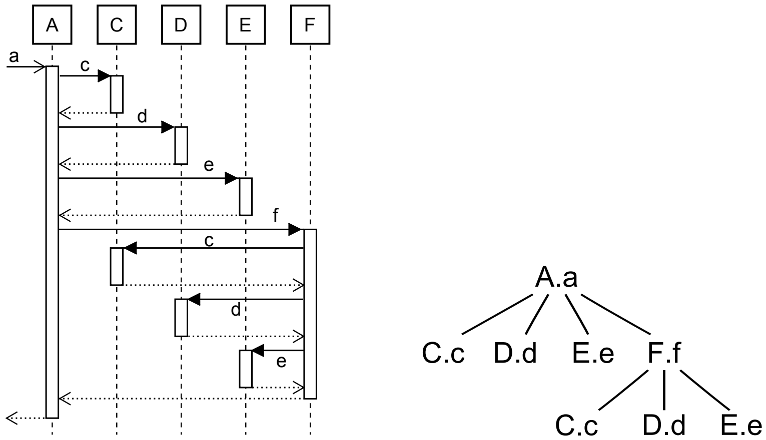
Tutorial for *Kieker*: Monitoring and Visualization of Software Behavior

DocumentURL: http://www.matthias-rohr.com/kieker/KiekerTutorial.pdf

Matthias Rohr and the Kieker project

August 8, 2008

Contents



(a) UML Sequence Diagram. (b) Dynamic call tree.

Figure 1: UML Sequence Diagram and its corresponding Dynamic Call Tree [Rohr et al., 2008].

1 Overview

Kieker is a set of tools to monitor, analyse, and visualize the runtime behavior of Java software applications. It can be used for plain Java applications, Java Web-applications that are deployed in a servlet container or application server, or for Eclipse plugins.

Kieker aims to provide a maintainable monitoring solution for Java Web applications. The overhead has to be acceptable for continuous monitoring during regular operation.

Besides monitoring, Kieker creates and visualizes models of current or past software system behavior in terms of **UML Sequence Diagrams**, **Markov chains** (for user requests or user sessions), **Component Dependency Graphs**, **Dynamic Call Trees** [Ammons et al., 1997], Trace Timing Diagrams, as well as Execution and Message trace models. Figure 1 shows an UML Sequence Diagram and a Dynamic Call Tree based on visualizations generated by Kieker.

Kieker's consists of two major components: the monitoring component ***Tpmon*** and the analysis and visualization component ***Tpan***. Both operate rather independently and are connected by a simple monitoring data format.

Kieker focuses on monitoring method (or service) calls. Therefore, it can be called a *service-level monitoring* tool. If monitoring of single statements (e.g., `a = a + 1`) is required profiling tools such as Intel's VTune can be used. Keep in mind that these tools, which can be denoted *statement-level monitoring* (= algorithm-level), usually are much more resource demanding and not supposed for continuous operation in production systems, because of the much finer monitoring granularity.

1.1 Monitoring using *Tpmon*

Tpmon is the instrumentation and monitoring component of *Kieker*. It integrates monitoring points into Java programs using aspect-oriented programming (AOP), records response times and calling paths, and stores the monitoring data into the local file system or into a database management system, such as MySQL.

Four steps are required to instrument a Java software application with *Tpmon*:

1. Download of *Tpmon* from `http://sourceforge.net/projects/kieker/`
2. Configuration of *Tpmon* (e.g., storage in file system or in a database)
3. Specification of monitoring points
4. Integration of monitoring code: Adaptation of the build- or startup-scripts of the application to be monitored;

1.2 Analysis and Visualization using *Tpan*

(The manual for *Tpan* has to be written)

1.3 Document Structure

Sections 2 and 4 contain small tutorials for *Tpmon* and *Tpan*. Complete description of *Tpmon* and *Tpan* are in Sections 3 and 5.

2 *Tpmon* Quick Start Guide – Hello World

In the following, the steps to instrument a Java program with *Tpmon* are briefly explained and demonstrated in the context of a Hello World example. Kieker's alternative monitoring and instrumentation modes are explained in Section 3.

Tpmon's essential **system requirements** are that at least Sun's Java SE 5 (JDK) (Version number $\geq$ 1.5.0) and Apache Ant[1] 1.7 are installed. *Tpmon* has been tested with Linux, Solaris, FreeBSD and Windows.

2.1 Downloading Kieker

Download the latest Kieker release from `http://sourceforge.net/projects/kieker/` and decompress it into a folder of your choice. This folder is denoted `$KIEKERHOME` in the

[1] Available at `http://ant.apache.org/`

remainder of this document. You may set it as environment variable[2], so that you can directly execute the commands shown in this Section.

- Linux/Unix: e.g.,

```
KIEKERHOME=/home/matthias/sourceforge/kieker
```

- Windows: e.g.,

```
set KIEKERHOME=c:\sourceforge\kieker
```

2.2 Configuration of *Tpmon*

The configuration file `$KIEKERHOME\tpmon.properties` specifies properties such as the folder for storing monitoring data, consisting of files (tpmon-[..].data). There are two ways to configure the data storage directory:

1. (Default) Store into the system's default temporary folder (java.io.tmpdir). For UNIX systems this is often `/tmp` and on Microsoft Windows systems it is typically some folder like `C:\\Docum...\USER...\Temp`. Be aware that the temporary folders may be deleted each time automatically by the operation system, e.g., after system restart.

