# Contents

1 **Introduction** 3  
1.1 What is Kieker? 3  
1.2 Structure of this User Guide 5  

2 **Quick Start Example** 6  
2.1 Download and Installation 6  
2.2 Bookstore Example Application 7  
2.3 Monitoring with Kieker.Monitoring 11  
2.4 Analysis with Kieker.Analysis 15  

3 **Kieker.Monitoring Component** 19  
3.1 Monitoring Controller 19  
3.2 Kieker.Monitoring Configuration 22  
3.3 Monitoring Records 23  
3.4 Monitoring Probes 25  
3.5 Monitoring Writers 25  

4 **Kieker.Analysis Component** 29  
4.1 Analysis Controller 29  
4.2 Monitoring Readers 30  
4.3 Analysis Plugins 31  
4.4 Monitoring Record Consumer Plugins 31  

5 **Kieker.TraceAnalysis Tool** 34  
5.1 Monitoring Trace Information 35  
5.2 Trace Analysis and Visualization 40  
5.3 Example Kieker.TraceAnalysis Outputs 42  

**Appendix** 49  

**Bibliography** 83
1 Introduction

Modern software applications are often complex and have to fulfill a large set of functional and non-functional properties. The internal behavior of such large systems cannot easily be determined on the basis of the source code. Furthermore, existing applications often lack sufficient documentation which makes it cumbersome to extend and change them for future needs. A solution to these problems can be monitoring, which allows to log the behavior of the application and to discover the application-internal control flows and response times of method executions.

The monitoring of the behavior can help in detecting performance problems and faulty behavior, capacity planning, and many other areas. The Kieker framework provides the necessary monitoring capabilities and comes with tools and libraries for the analysis of monitored data. Kieker was designed for continuous monitoring in production systems inducing only a very low overhead.

1.1 What is Kieker?

The present version of Kieker is a monitoring and analysis framework for Java applications. Support for other platforms, such as .NET, is currently under development. Figure 1.1 shows the framework’s composition based on the two main components Kieker-Monitoring and Kieker.Analysis.

![Figure 1.1: Overview of the framework components](image-url)
The **Kieker.Monitoring** component is responsible for program instrumentation, data collection, and logging. Its core is the **MonitoringController**. The component **Kieker.Analysis** is responsible for reading, analyzing, and visualizing the monitoring data. Its core is the **AnalysisController** which manages the life-cycle of the monitoring reader and all analysis plugins.

The monitoring and analysis parts of the **Kieker** framework are composed of subcomponents which represent the different functionalities of the monitoring and analysis tasks. The important interaction pattern among the components is illustrated in Figure 1.2 but will be explained furthermore throughout the course of this user guide.

![Figure 1.2: Communication among Kieker framework components](image)

The monitoring probes create the monitoring records containing the monitoring data and deliver them to the monitoring controller. The monitoring controller employs the monitoring writers to write these monitoring records to a monitoring log or stream. For analyzing purposes, a monitoring reader reads the records from the monitoring log/stream. These records can then be further processed by the analysis plugins.

**Kieker** includes monitoring writers and readers for filesystems, SQL databases, and the Java Messaging Service (JMS) [3]. A special feature of **Kieker** is the ability to monitor and analyze (distributed) traces of method executions and corresponding timing information. For monitoring this data, **Kieker** includes monitoring probes employing AspectJ [8], Java EE Servlet [4], Spring [5], and Apache CXF [6] technology. The **Kieker.TraceAnalysis** tool, itself implemented as a **Kieker.Analysis** plugin (Figure 1.1), allows to reconstruct and visualize architectural models of the monitored systems, e.g., as sequence and dependency diagrams.
1.2 Structure of this User Guide

Based on a simple example, Chapter 2 demonstrates how to manually instrument Java programs with Kieker.Monitoring in order to monitor timing information of method executions, and how to use Kieker.Analysis to analyze the monitored data. Chapter 3 provides a more detailed description of Kieker.Monitoring and shows how to implement and use custom monitoring records, monitoring probes, and monitoring writers. A more detailed description of Kieker.Analysis and how to implement and use custom monitoring readers, and analysis plugins follows in Chapter 4. Chapter 5 demonstrates how to use Kieker.TraceAnalysis for monitoring, analyzing, and visualizing trace information. Additional resources are included in the appendix.

The Java sources presented in this user guide are included in the examples/userguide/ directory of the Kieker distribution (see Section 2.1).
2 Quick Start Example

This chapter provides a brief introduction to Kieker based on a simple Bookstore example application. Section 2.1 explains how to download and install Kieker. The Bookstore application itself is introduced in Section 2.2, while the following sections demonstrate how to use Kieker for monitoring (Section 2.3) and analyzing (Section 2.4) the resulting monitoring data.

2.1 Download and Installation

The Kieker download site[1] provides archives of the binary and source distribution, the Javadoc API, as well as additional examples. For this quick start guide Kieker’s binary distribution, e.g., kieker-1.3_binaries.zip, is required and must be downloaded. After having extracted the archive, you’ll find the directory structure and contents shown in Figure 2.1.

```
kieker-1.3/
    bin/ .............................................. Call scripts for Kieker tools
    ...
    dist/ ............................................. The Kieker framework libraries
    kieker-1.3.jar
    kieker-monitoring-servlet-1.3.war
    doc/ ............................................. PDF file of this document
    kieker-1.3_userguide.pdf
    Examples/
    kieker-monitoring.properties
    ...
    ...
    META-INF/ ......................................... Example configuration files
    ...
```

Figure 2.1: Directory structure and contents of Kieker’s binary distribution

The Java sources presented in this user guide are included in the examples/user-guide/ directory. The file kieker-1.3.jar contains the Kieker.Monitoring and Kieker.-Analysis components, as well as the Kieker.TraceAnalysis tool. A Servlet-based Web application, provided in kieker-monitoring-servlet-1.3.war, can be used to control the status of Kieker.Monitoring in Java EE environments. The sample Kieker.Monitoring configuration file kieker.monitoring.properties will be detailed in Chapter 3. Since Kieker uses the Apache Commons library \[7\] as a logging interface, the file commons-logging-1.1.1.jar is the only dependency to a third-party library which is needed to execute Kieker in any case.

2.2 Bookstore Example Application

The Bookstore application is a small sample application resembling a simple bookstore with a market-place facility where users can search for books in an online catalog, and subsequently get offers from different book sellers. Figure 2.2 shows a class diagram describing the structure of the bookstore and a sequence diagram illustrating the dynamics of the application.

![UML class diagram and sequence diagram of the Bookstore application](image)

Figure 2.2: UML class diagram \[a\] and sequence diagram \[b\] of the Bookstore application
The bookstore contains a catalog for books and a customer relationship management
system (CRM) for the book sellers. To provide this service, the different classes provide
operations to initialize the application, search for books, and get offers or searched books.
In this example, the methods implementing these operations are merely stubs. However,
for the illustration of Kieker they are sufficient and the inclined reader may extend the
application into a real bookstore.

The directory structure of the Bookstore example is shown in Figure 2.3 and com-
prises four Java classes in its source directory src/bookstoreApplication/ which are
explained in detail below.

The class BookstoreStarter, shown in Listing 2.1 contains the application’s main
method, i.e., the program start routine. It initializes the Bookstore and issues five
search requests by calling the searchBook method of the bookstore object.

```java
package bookstoreApplication;

public class BookstoreStarter {

    public static void main(String[] args) {
        Bookstore bookstore = new Bookstore();
        for (int i = 0; i < 5; i++) {
            System.out.println("Bookstore.main: Starting request "+ i);
            bookstore.searchBook();
        }
    }
}
```

Listing 2.1: BookstoreStarter.java

Figure 2.3: The directory structure of the Bookstore application
The **Bookstore**, shown in Listing 2.2, contains a catalog and a CRM object, representing the catalog of the bookstore and a customer relationship management system which can provide offers for books out of the catalog. The business method of the bookstore is `searchBook()` which will first check the catalog for books and then check for offers.

