785.000</td>
<td>3284980.000</td>
</tr>
<tr>
<td>.../cpu.iowait#_all</td>
<td>0.071</td>
<td>0.000</td>
<td>0.164</td>
<td>0.000</td>
</tr>
<tr>
<td>.../cpu.nice#_all</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>.../cpu.user#_all</td>
<td>5.594</td>
<td>4.000</td>
<td>130.626</td>
<td>1.000</td>
</tr>
<tr>
<td>.../cpu.system#_all</td>
<td>0.311</td>
<td>0.000</td>
<td>0.565</td>
<td>0.000</td>
</tr>
<tr>
<td>.../cpu.softirq#_all</td>
<td>0.034</td>
<td>0.000</td>
<td>0.033</td>
<td>0.000</td>
</tr>
</tbody>
</table>

| type/percentiles     | 10 | 20 | 30 | 40 | ...
|----------------------|----|----|----|----|----|
| time                 | 28917998.500 | 29580559.200 | 30577046.600 | 31885300.800 | ...
| .../cpu.iowait#_all  | 0.000 | 0.000 | 0.000 | 0.000 | ...
| .../cpu.nice#_all    | 0.000 | 0.000 | 0.000 | 0.000 | ...
| .../cpu.user#_all    | 3.000 | 3.000 | 3.000 | 3.000 | ...
| .../cpu.system#_all  | 0.000 | 0.000 | 0.000 | 0.000 | ...
| .../cpu.softirq#_all | 0.000 | 0.000 | 0.000 | 0.000 | ...

Figure 7: Sample output from the JpmfStat tool