2. Store monitoring data into an explicitly specified folder.

These modes can be configured via the `tpmon.properties` file. Additionally, it is possible to overwrite the properties file settings via the Java command line parameters `java -Dtpmon.storeInJavaIoTmpdir=false -Dtpmon.customStoragePath=/var/log/`.

An example for the settings in the `tpmon.properties` file are given in the following listing:

Listing 1: Linux/Unix: Specification of the monitoring data storage location

```
...
#1.3.1.a (should the systems default temporary folder by used?)
tpmon.storeInJavaIoTmpdir=true

#1.3.1.b (use this custom storage folder)
# e.g., /var/tpmon/ or "c:\tmponData\" (ensure the folder exists)
tpmon.customStoragePath=/tmp/
...
```

The other settings do require changes for this example. The default settings are that *Tpmon* stores monitoring data in the file system using an asynchronous file system writer into the system's default temporary folder.

Tpmon has to be recompiled to activate any changes in the properties file:

[2]The error message "Error opening zip file or JAR manifest missing : /external-libs/aspectjweaver.jar" means that you used this environment variable without having it set correctly.

```
ant build-all
```

This produces three new files `$KIEKERHOME/dist/KiekerTpmonCTW.jar` for compile-time instrumentation, `$KIEKERHOME/dist/KiekerTpmonLTW.jar` for load-time instrumentation, and the stand alone library `$KIEKERHOME/dist/KiekerTpmonCtrl.jar` that can be used by other instrumentation frameworks to organize tracing and data storage.

2.3 Specification of monitoring points

Next, create the following program `HelloWorld.java` and save it into any folder:

Listing 2: `HelloWorld.java`

```java
public class HelloWorld {
  public static void main (String args[]) {
    System.out.println("Hello");
    doSomething();
  }
  public static void doSomething(){
    System.out.println("doing something");
  }
}
```

In this example, we use Java Annotations to specify the Java methods to be monitored by *Tpmon* directly in the source code. The two Java methods `main` and `doSomething` are instrumented, which requires three additional lines in `HelloWorld.java` as shown in Listing 3. The Java Annotations in Line 3 and 9 trigger the aspect-oriented programming tool AspectJ to integrate monitoring logic into the application. The `import` in Line 1 is required to specify the Annotation.

Listing 3: Lines 1, 3 and 9 are added to instrument this simple “Hello World” Java program.

```java
1 import kieker.tpmon.aspects.*;
2 public class HelloWorld {
3   @TpmonMonitoringProbe()
4   public static void main (String args[]) {
5     System.out.println("Hello");
6     doSomething();
7   }
8
9   @TpmonMonitoringProbe()
10  public static void doSomething(){
11    System.out.println("doing something");
12  }
13 }
```

2.4 Integrating Monitoring Code

Tpmon supports the integration of monitoring logic at the time of compilation (of the application to be monitored) or at runtime (= class load-time). In this example, we will use the load-time integration of monitoring code into the application to be monitored. This requires the execution of the following two commands:

Listing 4: Compilation and execution with AspectJ's Java agent.

```
javac -cp $KIEKERHOME/dist/KiekerTpmonLTW.jar HelloWorld.java
java -javaagent:$KIEKERHOME/external-libs/aspectjweaver.jar -cp $KIEKERHOME/dist/KiekerTpmonLTW.jar:.: HelloWorld
```

Listing 4 should result in output like shown in Listing 5.

Listing 5: Compilation and execution with AspectJ's Java agent.

```
1 Hello
2 doing something
3  The vmid is 9069ca78e7450a28:5996e72a:119c2f563bd:-7fff hashcode -1024138306
4  Virtual Machine start time 1210156670849
5 Tpmon: TpmonShutdownHook notifies all workers to initiate shutdown
6 Tpmon: TpmonShutdownHook can terminate since all workers are finished
```

Lines 1 and 2 are produced by the `printlns` of `HelloWorld`. The other lines are *Tpmon* status information.

Note, the command `java HelloWorld` does not fail if *Tpmon* is not in the classpath. Weaving and integration of monitoring logic is only performed if both the AspectJ javaagent is loaded into the Java VM *and* the *Tpmon* load-time library is in the classpath.

More verbose output Additional verbose output, such as on which Java files are woven, is produced for:

Listing 6: Verbose execution with AspectJ's Java agent.

```
java -Daj.weaving.verbose=true -Dorg.aspectj.weaver.showWeaveInfo=true -javaagent:$KIEKERHOME/external-libs/aspectjweaver.jar -cp $KIEKERHOME/dist/KiekerTpmonLTW.jar:.: HelloWorld

info AspectJ Weaver Version 1.5.2a built on Friday Aug 18, 2006 at 18:40:31 GMT
info register classloader sun.misc.Launcher$AppClassLoader@32582734
info using configuration file:/home/matthias/projekte/kieker/sourceforge/kieker/dist/KiekerTpmonLTW.jar!/META-INF/aop.xml
info register aspect kieker.tpmon.aspects.TpmonMonitorAnnotation
info weaving 'HelloWorld'
```

```
8 ...
```

Line 7 provides the important status information that *Tpmon*'s monitoring code is woven into HelloWorld.