In a real application these methods would pass objects to ensure the results of the catalog search will be available to the offer collecting method. However, for our example we omitted such code.

```java
package bookstoreApplication;

public class Bookstore {
    private final Catalog catalog = new Catalog();
    private final CRM crm = new CRM(catalog);

    public void searchBook() {
        catalog.getBook(false);
        crm.getOffers();
    }
}
```

Listing 2.2: Bookstore.java

The customer relationship management for this application is modeled in the **CRM** class shown in Listing 2.3. It only provides a business method to collect offers which uses the catalog for some lookup. The additional catalog lookup is later used to illustrate different traces in the application.

```java
package bookstoreApplication;

public class CRM {
    private final Catalog catalog;

    public CRM(final Catalog catalog) {
        this.catalog = catalog;
    }

    public void getOffers() {
        catalog.getBook(false);
    }
}
```

Listing 2.3: CRM.java
The final class is `Catalog` shown in Listing 2.4. It resembles the catalog component in the application.

```java
package bookstoreApplication;

public class Catalog {
    public void getBook(final boolean complexQuery) {
    }
}
```

Listing 2.4: Catalog.java

After this brief introduction of the application and its implementation, the next step is to see the example running. To compile and run the example, the commands in Listing 2.5 can be executed. This document assumes that the reader enters the commands in the example directory. For this first example this is `examples/userguide/ch2-bookstore-application/`.

Windows comes with two command-line interpreters called `cmd.exe` and `command.com`. Only the first one is able to handle wildcards correctly. So we recommend using `cmd.exe` for these examples.

```
> mkdir build
> javac src/bookstoreApplication/* .java \-d build
> java -classpath build bookstoreApplication.BookstoreStarter
```

Listing 2.5: Commands to compile and run the Bookstore application

The first command compiles the application and places the resulting four class files in the `build/` directory. To verify the build process, the `build/` directory can be inspected. The second command loads the bookstore application and produces the output shown in Listing 2.6.

```
Bookstore.main: Starting request 0
Bookstore.main: Starting request 1
Bookstore.main: Starting request 2
Bookstore.main: Starting request 3
Bookstore.main: Starting request 4
```

Listing 2.6: Example run of the Bookstore application

In this section, the Kieker example application was introduced and when everything went well, the bookstore is a runnable program. Furthermore, the composition of the application and its function should now be present. The next Section 2.3 will demonstrate how to monitor this example application employing `Kieker.Monitoring` using manual instrumentation.
2.3 Monitoring with Kieker.Monitoring

In the previous Sections 2.1 and 2.2, the Kieker installation and the example application have been introduced. In this section, the preparations for application monitoring, the instrumentation of the application, and the actual monitoring are explained.

In this example, the instrumentation is done manually. This means that the monitoring probe is implemented by mixing monitoring logic with business logic, which is often not desired since the resulting code is hardly maintainable. Kieker includes probes based on AOP (aspect-oriented programming, [2]) technology, as covered by Chapter 5. However, to illustrate the instrumentation in detail, the quick start example uses manual instrumentation.

The first step is to copy the Kieker jar-file kieker-1.3.jar to the lib/ directory of the example directory (see Section 2.2). The file is located in the kieker-1.3/dist/ directory of the extracted Kieker archive, as described in Section 2.1. The file commons-logging-1.1.1.jar is located in the kieker-1.3/lib/ directory and has to be copied to the lib/ directory of the example application. The final layout of the example directories is illustrated in Figure 2.4.

Kieker maintains monitoring data as so-called monitoring records. Section 3.3 describes how to define and use custom monitoring record types. The monitoring record type used in this example is an operation execution record which is included in the Kieker
distribution. Figure 2.5 shows the attributes which are relevant to this example. The record type will be detailed in Chapter 5.

The attributes relevant to this part are `className` and `operationName` for the class and method name, as well as `tin` and `tout` for the timestamp before and after the call of the instrumented method.

Listing 2.7 shows the instrumentation of the Bookstore class and its method `searchBook()`. In the lines 12 and 13, the monitoring controller is instantiated. It provides the monitoring service for the instrumentation.

```
private final static IMonitoringController MONITORING_CONTROLLER = MonitoringController.getInstance();

public void searchBook() {
    /* 1.) Call the Catalog component’s getBook() method
        * and log its entry and exit timestamp using Kieker. */
    final long tin = Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
    this.catalog.getBook(false);
    final long tout = Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
    final OperationExecutionRecord e =
        new OperationExecutionRecord(
            Catalog.class.getName(), "getBook(..)",
            tin, tout);
    Bookstore.MONITORING_CONTROLLER.newMonitoringRecord(e);
    /* 2.) Call the CRM catalog’s getOffers() method
        * (without monitoring). */
    this.crm.getOffers();
}
```

Listing 2.7: Instrumentation of the `getBook()` call in Bookstore.java
The lines 18 and 20 are used to determine the current time in nanoseconds before and after the `getBook()` call. In lines 21 to 24, a monitoring record for this measurement is created and initialized with the two time values. Additionally, the record has an attribute for the involved class `Catalog` and the called method `getBook()`. Finally the record is handed over to the monitoring controller which calls a monitoring writer to persist the record. In this example, the filesystem writer is used—`Kieker` uses this writer by default when no other writer is specified, as detailed in Section 3.5.

In addition to the instrumentation in the `Bookstore` class, the `getOffers()` method of the `CRM` class is instrumented as well. Similar to Listing 2.7, measurements are taken before and after the call of the `catalog`'s `getBook()` method, as shown in lines 20 and 22 of Listing 2.8. Not shown in the listing is the instantiation of the monitoring controller. However, it is done in the same way as illustrated in Listing 2.7. Finally, a record is created (see lines 23–26) and stored by calling the monitoring controller (see line 27).

```java
17 public void getOffers () {
18     /* Call the Catalog component’s getBook() method */
19     final long tin = CRM.MONITORING_CONTROLLER.getTimeSource().getTime();
20     final long tout = CRM.MONITORING_CONTROLLER.getTimeSource().getTime();
21     final OperationExecutionRecord e =
22         new OperationExecutionRecord(
23             Catalog.class.getName(), "getBook()",
24             tin, tout);
25     CRM.MONITORING_CONTROLLER.newMonitoringRecord(e);
26 }
```

Listing 2.8: Instrumentation of the `getBook()` call in CRM.java

The next step after instrumenting the code is running the instrumented application. Listing 2.9 shows the two commands to compile and run the application under UNIX-like systems. Listing 2.10 shows the same commands for Windows.

```bash
> mkdir build
> javac src/bookstoreApplication/*.java -classpath lib/kieker-1.3.jar -d build/
> java -classpath build:/
lib/kieker-1.3.jar:
lib/commons-logging-1.1.1.jar
bookstoreApplication.BookstoreStarter
```

Listing 2.9: Commands to compile and run the instrumented Bookstore under UNIX-like systems

Under Windows it is necessary to separate the classpath elements by a semicolon instead of a colon.
If everything worked correctly, a new directory for the monitoring data with a name similar to `kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON/` is created (see Figure 2.6). The numbers in the directory name represent the time and date of the monitoring. In Kieker’s default configuration, the log directory can be found in the default temporary directory: under UNIX-like systems, this is typically `/tmp/`; check the environment variable `%temp%` for the location under Windows. The monitoring directory contains two types of files: `.dat` files containing text representations of the monitoring records and a file named `kieker.map` which contains information on the types of monitoring records used.

```
/tmp/
  kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON/
    kieker.map
    kieker-20110427-142244920-UTC-Thread-1.dat
```

Figure 2.6: Directory structure after a monitoring run

The Listings 2.11 and 2.12 show example file contents. The `.dat`-file is saved in CSV format (Comma Separated Values)—in this case, the values of a monitoring record are separated by semicolons. To understand the `.dat`-file structure the semantics have to be explained. For this quick start example only some of the values are relevant. The first value `$1$` indicates the record type. The forth value indicates the class and method which has been called. And the seventh and eighth value are the start and end time of the execution of the called method.

```
$1;1303914164918306848;−1;bookstoreApplication.Catalog.getBook(..);N/A;−1
 ;13039141649155013739;13039141649171249777;N/A;−1:−1
$1;13039141649205444967;−1;bookstoreApplication.Catalog.getBook();N/A;−1
 ;13039141649184125684;13039141649205064469;N/A;−1:−1
```

Listing 2.11: kieker-20110427-142244920-UTC-Thread-1.dat (excerpt)

The second file is a simple mapping file referencing keys to monitoring record types. In Listing 2.12 the key `$1$` is mapped to the type of operation execution records which were used in the monitoring. The key value corresponds to the key values in the `.dat`-file.

```
$1=kieker.common.record.OperationExecutionRecord
```

Listing 2.12: kieker.map
By the end of this section, two Java classes of the Bookstore application have been manually instrumented using Kieker.Monitoring and at least one run of the instrumented application has been performed. The resulting monitoring log, written to the .dat-file in CSV format, could already be used for analysis or visualization by any spreadsheet or statistical tool. The following Section 2.4 will show how to process this monitoring data with Kieker.Analysis.