Monitoring data The monitoring data for Listing 4 is written to a file which is named `/tmp/tpmon--*-*.dat`, where $*$ represents some number. It should contain two lines like:

Listing 7: Example monitoring data

```
0;HelloWorld.doSomething();nosession;1210156671935-1-0;
   1210156671935804069;1210156671940322555;-1024138307;-1;-1
0;HelloWorld.main(java.lang.String[]);nosession;1210156671935-1-0;
   1210156671935203951;1210156671948286691;-1024138307;-1;-1
```

Columns 5 and 6 are start timestamps and end timestamps for executions of operations. The timestamps are provided in nanoseconds since 1.1.1970. The monitoring data file is a CSV file. Instead of commas, it uses semi-colons as separators, because commas can be part of operation names (e.g., `sum(int a,int b)`). A detailed description of the monitoring data format can be found in Section 3.6.

3 *Tpmon*: Instrumenting and Monitoring Java Applications

The small tutorial in Section 2 uses only one of *Tpmon*'s monitoring modes. You have to make the following design decisions when you are using *Tpmon*:

- When and how should the monitoring logic be integrated?
  - **Compile-time integration** (using compile-time-weaving): In this mode, you the build-scripts or build-commands of your Java program are adjusted, such that the bytecode is already instrumented. During runtime, no aspect-oriented programming features (related to reflection and class loading) are required and only the *Tpmon*library has to be added to the execution classpath.
  - Class **load-time integration**: Load-time integration combines monitoring logic with program logic of a class at the time the Java Virtual Machine accesses program class for the first time (during runtime. A advantage of load-time weaving is that no large changes to the build scripts are required (only *Tpmon* might have to be added to the compile-time classpath). It is can also be used to instrument (full instrumentation) programs for that no source code is available. However, the start-scripts for your application, or the configuration files of the middleware that runs your application, have to be extended by adding the AspectJ javaagent that combines monitoring logic and program logic during runtime. Load-time integration may fail in some environments that use special dynamic features such as

reflection, customized class-loaders, and particular types of remote communication. For instance, runtime instrumentation fails for Eclipse Plugins[3], RMI objects (code base changes are detected and cause exceptions), and for classes and objects that are defined during runtime via reflection, such as it is used for the Hessian Web-service library.

- Which Java methods should be monitored?
  - **Full instrumentation**: A full instrumentation of a complete Java package does not require changes of the source code of the program to be monitored. However, instrumenting all methods within a package can impose a too large monitoring overhead.
  - **Partial instrumentation**: Partial instrumentation is done by placing a special Java Annotation in front of every Java method for which instrumentation is desired. The annotations can be used by compile-time integration or load-time integration.
- Where should the monitoring data be stored?
  - **File system**: Storing the monitoring data into the file system is the most performant data storage mode of Kieker. However, if the files have to be transported regularly to an other host for analysis, it might be better directly store it into a remote database. In particular if continuous monitoring during regular operation is required, it is suggested to transfer the monitoring data just after it has been observed to an other host's database for persistent storage and analysis.
  - **Database management system**: Database management systems provide means to efficiently store large amounts of data and provide a powerful query language for data access. *Tpmon* requires a small additional overhead for storing in a remote database system compared to storing in a local file system. The overhead (e.g., 0.1-5 milliseconds in average) is only a problem if precise timing analysis, e.g., for anomaly detection, is required.

3.1 Downloading Kieker

The latest Kieker release can be downloaded from `http://sourceforge.net/projects/kieker/`. The folder where the uncompressed release is located on your hard disc is denoted `$KIEKERHOME` in the remainder of this document. You may set it as environment variable, so that you can directly execute the commands shown in this section

- Linux/Unix: e.g.,

```
KIEKERHOME=/home/matthias/sourceforge/kieker
```

- Windows: e.g.,

```
set KIEKERHOME=c:\sourceforge\kieker
```

[3] Other profilers have also problems with profiling Eclipse Plugins

3.2 *Tpmon*'s configuration file `tpmon.properties`

The configuration file `tpmon.properties` specifies *Tpmon*'s runtime behavior (database or file system storage) and the path to required libraries for the build process. In the following, we explain properties (default settings are underlined):

1.1.1 `storeInDatabase=[true|false]`
Here it is specified whether *Tpmon* stores the data in a remote database or in the local file system.

1.1.2 `debug=[true|false]`
In debug mode, tpmon is very verbose during runtime.

1.1.3 `monitoringEnabled=[true|false]`
Should monitoring be enabled right after the application start? You can also use the tpmon-control-servlet to activate monitoring later.

1.2.1 `dbconnectionAddress=`
Specifies the database connection parameter for the initialization of the database connection. Example: `jdbc:mysql://HOST/DATABASENAME?user=USER&password=PASS`

1.2.2 `dbTableName=turbomon9`
Name of the database table. The SQL statement for creating a suitable database table in MySQL can be found in file `table-for-monitoring.sql`.

1.2.3 `setInitialExperimentIdBasedOnLastId=[true|false]`
Should *Tpmon* look into the database for the last experiment identifier and use a new one increased by one?

1.2.3 `useAsyncDbconnector=[true|false]`
If true, *Tpmon* uses a connection pool for storing data into the database. This is more performant and the communication with the database runs in separate threads so that the application under monitoring has not to wait each time a monitoring observation has to be stored.

1.3.1 `tpmon.storeInJavaIoTmpdir=[true|false]` & `tpmon.customStoragePath=..`
There are two ways to specify where *Tpmon* should place its monitoring data files (`tpmon--*-*.dat`).:

a `tpmon.storeInJavaIoTmpdir=[true|false]`
Store into the system's default temporary folder (java.io.tmpdir). For UNIX systems this is often defaults to `/tmp` and on Windows systems to some folder like `C:\Docum...\USER...\Temp`. Be aware that the temporary folders may be deleted each time automatically by the operation system, e.g., after system restart.

b `tpmon.customStoragePath=/tmp/`
Store monitoring data into an explicitly specified folder.

These modes can be configured via the `tpmon.properties` file. Additionally, it is possible to overwrite the properties file setting via the Java command line parameters `java -Dtpmon.storeInJavaIoTmpdir=false -Dtpmon.customStoragePath=/var/log/`.

A new file will be created each 22,000 lines.

An example for the settings in the `tpmon.properties` file are given in the following listing:

Listing 8: Linux/Unix: Specification of the monitoring data storage location

```
...
#1.3.1.a (should the systems default temporary folder by used?)
tpmon.storeInJavaIoTmpdir=true

#1.3.1.b (use this custom storage folder)
# e.g., /var/tpmon/ or "c:\tmponData\" (ensure the folder
   exists)
tpmon.customStoragePath=/tmp/
...
```

1.3.2 `asyncFsWriter=[true|false]`

If true, *Tpmon* will use a pool of independent threads to store monitoring data in the file system. This results in less outliers in the monitoring data that can occur for I/O processing. You can safely use `true` here, but you will have to add a `system.exit(0)` to your application (not required for application servers, or servlet containers) or it will never terminate. *Tpmon* ensures that no data is lost when the application terminates normally.

2.1 `tpmon.javac.debug=[true|false]`

If true, `javac` will be called with the debug flag.

3.3 Building the *Tpmon* Libraries

The *Tpmon* libraries (`KiekerTpmonLTW.jar` and `KiekerTpmonCTW.jar`) have to be recompiled each time `tpmon.properties` or `aop.xml` is changed. The command for rebuilding *Tpmon* is `ant build`. If compilation succeeds, Ant returns `BUILD SUCCESSFUL` and the two libraries are created in the folder `dist`.

By default, the *tpmonltw.jar* will be configured to look for potential monitoring points in all packages except in the Java's own libraries and in some internal Kieker packages.

3.4 Instrumentation

As discussed at the beginning of this section, you have to chose whether you want that the monitoring logic is combined with the application logic during during load-time, i.e., the time at which the Java Virtual Machine load class file when they are required the first time during runtime, or at compile-time. Additionally, you have to chose whether full or partial instrumentation should be used.

3.4.1 Load-time instrumentation

There are two possible modes for load-time instrumentation supported by *Tpmon*: Full instrumentation of selected Java packages and partial instrumentation via Java Annotations in the program source code.

Full instrumentation This variant, i.e. full instrumentation via load-time weaving, can be used if no source code is available. This means only the `.jar` or `.class` files have to be provided. However, all methods of the classes in the packages specified in the `aop.xml` are monitored by the aspect `TpmonMonitorFullInstrumentation`, which may cause large monitoring overhead.

- Specification of packages to instrument in `aop.xml`[4]: Specify in the first part of the `aop.xml` via `include exclude` tags which packages should be instrumented. Take a look at `aop.xml.example` for examples.
- Activating the full-instrumentation aspect in `aop.xml`: In the last part of the `aop.xml` uncomment the aspect `TpmonMonitorFullInstrumentation` and put all other aspects into comments.

Full instrumentation via load time weaving can be tested using the ant build target `run-tests-loadTimeWeaving-bookstoreWithoutAnnotation` of the `build.xml` tpmon package. For this test, only the `aop.xml` needs to be adjusted, as described above.

Partial instrumentation (using Java Annotations)

- Specification of packages to instrument in `aop.xml`[5]: Specify in the first part of the `aop.xml` via `include exclude` tags which packages will be scanned for the `TpmonMonitoringProbe` Java Annotations. Take a look at `aop.xml.example` for examples.
- Specification of the instrumentation mode `aop.xml`: In the last part of the `aop.xml` uncomment the aspect `TpmonMonitorAnnotation` and put all other aspects into comments.
- Modify the source code of the program to be instrumented such as in the Listing 3 of the Section 2.3. Also add the `import` as in Line 1 of Listing 3.

[4]Located in `src/kieker/tpmon/META-INF/`
[5]Located in `src/kieker/tpmon/META-INF/`

3.4.2 Compile-time instrumentation

For compile-time instrumentation, it is required install the AspectJ compiler from `http://www.eclipse.org/aspectj/`.