2.4 Analysis with Kieker.Analysis

In this section, the monitoring data recorded in the previous section is analyzed with Kieker.Analysis. For this quick example guide, the analysis tool is very simple and does not show the full potential of Kieker. For more detail read Chapter 5 which uses Kieker.TraceAnalysis.

The analysis application developed in this section comprises the files Consumer.java and BookstoreAnalysisStarter.java, as shown in Figure 2.7. These files can also be found in the directory examples/userguide/ch2-manual-instrumentation/

Listing 2.13 on page 16 shows the content of the Consumer.java file which implements the IMonitoringRecordConsumerPlugin interface. This is the standard interface for the consumer of Kieker monitoring records. The consumer is part of the Kieker.Analysis component. It processes records provided by the monitoring readers (see Chapter 1). The consumer checks if the response time of each method call, in this case getBook(), is below a specified threshold value. This threshold is set during construction of the Consumer class (see lines 13–15).
package bookstoreApplication;

import java.util.Collection;

import kieker.analysis.plugin.IMonitoringRecordConsumerPlugin;
import kieker.common.record.IMonitoringRecord;
import kieker.common.record.OperationExecutionRecord;

public class Consumer implements IMonitoringRecordConsumerPlugin {

    private long maxResponseTime;

    public Consumer(long maxResponseTime) {
        this.maxResponseTime = maxResponseTime;
    }

    @Override
    public Collection<Class<? extends IMonitoringRecord>> getRecordTypeSubscriptionList() {
        return null;
    }

    @Override
    public Collection newMonitoringRecord(IMonitoringRecord record) {
        if (!(record instanceof OperationExecutionRecord)) {
            return true;
        }
        OperationExecutionRecord rec = (OperationExecutionRecord) record;
        long responseTime = rec.tout - rec.tin;
        if (responseTime > maxResponseTime) {
            System.err.println("maximum response time exceeded by " + (responseTime - maxResponseTime) + " ns: " + rec.className + "." + rec.operationName);
        } else {
            System.out.println("response time accepted: " + rec.className + "." + rec.operationName);
        }
        return true;
    }

    @Override
    public boolean execute() {
        return true;
    }

    @Override
    public void terminate(boolean error) {
    }
}

Listing 2.13: Consumer.java
For the data analysis, the method \texttt{newMonitoringRecord()} (see lines 23–40) is used. This method is called for every monitoring record. At first, the method tests if the monitoring record is an \texttt{OperationExecutionRecord}, as this is the only record type it can process. Then the method calculates the response time of one recorded \texttt{getBook()} call. If the method call takes longer than specified, a message is written directly to the error stream.

The framework methods \texttt{terminate} and \texttt{execute} don’t do anything in this example, due to the fact that this consumer does not need any initialization. If the consumer would, for example, use threads then these methods would be the correct location to start and stop them.

After implementing a consumer, the application’s main class has to be created. In this case the main program is located in the \texttt{BookstoreAnalysisStarter.java} file shown in Listing \ref{lst:bookstoreAnalysisStarter}.

```java
package bookstoreApplication;

import kieker.analysis.AnalysisController;
import kieker.analysis.plugin.MonitoringRecordConsumerException;
import kieker.analysis.reader.MonitoringReaderException;
import kieker.analysis.reader.filesystem.FSReader;

public class BookstoreAnalysisStarter {

    public static void main(final String[] args)
            throws MonitoringReaderException, MonitoringRecordConsumerException {

        if (args.length == 0) {
            return;
        }

        /* Create Kieker.Analysis instance */
        final AnalysisController analysisInstance = new AnalysisController();
        /* Register our own consumer; set the max. response time to 1.9 ms */
        analysisInstance.registerPlugin(new Consumer(1900000));

        /* Set filesystem monitoring log input directory for our analysis */
        final String inputDirs[] = {args[0]};
        analysisInstance.setReader(new FSReader(inputDirs));

        /* Start the analysis */
        analysisInstance.run();
    }
}
```

Listing 2.14: BookstoreAnalysisStarter.java

The \texttt{BookstoreAnalysisStarter} follows a simple scheme. Each analysis tool has to create at least one \texttt{AnalysisController} which can be seen in Listing \ref{lst:bookstoreAnalysisStarter} in line 18.
Then the consumers are registered with the analysis instance. In this case, the previously
described Consumer is instantiated and the maximum response time is set to 1.9
milliseconds. Line 24 sets the file system monitoring log reader which is initialized with
the first command-line argument value as the input directory. The application expects
the output directory from the earlier monitoring run (see Section 2.3) as the only argu-
ment value, which must be passed manually. The analysis is started by calling its run
method (line 27).

The Listings 2.15 and 2.16 describe how the analysis application can be compiled and
executed under UNIX-like systems and Windows.

```bash
$ mkdir build
$ javac src/bookstoreApplication/*.java
   -classpath lib/kieker-1.3.jar
   -d build/

$ java -classpath build:/
   lib/kieker-1.3.jar:
   lib/commons-logging-1.1.1.jar
   bookstoreApplication.BookstoreAnalysisStarter
   /tmp/kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON
```

Listing 2.15: Commands to compile and run the analysis under UNIX-like systems

```bash
$ mkdir build
$ javac src\bookstoreApplication\*.java
   -classpath lib\kieker-1.3.jar
   -d build\n
$ java -classpath build:
   build:\:
   lib\kieker-1.3.jar:
   lib\commons-logging-1.1.1.jar
   bookstoreApplication.BookstoreAnalysisStarter
   C:\Temp\kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON
```

Listing 2.16: Commands to compile and run the analysis under Windows

You need to make sure that the application gets the correct path from the monitoring
run. The consumer prints an output message for each record received. An example
output can be found in Appendix D.1.
3 Kieker.Monitoring Component

The Java sources of this chapter can be found in the examples/userguide/ch3-4-custom-components/ directory of the binary release.

3.1 Monitoring Controller

The MonitoringController constructs and controls a Kieker.Monitoring instance. As depicted by the class diagram in Figure 3.1, it provides methods for

- Creating IMonitoringController instances (Section 3.1.1),
- Logging monitoring records employing the configured monitoring writer (Section 3.1.2),
- Retrieving the current time via the configured time source (Section 3.1.3),
- Scheduling and removing period samplers (Section 3.1.4), and
- Controlling the monitoring state (Section 3.1.5).

3.1.1 Creating MonitoringController Instances

The MonitoringController provides two different static methods for retrieving instances of IMonitoringController:

1. The method MonitoringController.getInstance() returns a singleton instance. As described in Section 3.2, the configuration is read from a properties file that has been passed to the JVM, is located in the classpath, or conforms to the default configuration (Appendix B).

![Class diagram of the MonitoringController](image)

Figure 3.1: Class diagram of the MonitoringController (including selected methods)
2. The method `MonitoringController.createInstance(Configuration config)` can be used to create an instance that is configured according to the passed `Configuration` object, as described in Section 3.2.

### 3.1.2 Logging Monitoring Records

Monitoring records are sent to the configured monitoring writers by passing these records, in form of `IMonitoringRecord` objects, to the `MonitoringController`'s `newMonitoringRecord` method. Note, that this is not the case if monitoring is disabled or terminated (Section 3.1.5).