Full instrumentation

- Copy the aspect file `TpmonMonitorFullInstrumentation.aj` into a folder of your program sources.
- In your application build-scripts, replace the compiler with the AspectJ compiler. Add `KiekerTpmonCTW.jar` to compilation classpath. An example ant script target for this compilation can be found in *Tpmon*'s `build.xml` (see target `compile-tests-compileTimeWeaving-bookstore`).

Partial instrumentation (using Java Annotations)

- Copy the aspect file `TpmonMonitorAnnotation.aj` into a folder of your program sources.
- Modify the source code of the program to be instrumented as shown in Listing 3 of the Section 2.3. Also add the `import` as in Line 1 of Listing 3.
- In your application build-scripts, replace the compiler with the AspectJ compiler. Add `KiekerTpmonCTW.jar` to the compilation classpath. An example ant script target for this compilation can be found in *Tpmon*'s `build.xml` (see target `compile-tests-compileTimeWeaving-bookstore`).

3.5 Executing the instrumented program

The execution scripts and application server startup scripts have to be adjusted to run the instrumented program with monitoring.

Executing compile-time instrumented programs Only the `KiekerTpmonCTW.jar` has to be added to the runtime classpath in order to execute programs that have been instrumented with compile-time instrumentation. It is also be required to the database driver to the classpath if database storage was selected. Output similar to that in Lines 6 and 7 in Listing 9 results short after the first instrumented method is executed.

Listing 9: Execution of a compile-time instrumented program

```
1 java -cp $KIEKERHOME/dist/KiekerTpmonCTW.jar:.: Main
2
3 ...
4 The vmid is 9069ca78e7450a28:5996e72a:119c2f563bd:-7fff hashcode
    -1024138306
5 Virtual Machine start time 1210156670849
6 ...
```

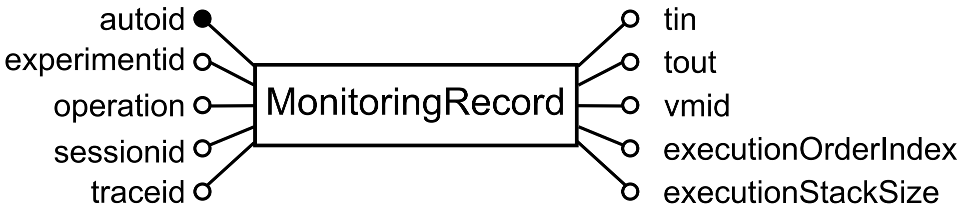


Figure 2: Database schema for the monitoring data (version 0.9).

Executing load-time instrumented programs Executing load-time instrumented programs requires:

- adding `KiekerTpmonLTW.jar` to the runtime classpath (maybe also the database driver), and
- adding the parameter `-javaagent:ADJUST/aspectjweaver.jar` to the Java Virtual Machine startup call.

To activate monitoring for load-time instrumented programs in the Apache Tomcat servlet container, add the following line to Tomcat's `catalina.sh` (right at the beginning of the file):

```
JAVA_OPTS="-javaagent:ADJUST/aspectjweaver.jar"
```

Windows users have to add "set" in front of this line.

An example for executing a load-time instrumented program can be found in the tutorial in Section 2.

3.6 Monitoring data format

The **database schema** of the monitoring data is displayed in the Entity-relationship Diagram (Figure 2). The data format for **monitoring data files** only differs to this schema by not containing the first column `autoid`. Each entry, called monitoring record, contains two timestamps (`tin` and `tout`) that denote the start time and end time of an operation execution, and attributes that describe a context of the measurement (`experimentid`, `operation`, `sessionid`, `traceid`, `vmid`). The `operation` attribute names the operation that corresponds to the execution monitored, and `traceid` is unique for executions of the same trace. Java Web application technology provides the concept of sessions (monitored as `sessionid`) that connect single user requests. The `vmid` allows to distinguish different Java Virtual Machines. The `executionOrderIndex` and `executionStackSize` are used if *Tpmon* monitors distributed systems. Local clock times are not a reliable source for determining the order of executions (e.g., for creating Sequence Diagrams) in a distributed system.

Listing 10: SQL script for preparing the database table

```
CREATE TABLE turbomon9 (
`autoid` BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
`experimentid` SMALLINT NOT NULL DEFAULT '0',
`operation` VARCHAR( 160 ) NOT NULL ,
`sessionid` VARCHAR( 34 ) NOT NULL ,
`traceid` VARCHAR( 34 ) NULL ,
`tin` BIGINT( 19 ) UNSIGNED NOT NULL ,
`tout` BIGINT( 19 ) UNSIGNED NOT NULL ,
`vmid` INT( 10 ) NOT NULL DEFAULT '-1',
`executionOrderIndex` INT( 10 ) NOT NULL DEFAULT '-1',
`executionStackSize` INT( 10 ) NOT NULL DEFAULT '-1',
INDEX (operation(16)), INDEX (traceid), INDEX (tin)
) ENGINE = MYISAM;
```

3.7 Monitoring overhead

The monitoring overhead depends on many aspects such as the number of monitoring points, on the workload, and on the monitoring point invocation frequency. Therefore, it is no possible to specify the monitoring overhead in general. Since *Tpmon* is created for continuous operation during regular operation, it is designed to keep the monitoring overhead reasonably low.