### 3.1.3 Retrieving the Current Time and Using Custom Time Sources

The current time is maintained by a so-called time source. The `MonitoringController`'s method `getTimeSource` returns an `ITimeSource` object which provides a method `getTime`. Kieker’s default time source (`DefaultSystemTimer`) returns the current system time as the number of nanoseconds elapsed since 1 Jan 1970 00:00 UTC. The easiest way to use a custom time source is to extend the `AbstractTimeSource` and to implement the method `getTime()`. Custom time sources make sense, for instance, in simulations where not the current system time but the current simulation time is relevant. The configuration needs to be adjusted to use a custom time source class.

### 3.1.4 Scheduling and Removing Periodic Samplers

For certain applications, it is required to monitor runtime data periodically, e.g., the utilization of system resources such as CPUs. For this purpose, Kieker supports special monitoring probes, called samplers. Samplers must implement the interface `ISampler` which includes a single method `sample(IMonitoringController monitoringController)`. This method is called in periodic time steps, as specified by the `MonitoringController`'s registration function `schedulePeriodicSampler`. Periodic samplers can be stopped by calling the `MonitoringController`'s method `removeScheduledSampler`.

Listing 3.1 shows the `sample` method of the `MemSwapUsageSampler` which can be used to monitor memory and swap usage employing the Sigar library. Likewise to other monitoring probes described in this user guide (see for example Sections 3.4 and 2.3), it collects the data of interest (lines 39–40), creates a monitoring record (lines 41–46), and passes this monitoring record to the monitoring controller (line 47). The available Sigar-based samplers for monitoring system-level monitoring data, such as CPU and memory usage, are discussed in Appendix H.
public void sample(final IMonitoringController monitoringController) throws Exception {
    final Mem mem = this.sigar.getMem();
    final Swap swap = this.sigar.getSwap();
    final MemSwapUsageRecord r =
        new MemSwapUsageRecord(monitoringController.getTimeSource() 
            .getTime(),
            monitoringController.getHost(), mem.getTotal(),
            mem.getActualUsed(), mem.getActualFree(),
            swap.getTotal(), swap.getUsed(), swap.getTotal());
    monitoringController.newMonitoringRecord(r);
}
3.2 Kieker.Monitoring Configuration

Kieker.Monitoring instances can be configured by properties files, Configuration objects, and by passing property values as JVM arguments. If no configuration is specified, a default configuration is being used. Appendix B lists this default configuration with a documentation of all available properties. The default configuration properties file, which can be used as a template for custom configurations, is provided by the file kieker-monitoring.properties in the directory kieker-1.3/META-INF/ of the binary release (see Section 2.1).

Configurations for Singleton Instances

In order to use a custom configuration file, its location needs to be passed to the JVM using the parameter kieker.monitoring.configuration as follows:

```
java -Dkieker.monitoring.configuration=<ANY−DIR>/my.kieker.monitoring.properties [...]
```

Alternatively, a file named kieker.monitoring.properties can be placed in a directory called META-INF/ located in the classpath. The available configuration properties can also be passed as JVM arguments, e.g., `-Dkieker.monitoring.enabled=true`.

---

Figure 3.2: Screenshots of the jconsole JMX client accessing the MonitoringController’s attributes and operations via the MBean interface.
Configurations for Non-Singleton Instances

The class Configuration provides factory methods to create Configuration objects according to the default configuration or loaded from a specified properties file: createDefaultConfiguration, createConfigurationFromFile, and createSingletonConfiguration. Note, that JVM parameters are only evaluated when using the factory method createSingletonConfiguration. The returned Configuration objects can be adjusted by setting single property values using the method setProperty.

3.3 Monitoring Records

Monitoring records are objects that contain the monitoring data, as mentioned in the previous chapters. Typically, an instance of a monitoring record is constructed in a monitoring probe (Section 3.4), passed to the monitoring controller (Section 3.1), serialized and deserialized by a monitoring writer (Section 3.5) and a monitoring reader (Section 4.2), and provided to the analysis plugins (Section 4.4) by the analysis controller (Section 4.1). Figure 1.2 illustrates this life cycle of a monitoring record.

In Chapter 2, we’ve already introduced and used the monitoring record type OperationExecutionRecord. Kieker allows to use custom monitoring record types. Corresponding classes must implement the interface IMonitoringRecord shown in Figure 3.3. The methods initFromArray, toArray, getValueTypes are used for serialization and deserialization of the monitoring data contained in the record. The method getLoggingTimestamp is used by the monitoring controller to store the date and time when a record is received by the controller. The method getLoggingTimestamp can be used during analysis to retrieve this value. Kieker.Monitoring provides the abstract class AbstractMonitoringRecord (Figure 3.3) which already implements the methods to maintain the logging timestamp.

```
+ className : String
+ methodName : String
+ responseTimeNanos : long
+ initFromArray ( values : Object[*] )
+ toArray ( ) : Object[*]
+ getValueTypes ( ) : Class[*]
```

Figure 3.3: Class diagram with the IMonitoringRecord interface, the abstract class AbstractMonitoringRecord, and a custom monitoring record type MyResponseTimeRecord
Employing the abstract class for implementing your own monitoring record type, you need to:

1. Create a class that extends `AbstractMonitoringRecord` and
2. Override the methods `initFromArray`, `toArray`, `getValueTypes`.

The class `MyResponseTimeRecord`, shown in the class diagram in Figure 3.3 and in Listing 3.2 is an example of a custom monitoring record type that can be used to monitor response times of method executions.

```java
package bookstoreApplication;
import kieker.common.record.AbstractMonitoringRecord;

public class MyResponseTimeRecord extends AbstractMonitoringRecord {
    private static final long serialVersionUID = 1775L;
    private final static String NA_VAL = "N/A";
    public volatile String className = MyResponseTimeRecord.NA_VAL;
    public volatile String methodName = MyResponseTimeRecord.NA_VAL;
    public volatile long responseTimeNanos = -1;

    @Override
    public final void initFromArray(final Object[] values) {
        this.className = (String) values[0];
        this.methodName = (String) values[1];
        this.responseTimeNanos = (Long) values[2];
    }

    @Override
    public final Object[] toArray() {
        return new Object[]{this.className, this.methodName, this.responseTimeNanos};
    }

    @Override
    public final Class<?>[] getValueTypes() {
        return new Class[]{String.class, String.class, long.class};
    }
}
```

Listing 3.2: MyResponseTimeRecord.java
3.4 Monitoring Probes

The probes are responsible for collecting the monitoring data and passing this monitoring data to the monitoring controller. In Chapter 2.3, we have already demonstrated how to manually instrument a Java application. Listing 3.3 shows a similar manual monitoring probe which uses the monitoring record type `MyResponseTimeRecord` defined in the previous Section 3.3.

```
final long tin = Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
this.catalog.getBook(false);
final long tout = Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
/* Create a new record and set values */
final MyResponseTimeRecord e = new MyResponseTimeRecord();
e.className = "mySimpleKiekerExample.bookstoreTracing.Catalog";
e.methodName = "getBook(..)";
e.responseTimeNanos = tout - tin;
/* Pass the record to the monitoring controller */
Bookstore.MONITORING_CONTROLLER.newMonitoringRecord(e);
```

Listing 3.3: Excerpt from Bookstore.java

In order to avoid multiple calls to the `getInstance` method of the `MonitoringController` class, singleton instances should be stored in a final static variable, as shown in Listing 3.4.

```
private final static IMonitoringController MONITORING_CONTROLLER =
    MonitoringController.getInstance();
```

Listing 3.4: Singleton instance of the monitoring controller stored in a final static variable (excerpt from Bookstore.java)

When manually instrumenting an application, the monitoring probe is implemented by mixing monitoring logic with business logic, which is often not desired since the resulting code is hardly maintainable. Many middleware technologies, such as Java EE Servlet [4], Spring [5], and Apache CXF [6] provide interception/AOP [2] interfaces which are well-suited to implement monitoring probes. AspectJ [8] allows to instrument Java applications without source code modifications. Chapter 5 describes the Kieker probes based on these technologies allowing to monitor trace information in distributed applications.