Tpmon is installed in several production system of industry partners to instrument the most important services (about 50 monitoring points to say a number). No significantly noticeable monitoring overhead was reported for these installations. In general, it is our experience that imposing less than 10-20% overhead on response times is accepted by enterprise system administrators in exchange for monitoring and its benefits.

In the case studies using our instrumentation prototype Kieker, we observed an overhead on response times of 14% for the default asynchronous file system storage mode (TpmonAsyncFs) (see Section 3.2). To quantify the overhead, a case study was performed using the iBATIS JPetStore demo Web-application. This software was deployed in a standard Java servlet container (Apache Tomcat). Probabilistic medium-level multi-user workload was generated using Markov4JMeter and Apache JMeter. Based on Tomcat's access logs (12 services for JPetStore) we compared container measured response times without and with *Tpmon* monitoring using 19 *Tpmon* monitoring points.

In Figure 3.7 and Table 1 and 2 the response time statistics of various instrumentation variants in the case study are compared. Table 1 shows that the asynchronous writing to the local file system is the most performant storage mode, imposing in average an overhead of 14%. These results conform with the results provided by Govindraj et al. [2006] reporting about 10% overhead for the monitoring framework InfraRED, which also uses aspect-oriented programming (AOP).

For distributed software systems, an additional overhead exists for remote communication, in order to connect traces over multiple nodes.

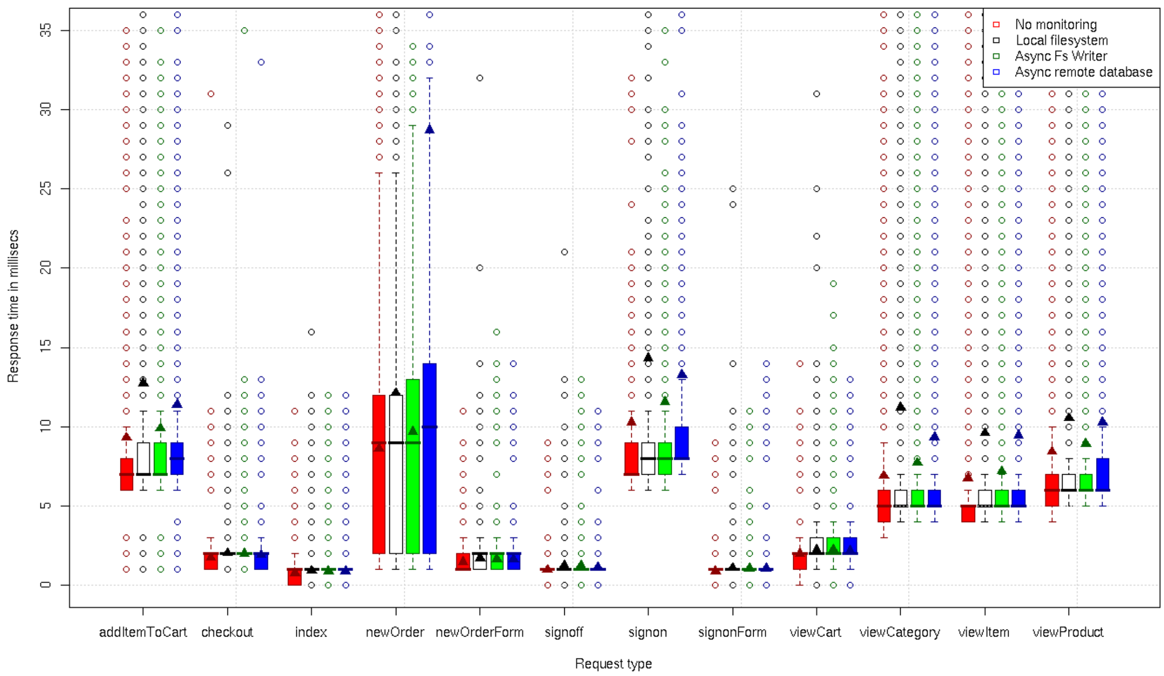


Figure 3: JPetStore: Comparison of Monitoring Overhead. Triangles indicate the corresponding arithmetic mean.

	TpmonFs	TpmonAsyncFS	TpmonAsyncDB	TpmonSyncDB
addItemToCart	0.37	0.06	0.22	4.99
checkout	0.17	0.16	0.11	24.12
index	0.22	0.19	0.16	43.51
newOrder	0.41	0.13	2.34	4.99
newOrderForm	0.17	0.12	0.11	23.83
signoff	0.25	0.25	0.17	41.55
signon	0.39	0.12	0.29	5.02
signonForm	0.24	0.19	0.19	42.57
viewCart	0.12	0.12	0.09	17.71
viewCategory	0.63	0.12	0.35	6.91
viewItem	0.42	0.06	0.40	6.72
viewProduct	0.25	0.06	0.22	5.24

Table 1: JPetStore: Relative average slowdown compared to no monitoring

	TpmonFs	TpmonAsyncFS	TpmonAsyncDB	TpmonSyncDB
addItemToCart	3.42	0.59	2.09	46.36
checkout	0.29	0.28	0.19	41.26
index	0.16	0.14	0.12	32.02
newOrder	3.53	1.08	20.11	42.78
newOrderForm	0.24	0.18	0.16	34.37
signoff	0.24	0.24	0.17	39.92
signon	4.04	1.26	2.97	51.63
signonForm	0.21	0.16	0.17	37.01
viewCart	0.23	0.25	0.18	35.23
viewCategory	4.31	0.84	2.43	47.53
viewItem	2.84	0.44	2.68	45.32
viewProduct	2.11	0.49	1.85	44.07

Table 2: JPetStore: Absolute average slowdown compared to no monitoring in milliseconds

3.8 Troubleshooting – Why does *Tpmon* fail?

***Tpmon* does not compile – “ant build” fails** There can be several reasons why it fails. First you should ensure that *Tpmon*’s basic requirements (at least Java SD 5 (version number 1.5), and Apache Ant 1.7) are satisfied. Possible reasons for compilation failures are

- Your user account might use a Java version older than version number 1.5. Ensure that “`java -version`” reports at least 1.5.* = version 5.
- Apache Ant might use a Java version older than version number 1.5 or an old version of Ant is used. Ensure that “`ant -v build`” starts like this:

```
Apache Ant version 1.7.0 compiled on March 11 2007
Buildfile: build.xml
Detected Java version: 1.6 in: /opt/sun-jdk-1.6.0.02/jre
```

If an old Java version is used compilation will fail for Java Annotations (@ ...) and generics (e.g., `Vector<String>` ...).

Checking the data storage part of *Tpmon* If *Tpmon* compiles, but it does not store monitoring data, it is a good starting point to first check the simple parts of *Tpmon*The most simple part is the logic responsible for storing observed response times into the file system or database. This might fail, because of wrong database connection parameters, or in case of file system storage, the output folder or writing permissions might be missing.

To test the storage part of tpmon go to the folder `$KIEKERHOME` and run

Listing 11: Testing the storage functionality

```
ant tpmon-test-storage
```

The results depends of the properties `storeInDatabase, monitoringEnabled` and `asyncFsWriter` in `tpmon.properties`. For storing in to the file system folder `\tmp` the output might be something like this:

Listing 12: Output example for storage test

```
[java] Starting test by adding 10000 monitoring events
   [java] The vmid is 0cfeed62999abb27:a8b0fa2:119c26cc798:-7fff
      hashcode -1033091175
   [java] Virtual Machine start time 1210147718971
   [java] Sleeping for 8 seconds
   [java] 10000 more monitoring points
   [java] Calling system.exit(0)
   [java] Tpmon: TpmonShutdownHook notifies all workers to initiate
       shutdown
   [java] Tpmon: TpmonShutdownHook can terminate since all workers
      are finished
```

This should create 20,000 lines of fake monitoring data created in a file `tpmon-*.dat` in the system's default temporary folder (e.g., `/tmp/` or `C:\Docum...\USER...\Temp`):

Listing 13: Example fake monitoring data produced by the storage test

```
0;0component0method;sessionid;requestid;123;123;-1;0;0
0;1component1method;sessionid;requestid;123;123;-1;1;1
0;2component2method;sessionid;requestid;123;123;-1;2;2
0;3component3method;sessionid;requestid;123;123;-1;3;3
0;4component4method;sessionid;requestid;123;123;-1;4;4
0;5component5method;sessionid;requestid;123;123;-1;5;5
...
```

The storage test should be successful before other *Tpmon* tests are executed. Possible reasons for failures are:

- Both storage modes:
  - The file `$KIEKERHOME/dist/KiekerTpmonCTW.jar` is outdated or missing (it contains the tpmon.properties used during runtime). Create it: "`ant build-tpmon-ctw`".
  - You did not call "`ant tpmon-test-storage`" from the folder `$KIEKERHOME`
- Monitoring data is stored in the file system:
  - The folder configured for storing data does not exist. Try using a different folder a the system's default temporary folder (see Section 3.2, properties on file system storage (1.3.1)).
  - The current user has no permissions to create a file `tpmon*` in the folder specified.
- Monitoring data is stored in a database:
  - Wrong database connection properties specified in `tpmon.properties`

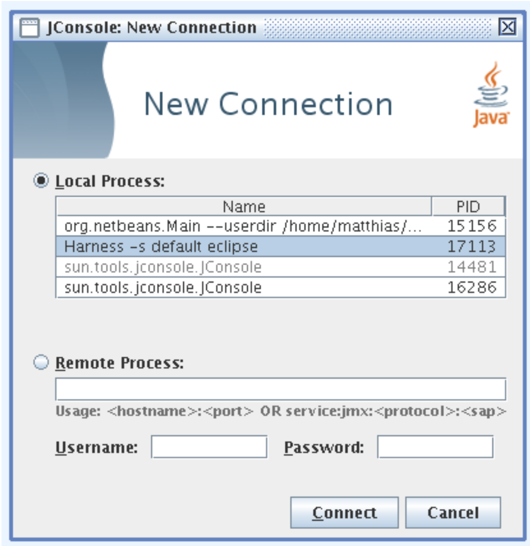


Figure 4: Attaching Java's *console* to a Virtual Machine instance.