3.5 Monitoring Writers

Monitoring log writers serialize monitoring records to the monitoring log and must implement the interface `IMonitoringWriter`. The monitoring controller passes the received
records to the writer by calling the method \textit{newMonitoringRecord}. Writers can use the methods to serialize the record contents, as described in Section 3.3.

Figure 3.4 shows the monitoring writers already implemented in Kieker.Monitoring. The writers \texttt{AsyncFsWriter}, \texttt{SyncFsWriter}, \texttt{AsyncDbWriter}, and \texttt{SyncDbWriter} can be used to store monitoring records to filesystems and databases respectively. The variants with the prefix \texttt{Async} are implemented using asynchronous threads that decouple the I/O operations from the control flow of the instrumented application. The \texttt{AsyncFsWriter} is the default writer which has already been used in Section 2.3. Currently, the database writer only supports the record type \texttt{OperationExecutionRecord}.

The \texttt{AsyncJMSWriter} writes records to a JMS (Java Messaging Service \cite{JavaMessaging}) queue. This allows to implement on-the-fly analysis in distributed systems, i.e., analysis while continuously receiving new monitoring data from an instrumented application potentially running on another machine. A brief description of how to use the \texttt{AsyncJMSWriter} can be found in Appendix C.

Listing 3.5 on page 27 shows a custom writer \texttt{MyPipeWriter} which uses a named pipe to write the given records into a buffer located in the memory. The source code of the class \texttt{MyPipe} is listed in Appendix C.1.

The monitoring writer to be used is selected and configured by the Kieker.Monitoring configuration properties (Section 3) \texttt{monitoringDataWriter} and \texttt{monitoringDataWriterInitString}. Listing 3.5 demonstrates how to use the custom writer \texttt{MyPipeWriter} defined above. In this example, the pipe name is passed as the property value \texttt{monitoringDataWriterInitString}.

\begin{verbatim}
monitoringDataWriter=bookstoreApplication.MyPipeWriter
monitoringDataWriterInitString =pipeName=somePipe
\end{verbatim}
```java
package bookstoreApplication;

import java.util.Properties;

import kieker.common.record.IMonitoringRecord;
import kieker.monitoring.core.configuration.Configuration;
import kieker.monitoring.writer.AbstractMonitoringWriter;

public class MyPipeWriter extends AbstractMonitoringWriter {

    private static final String PREFIX = MyPipeWriter.class.getName() + ".";
    private static final String PROPERTY_PIPE_NAME = MyPipeWriter.PREFIX + "pipeName";
    private volatile String pipeName;
    private volatile MyPipe pipe;

    public MyPipeWriter(final Configuration configuration) {
        super(configuration);
    }

    @Override
    public boolean newMonitoringRecord(final IMonitoringRecord record) {
        try {
            /* Just write the content of the record into the pipe. */
            this.pipe.put(new PipeData(record.getLoggingTimestamp(), record.toArray()));
        } catch (final InterruptedException e) {
            return false; // signal error
        }
        return true;
    }

    @Override
    protected Properties getDefaultProperties() {
        final Properties properties = new Properties(super.getDefaultProperties());
        properties.setProperty(MyPipeWriter.PROPERTY_PIPE_NAME, "myPipeName");
        return properties;
    }

    @Override
    protected void init() throws Exception {
        this.pipeName = this.configuration
            .getStringProperty(MyPipeWriter.PROPERTY_PIPE_NAME);
        this.pipe = MyNamedPipeManager.getInstance().acquirePipe(this.pipeName);
    }

    public void terminate() {
    }

    Listing 3.5: MyWriter.java
```
As the data structure of this kind of monitoring log, we created a class PipeData in order to demonstrate the use of the toArray and initFromArray (in Section 4.2) methods. A PipeData object holds a logging timestamp and an Object array containing the serialized record data. Appendix C.1 includes a source code listing of this class. Alternatively, we could have used IMonitoringRecord as the data structure used by the pipe. This is the way, Kieker's PipeWriter works.
4 Kieker.Analysis Component

The Java sources of this chapter can also be found in the examples/userguide/ch3-4-custom-components/ directory of the binary release.

4.1 Analysis Controller

An analysis with Kieker.Analysis is set up and executed employing the class AnalysisController. Kieker.Analysis requires a monitoring reader (Section 4.2) and at least one monitoring record consumer plugin (Section 4.4). In addition to the monitoring record consumer plugin, other analysis plugins can be registered. Figure 4.1 shows the class diagram with the important Kieker.Analysis classes and their relationship.

Figure 4.1: Class diagram showing important Kieker.Analysis classes and their relationship

Setting up and running an analysis with Kieker.Analysis requires the following steps to be performed, as described in Section 2.4 already:

1. Creating an instance of the AnalysisController class
2. Creating and registering the monitoring reader (setReader) as well as the monitoring record consumers and other analysis plugins (registerPlugin).
3. Starting the analysis instance (run).
In the following Sections 4.2 and 4.4, we will create a custom monitoring reader MyPipeReader and a monitoring record consumer plugin MyResponseTimeConsumer. The following Listing 4.1 shows how to create and run an analysis with these custom components:

```java
26 final AnalysisController analysisController = new AnalysisController();
27 final IMonitoringReader reader =
28 new MyPipeReader("somePipe");
29 final IMonitoringRecordConsumerPlugin consumer =
30 new MyResponseTimeConsumer();
31 analysisController .setReader(reader);
32 analysisController .registerPlugin (consumer);
33 analysisController .run();
```

Listing 4.1: Code snippet setting up and running a Kieker.Analysis instance (Starter.java)

On invocation of the `run` method, the `Analysis Controller` calls the `execute` method of all analysis plugins allowing them to initialize. Then, it starts the configured monitoring reader by calling its `read` method. Monitoring record consumers receive the monitoring records provided by the reader. As soon as the reader returns from the execution of its `read` method, the method `terminate` of each registered plugin is called by the `Analysis Controller`.

### 4.2 Monitoring Readers

The monitoring readers are the direct counterpart to the monitoring writers. While writers receive records and write them into files or other kinds of monitoring logs/streams, readers deserialize monitoring data and provide it as `IMonitoringRecord` instances.

Figure 4.2: Interface `IMonitoringReader` and implementing classes
There are already some readers implemented in Kieker as shown in the class diagram in Figure 4.2. The FSReader has already been used in Section 2.4. The FSReaderRealtime can be used to simulate continuous monitoring of a production system. It adds delays between the delivery of the monitoring records to its consumers according to the original delays reconstructed from the logging timestamps (Section 3.3). A brief description of how to use the JMSReader can be found in Appendix G.

The implementation of a custom reader is similar to implementing a monitoring writer. Custom reader should extend the class AbstractKiekerMonitoringReader which already provides an implementation of the observer pattern. By invoking the method deliverRecord, the delivery of records is then delegated to the super class.

Listing 4.2 on page 32 shows a simple reader which polls records from the named pipe introduced in the previous Chapter 3.

4.3 Analysis Plugins

Any analysis or visualization component used with Kieker.Analysis must implement the interface IAnalysisPlugin (Figure 4.1). As described in Section 4.1, the life-cycle of each registered plugin is controlled by the Analysis Controller instance employing the methods execute and terminate. Analysis plugins must implement these methods for initialization and termination.

The monitoring record consumer plugins described in the following Section 4.4, are special analysis plugins that receive the monitoring records provided by the monitoring log reader. Starting with these monitoring record plugins, analysis plugins can be connected in a pipe-and-filter style to implement more complex analyses. Kieker provides input and output port interfaces and implementing classes to implement such analyses. See the documentation of the classes AbstractInputPort and OutputPort for details. Kieker.TraceAnalysis is implemented based on this pattern.

4.4 Monitoring Record Consumer Plugins

As just mentioned, consumer plugins are special analysis plugins which receive the records provided by the monitoring log reader and implement analyses or visualizations based on these records. Consumer plugins must implement the interface IMonitoringRecordConsumerPlugin (see Figure 4.1). By implementing the getRecordTypeSubscriptionList method, a consumer plugin can specify the desired types of monitoring records to be received via the method newMonitoringRecord.

The custom consumer in Listing 4.3 on page 33 simply writes the content of the received response time records to the standard output stream.
public class MyPipeReader extends AbstractMonitoringReader {

    private static final Log log = LogFactory.getLog(MyPipeReader.class);
    private static final String PROPERTY_PIPE_NAME = "pipeName";
    private volatile MyPipe pipe;
    
    public MyPipeReader () {
    }
    
    public MyPipeReader (final String pipeName) {
        this.init(MyPipeReader.PROPERTY_PIPE_NAME+"="+pipeName);
    }
    
    @Override
    public boolean init (final String initString ) throws IllegalArgumentException {
        try {
            final PropertyMap propertyMap = new PropertyMap(initString, "|", "=");
            final String pipeName = propertyMap.getProperty(MyPipeReader.PROPERTY_PIPE_NAME);
            this.pipe = MyNamedPipeManager.getInstance().acquirePipe(pipeName);
        } catch (final Exception exc) {
            MyPipeReader.log.error("Failed to parse initString '" + initString "+" + exc.getMessage());
            return false;
        }
        return true;
    }
    
    @Override
    public boolean read() {
        try {
            PipeData data;
            /* Wait max. 4 seconds for the next data. */
            while ((data = this.pipe.poll(4)) != null) {
                /* Create new record, init from received array ... */
                final MyResponseTimeRecord record = new MyResponseTimeRecord();
                record.initFromArray(data.getRecordData());
                record.setLoggingTimestamp(data.getLoggingTimestamp());
                /* ... and delegate the task of delivering to the super class. */
                this.deliverRecord (record);
            }
        } catch (final InterruptedException e) {
            return false; // signal error
        }
        return true;
    }
}

Listing 4.2: MyPipeReader.java (excerpt)
package bookstoreApplication;

import java.util.Collection;

import kieker.analysis.plugin.IMonitoringRecordConsumerPlugin;
import kieker.common.record.IMonitoringRecord;

public class MyResponseTimeConsumer implements IMonitoringRecordConsumerPlugin {

    @Override
    public Collection<? extends IMonitoringRecord> getRecordTypeSubscriptionList() {
        return null;
    }

    @Override
    public boolean newMonitoringRecord(IMonitoringRecord record) {
        if (record instanceof MyResponseTimeRecord) {
            MyResponseTimeRecord myRecord = (MyResponseTimeRecord) record;
            System.out.println("[Consumer] " + myRecord.getLoggingTimestamp() + ": " + myRecord.className + ", " + myRecord.methodName + ", " + myRecord.responseTimeNanos);
        }
        return true;
    }

    @Override
    public boolean execute() {
        return true;
    }

    @Override
    public void terminate(boolean error) {
    }

    public Collection getRecordTypeSubscriptionList() {
        return null;
    }

    public boolean newMonitoringRecord(IMonitoringRecord record) {
        if (record instanceof MyResponseTimeRecord) {
            MyResponseTimeRecord myRecord = (MyResponseTimeRecord) record;
            System.out.println("[Consumer] " + myRecord.getLoggingTimestamp() + ": " + myRecord.className + ", " + myRecord.methodName + ", " + myRecord.responseTimeNanos);
        }
        return true;
    }

    public boolean execute() {
        return true;
    }

    public void terminate(boolean error) {
    }
}

Listing 4.3: MyResponseTimeConsumer.java
5 Kieker.TraceAnalysis Tool

Kieker.TraceAnalysis implements the special feature of Kieker allowing to monitor, analyze and visualize (distributed) traces of method executions and corresponding timing information. For this purpose, it includes monitoring probes employing AspectJ [8], Java EE Servlet [9], Spring [5], and Apache CXF [6] technology. Moreover, it allows to reconstruct and visualize architectural models of the monitored systems, e.g., as sequence and dependency diagrams.

Section 2 already introduced parts of the monitoring record type OperationExecutionRecord. Kieker.TraceAnalysis uses this record type to represent monitored executions and associated trace and session information. Figure 5.1 shows a class diagram with all attributes of the record type OperationExecutionRecord. The attributes className, operationName, tin, and tout have been introduced before. The attributes traceId and sessionId are used to store trace and session information; eoi and ess contain control-flow information needed to reconstruct traces from monitoring data. For details on this, please refer to our technical report [9].

![Class diagram of the operation execution record](image)

Figure 5.1: The class diagram of the operation execution record

Section 5.1 describes how to instrument Java applications for monitoring trace information. It presents the technology-specific probes provided by Kieker for this purpose—with a focus on AspectJ. Additional technology-specific probes can be implemented based on the existing probes. Section 5.2 presents the tool which can be used to analyze and visualize the recorded trace data. Examples for the available analysis and visualization outputs provided by Kieker.TraceAnalysis are presented in Section 5.3.
5.1 Monitoring Trace Information

The following Sections describe how to use the monitoring probes based on AspectJ (Section 5.1.1), the Java Servlet API (Section 5.1.2), the Spring Framework (Section 5.1.3), and Apache CXF (Section 5.1.4) provided by Kieker.

5.1.1 AspectJ-Based Instrumentation

AspectJ [8] allows to weave code into the byte code of Java applications and libraries without requiring manual modifications of the source code. Kieker includes the AspectJ-based monitoring probes OperationExecutionAspectAnnotation, OperationExecutionAspectAnnotationServlet, OperationExecutionAspectFull, and OperationExecutionAspectFullServlet which can be woven into Java applications at compile time and load time. These probes monitor method executions and corresponding trace and timing information. The probes with the postfix Servlet additionally store a session identifier within the OperationExecutionRecord. When the probes with name element Annotation are used, methods to be monitored must be annotated by the Kieker annotation @OperationExecutionMonitoringProbe. This section demonstrates how to use the AspectJ-based probes to monitor traces based on the Bookstore application from Chapter 2.

The Java sources of the AspectJ example presented in this section can be found in the examples/userguide/ch5-trace-monitoring-aspectj/ directory of the binary release.

Figure 5.2: The new directory structure of the Bookstore application
Figure 5.2 shows the directory used by the example of this section. The jar-file \texttt{aspectjweaver-1.6.11.jar} is included in the \texttt{lib/} directory of the Kieker binary release. Its \texttt{META-INF/} directory contains an example \texttt{aop.xml}. The library \texttt{aspectjweaver-1.6.11.jar} contains the *AspectJ weaver* which is registered with the JVM and weaves the monitoring instrumentation into the Java classes. It will be configured based on the configuration file \texttt{aop.xml}.

Once the necessary files have been copied to the example directory, the source code can be instrumented with the annotation \texttt{OperationExecutionMonitoringProbe}. Listing 5.1 shows how the annotation is used.

```java
package bookstoreTracing;

import kieker.monitoring.annotation.OperationExecutionMonitoringProbe;

public class Bookstore {
    private final Catalog catalog = new Catalog();
    private final CRM crm = new CRM(catalog);

    @OperationExecutionMonitoringProbe
    public void searchBook() {
        catalog.getBook(false);
        crm.getOffers();
    }
}
```

Listing 5.1: Bookstore.java

As a first example, each method of the Bookstore application will be annotated. The annotation can be used to instrumented all methods except for constructors.