- The database driver is not within the classpath during execution
- The database does not accept connections form your host. In case of MySQL, ensure that you can connect to the database using the command line command "`mysql`".
- There is no table for storing the monitoring data in the database – create it with the SQL statement in the file `table-for-monitoring.sql`.
- The file `$KIEKERHOME/dist/KiekerTpmonCTW.jar` is outdated – rebuild it with "`ant build-tpmon-ctw`".

Problem diagnosis using *jconsole* *Tpmon* fails to monitor if essential libraries are not available during runtime. In all instrumentation variants, a *Tpmon* library must be present in the classpath (or found by the application server). A common load-time instrumentation problem is that the *aspectjweaver.jar* is not loaded as javaagent.

The *console* command-line tool, which is part of Sun's JDK, allows to diagnose these problems, as it shows which libraries are in the classpath and which javaagents have been added. If *jconsole* is stared after the application to be monitored, it should offer to connect to the active Virtual Machine instances (see Figure 4). As shown in Figure 5, *jconsole* allows to check whether the *Tpmon* library is in the classpath and whether the *aspectjweaver.jar* is used as javaagent (which is required for AspectJ load-time instrumentation). Some older Java versions may require the additional Virtual Machine parameter *-Dcom.sun.management.jmxremote* in order to use *jconsole*.

4 *Tpan* quick start guide – Hello world

...

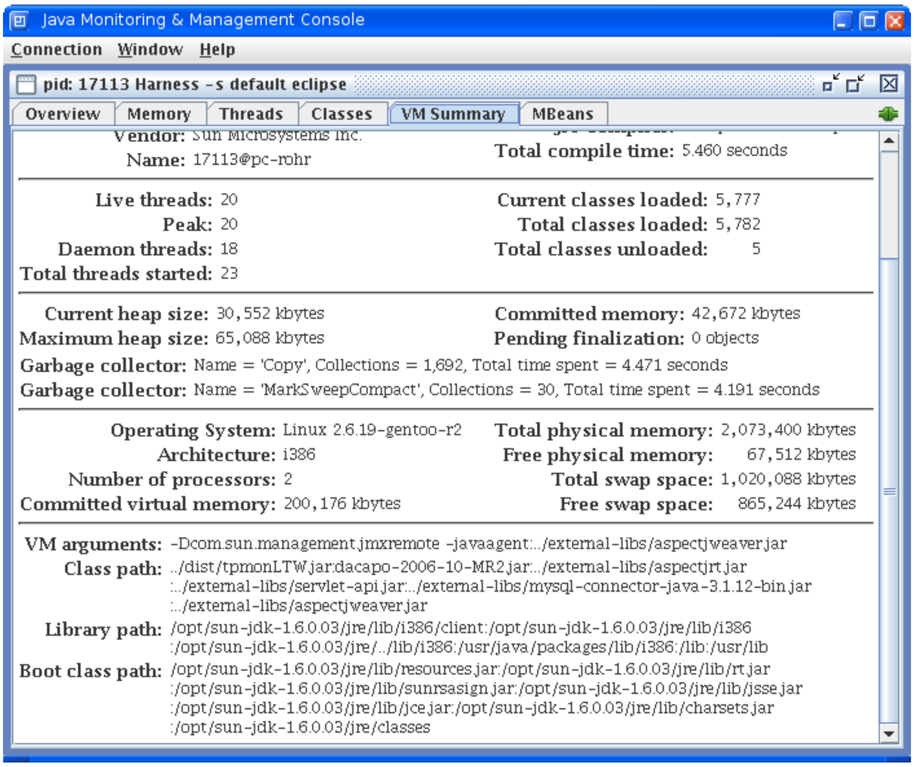


Figure 5: Problem diagnosis using *jconsole*: A *Tpmon* library has to be within the classpath (except an application server is used that itself loads additional libraries). For load-time instrumentation, the *aspectjweaver.jar* has to be used as javaagent (see VM arguments).

5 *Tpan*: Kieker's Analysis and Visualization Component

...

References

G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming language design and implementation (PLDI'97)*, pages 85–96. ACM, 1997. ISBN 0-89791-907-6. doi: 10.1145/258915.258924.

K. Govindraj, S. Narayanan, B. Thomas, P. Nair, and S. P. On using AOP for Application Performance Management. In M. Chapman, A. Vasseur, and G. Kniesel, editors, *AOSD 2006 - Industry Track Proceedings (Technical Report IAI-TR-2006-3, University of Bonn)*, pages 18–30, Mar. 2006.

M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, and W. Hasselbring. Trace-context sensitive performance models from monitoring data of software systems. In C. Lebsack, editor, *Proceedings of the Workshop on Tools Infrastructures and Methodologies for the Evaluation of Research Systems (TIMERS'08) at IEEE International Symposium on Performance Analysis of Systems and Software 2008*, pages 37–44, Apr. 2008.