The \texttt{aop.xml} file has to be modified to specify the classes to be considered for instrumentation by the AspectJ weaver. Listing 5.2 shows the modified configuration file.

```xml
<DOCTYPE aspectj PUBLIC "−//AspectJ//DTD//EN" "http://www.aspectj.org/dtd/aspectj_1_5_0.dtd">

<aspectj>
    <weaver options=""/>
    <include within="bookstoreTracing..+"/>
</weaver>

<aspects>
    <aspect name="kieker.monitoring.probe.aspectJ.operationExecution.OperationExecutionAspectAnnotation"/>
</aspects>

Listing 5.2: aop.xml
Line 5 tells the AspectJ weaver to consider all classes inside the package `bookstore-Tracing`. AspectJ allows to use wild-cards for the definition of classes to include—e.g., `<include within="bookstoreTracing.Bookstore"/>` to weave all classes in this package with the prefix `Bookstore`.

Line 9 specifies the aspect to be woven into the classes. In this case, the `Kieker` probe `OperationExecutionAspectAnnotation` is used. It requires that methods intended to be instrumented are annotated by `@OperationExecutionMonitoringProbe`, as mentioned before.

Listings 5.4 and 5.3 show how to compile and run the annotated Bookstore application. The `aop.xml` must be located in a `META-INF/` directory in the classpath—in this case the `build/` directory. The AspectJ weaver has to be loaded as a so-called Java-agent. It weaves the monitoring aspect into the byte code of the Bookstore application. Additionally, a `kieker.monitoring.properties` is copied to the `META-INF/` directory. The configuration may be adjusted as desired (see Section 3.2).

After a complete run of the application, the monitoring files should appear in the same way as mentioned in Section 2.3 including the additional trace information. An example log of a complete run can be found in Appendix D.2.
Instrumentation without annotations  AspectJ-based instrumentation without using annotations is quite simple. It is only necessary to modify the file `aop.xml`, as shown in Listing 5.5.

```xml
<DOCTYPE aspectj PUBLIC "−///AspectJ//DTD//EN" "http://www.aspectj.org/dtd/aspectj_1.5.0.dtd">
<aspectj>
  <weaver options="">
    <include within="bookstoreTracing.BookstoreStarter"/>
  </weaver>

  <aspects>
    <aspect name="kieker.monitoring.probe.aspectJ.operationExecution.OperationExecutionAspectFull"/>
  </aspects>
</aspectj>
```

Listing 5.5: aop.xml

The alternative aspect `OperationExecutionAspectFull` is being activated in line 9. As indicated by its name, this aspect makes sure that every method within the included classes/packages will be instrumented and monitored. Listing 5.5 demonstrates how to limit the instrumented methods to those of the class `BookstoreStarter`.

The commands shown in the Listings 5.4 and 5.3 can again be used to compile and execute the example. Note that the annotations within the source code have no effect when using this aspect.

When using a custom aspect, it can be necessary to specify its class-name in the include directives of the aop.xml.

5.1.2 Servlet Filters

The Java Servlet API [4] includes the `javax.servlet.Filter` interface. It can be used to implement interceptors for incoming HTTP requests. Kieker includes the probes `OperationExecutionRegistrationFilter` and `OperationExecutionRegistrationAndLoggingFilter` which implement the `javax.servlet.Filter` interface. Both initialize the session and trace information for incoming requests. The latter additionally creates an `OperationExecutionRecord` for each invocation of the filter and passes it to the `MonitoringController`.

Listing 5.6 demonstrates how to integrate the `OperationExecutionRegistrationAndLoggingFilter` in the `web.xml` file of a web application.

The Java EE Servlet container example described in Appendix employs the the `OperationExecutionRegistrationAndLoggingFilter`.
5.1.3 Spring

The Spring framework [5] provides interfaces for intercepting Spring services and web requests. Kieker includes the probes OperationExecutionMethodInvocationInterceptor and OperationExecutionWebRequestRegistrationInterceptor. The OperationExecutionMethodInvocationInterceptor is similar to the AspectJ-based probes described in the previous section and monitors method executions as well as corresponding trace and session information. The OperationExecutionWebRequestRegistrationInterceptor intercepts incoming Web requests and initializes the trace and session data for this trace.

See the Spring documentation for instructions how to add the interceptors to the server configuration.

5.1.4 CXF SOAP Interceptors

The Apache CXF framework [6] allows to implement and interceptors for service calls, for example, based on the SOAP web service protocol. Kieker includes the probes OperationExecutionSOAPRequestOutInterceptor, OperationExecutionSOAPRequestInInterceptor, OperationExecutionSOAPResponseOutInterceptor, and OperationExecutionSOAPResponseInInterceptor which can be used to monitor SOAP-based web service calls. Session and trace information is written to and read from the SOAP header of service requests and responses allowing to monitor distributed traces. See the CXF documentation for instructions how to add the interceptors to the server configuration.
5.2 Trace Analysis and Visualization

Monitoring data including trace information can be analyzed and visualized with the Kieker.TraceAnalysis tool which is included in the Kieker binary as well.

In order to use this tool, it is necessary to install two other programs:

1. **Graphviz** A graph visualization software which can be downloaded from http://www.graphviz.org/.

2. **GNU PlotUtils** A set of tools for generating 2D plot graphics which can be downloaded from http://www.gnu.org/software/plotutils/ (for Linux) and from http://gnuwin32.sourceforge.net/packages/plotutils.htm (for Windows).

Under Windows it is recommended to add the `bin/` directories of both tools to the “path” environment variable.

Once both have been installed, the Kieker.TraceAnalysis tool can be used. It can be accessed via the wrapper-script `trace-analysis.sh` or `trace-analysis.bat` (Windows) in the `bin/` directory. Non-parameterized calls of the scripts print all possible options on the screen, as listed in Appendix A.3.

The commands shown in Listings 5.7 and 5.8 generate a sequence diagram as well as a call tree to an existing directory named `out/`. The monitoring data is assumed to be located in the directory `/tmp/kieker-20110428-142829399-UTC-Kaapstad-KIEKER/` or `%temp%\kieker-20100813-121041532-UTC-virus-KIEKER` under Windows.

```
$ ./trace-analysis.sh --inputdirs /tmp/kieker-20110428-142829399-UTC-Kaapstad-KIEKER
   --outputdir out/
   --plot-Allocation-Sequence-Diagrams
   --plot-Call-Trees
```

Listing 5.7: Commands to produce the diagrams under UNIX-like systems

```
> ./trace-analysis.bat --inputdirs %temp%\kieker-20100813-121041532-UTC-virus-KIEKER
   --outputdir out\
   --plot-Allocation-Sequence-Diagrams
   --plot-Call-Trees
```

Listing 5.8: Commands to produce the diagrams under Windows

The resulting contents of the `out/` directory should be similar to the following tree:
The `.pic` and `.dot` files can be converted into other formats, such as `.pdf`, by using the `Graphviz` and `PlotUtils` tools `dot` and `pic2plot`. The following Listing 5.9 demonstrates this.

```
> dot callTree-6120391893596504065.dot -Tpng -o callTree.png
> pic2plot allocationSequenceDiagram-6120391893596504065.pic -Tpng > sequenceDiagram.png
```

Listing 5.9: Commands to convert the diagrams

The scripts `dotPic-fileConverter.sh` and `dotPic-fileConverter.bat` convert all `.pic` and `.dot` in a specified directory. See Appendix A.4 for details.

Examples of all available visualization are presented in the following Section 5.3.
5.3 Example Kieker.TraceAnalysis Outputs

The examples presented in this section were generated based on the monitoring data which can be found in the directory examples/userguide/ch5-trace-monitoring-aspectj/testdata/kieker-20100830-082225522-UTC/. It consists of 1635 traces of the Bookstore application with AspectJ-based instrumentation, as described in Section 5.1.1. In order to illustrate the visualization of distributed traces, the hostname of the Catalog’s method getBook was probabilistically changed to a second hostname. For a more detailed description of the underlying formalisms, we refer to our technical report [9].

5.3.1 Textual Trace and Equivalence Class Representations

Execution Traces

Textual execution trace representations of valid/invalid traces are written to an output file using the command-line options --print-Execution-Traces and --print-invalid-Execution-Traces. Listing 5.10 shows the execution trace representation for the valid trace ...6129.

Listing 5.10: Textual output of trace 6488138950668976129’s execution trace representation

Message Traces

Textual message trace representations of valid traces are written to an output file using the command-line option --print-Message-Traces. Listing 5.11 shows the message trace representation for the valid trace ...6129.

Listing 5.11: Textual output of trace 6488138950668976129’s message trace representation
Listing 5.11: Textual output of trace 6488138950668976129’s message trace representation

For listing 5.11, the section titled "Trace Equivalence Classes" provides detailed information on deployment and assembly-level trace equivalence classes computed and written to output files using command-line options. Listings 5.12 and 5.13 show the output generated for the monitoring data used in this section.

Listing 5.12: Textual output of information on the deployment-level trace equivalence classes

Listing 5.13: Textual output of information on the assembly-level trace equivalence class
5.3.2 Sequence Diagrams

Deployment-Level Sequence Diagrams

Deployment-level sequence diagrams are generated using the command-line option `--plot-Deployment-Sequence-Diagrams`. Figures 5.3(a)–5.3(d) show these sequence diagrams for each deployment-level trace equivalence representative (Section 5.3.1).

Figure 5.3: Deployment-level sequence diagrams of the trace equivalence class representatives (Listing 5.13)
Assembly-Level Sequence Diagrams

Assembly-level sequence diagrams are generated using the command-line option \texttt{- -plot-Assembly-Sequence-Diagrams}. Figure 5.4 shows the sequence diagram for the assembly-level trace equivalence representative (Section 5.3.1).

5.3.3 Call Trees

Trace Call Trees

Trace call trees are generated using the command-line option \texttt{- -plotCallTrees}. Figures 5.5(a)–5.5(d) show these call trees for each deployment-level trace equivalence representative (Section 5.3.1).
Aggregated Call Trees

Aggregated deployment/assembly-level call trees are generated using the command-line options `--plot-Aggregated-Deployment-Call-Tree` and `--plot-Aggregated-Assembly-Call-Tree`. Figures 5.6(a) and 5.6(b) show these aggregated call trees for the traces contained in the monitoring data used in this section.

Figure 5.6: Aggregated call trees generated from the 1635 traces
5.3.4 Dependency Graphs

Container Dependency Graphs

A container dependency graph is generated using the command-line option `-plot-Container-Dependency-Graph`. Figure 5.7 shows the container dependency graph for the monitoring data used in this section.

![Container dependency graph](image)

Figure 5.7: Container dependency graph

Component Dependency Graphs

Deployment/assembly-level component dependency graphs are generated using the command-line options `-plot-Deployment-Component-Dependency-Graph` and `-plot-Assembly-Component-Dependency-Graph`. Figures 5.8(a) and 5.8(b) show the component dependency graphs for the monitoring data used in this section.

![Component dependency graphs](image)

Figure 5.8: Component dependency graphs
Operation Dependency Graphs

Deployment/assembly-level operation dependency graphs are generated using the command-line options `-plot-Deployment-Operation-Dependency-Graph` and `-plot-Assembly-Operation-Dependency-Graph`. Figures 5.9(a) and 5.9(b) show the operation dependency graphs for the monitoring data used in this section.

Figure 5.9: Operation dependency graphs
5.3.5 HTML Output of the System Model

Kieker.TraceAnalysis writes an HTML representation of the system model reconstructed from the trace data to a file `system-entities.html`. Figure 5.10 shows a screenshot of this file rendered by a web browser.

![HTML output of the system model](example.png)

Figure 5.10: HTML output of the system model reconstructed from the traces
A Wrapper scripts

The `bin/` directory of Kieker’s binary release contains some `.sh` and `.bat` scripts to invoke tools included in `kieker-1.3.jar`. The following sections give a short description of their functionality and list their usage outputs as printed to the standard output stream when called without command-line parameters. In addition to the standard output stream, the file `kieker.log` is used for logging output during execution.

A.1 Script `convertLoggingTimestamp.sh|bat`

The script converts Kieker.Monitoring logging timestamps, representing the number of nanoseconds since 1 Jan 1970 00:00 UTC, to a human-readable textual representation in the UTC and local timezones.

Main-class: `kieker.tools.loggingTimestampConverter.LoggingTimestampConverterTool`

Usage

```
usage: kieker.tools.loggingTimestampConverter.LoggingTimestampConverterTool [-t <timestamp1 ... timestampN>]
-t, --timestamps <timestamp1 ... timestampN> List of timestamps (UTC timezone) to convert
```

Example

The following listing shows the command to convert two logging timestamps as well as the resulting output.

```
$ bin/convertLoggingTimestamp.sh --timestamps 1283156545581511026 1283156546127117246
```

A.2 Script `logReplay.sh|bat`

Replays filesystem monitoring logs created by Kieker.Monitoring. Example applications are:

Kieker 1.3 User Guide 50
• Merging multiple directories containing monitoring data into a single output directory.
• Importing a filesystem monitoring log to another monitoring log, e.g., a database. Therefore, an appropriate `Kieker.Monitoring` configuration file must be passed to the script (see Section 3.2).
• Replaying a recorded filesystem monitoring log in real-time in order to simulate incoming monitoring data from a running system, e.g., via JMS (see also Appendix C).

Main-class: `kieker.tools.logReplayer.FilesystemLogReplayerStarter`

Usage

```
usage: kieker.tools.logReplayer.FilesystemLogReplayerStarter
-i, --inputdirs <dir1 ... dirN> Log directories to read data from
--ignore-records-after-date <yyyyMMddHHmmss> Records logged after this date (UTC timezone) are ignored (disabled by default).
--ignore-records-before-date <yyyyMMddHHmmss> Records logged before this date (UTC timezone) are ignored (disabled by default).
-k, --keep-logging-timestamps <true|false> Replay the original logging timestamps (defaults to true)?
-n, --realtime-worker-threads <num> Number of worker threads used in realtime mode (defaults to 1).
-r, --realtime <true|false> Replay log data in realtime?
```

Example

The following command replays the monitoring testdata included in the binary release to another directory:

```
$ bin/logReplay.sh
  -i inputdirs examples/userguide/ch5–trace-monitoring-aspectj/testdata/kieker-20100830-08225522-UTC
  -k keep-logging-timestamps true
  -r realtime false
```

A.3 Script `trace-analysis.sh|bat`

Calls Kieker.TraceAnalysis to analyze and visualize monitored trace data, as described in Chapter 5

Main-class: `kieker.tools.traceAnalysis.TraceAnalysisTool`
Usage

usage: kieker.tools.traceAnalysis.TraceAnalysisTool −i <dir1 ... dirN> −o <dir>−p <prefix>] [--plot−Deployment−Sequence−Diagrams]
               [--plot−Deployment−Component−Dependency−Graph]
               [--plot−Assembly−Component−Dependency−Graph]
               [--plot−Container−Dependency−Graph]
               [--plot−Deployment−Operation−Dependency−Graph]
               [--plot−Assembly−Operation−Dependency−Graph]
               [--plot−Aggregated−Deployment−Call−Tree]
               [--plot−Aggregated−Assembly−Call−Tree]
               [--plot−Message−Traces] [--plot−Execution−Traces]
               [--plot−Deployment−Sequence−Diagrams]
               [--plot−Assembly−Sequence−Diagrams]
               [--plot−Deployment−Operation−Dependency−Graph]
               [--plot−Assembly−Operation−Dependency−Graph]
               [--plot−Aggregated−Deployment−Call−Tree]
               [--plot−Assembly−Deployment−Call−Tree]
               [--plot−Deployment−Sequence−Diagrams]
               [--plot−Assembly−Sequence−Diagrams]
               [--plot−Deployment−Component−Dependency−Graph]
               [--plot−Assembly−Component−Dependency−Graph]
               [--plot−Container−Dependency−Graph]
               [--plot−Deployment−Operation−Dependency−Graph]
               [--plot−Assembly−Operation−Dependency−Graph]
               [--plot−Aggregated−Deployment−Call−Tree]
               [--plot−Aggregated−Assembly−Call−Tree]
               [--plot−Deployment−Sequence−Diagrams]
               [--plot−Deployment−Component−Dependency−Graph]
               [--plot−Deployment−Operation−Dependency−Graph]
               [--plot−Assembly−Operation−Dependency−Graph]
               [--plot−Aggregated−Deployment−Call−Tree]
               [--plot−Aggregated−Assembly−Call−Tree]
               [--plot−Deployment−Sequence−Diagrams]
               [--plot−Deployment−Component−Dependency−Graph]
               [--plot−Deployment−Operation−Dependency−Graph]
               [--plot−Assembly−Operation−Dependency−Graph]
               [--plot−Aggregated−Deployment−Call−Tree]
               [--plot−Aggregated−Assembly−Call−Tree]

−i, −−inputdirs <dir1 ... dirN> Log directories to read data from

−o, −−outputdir <dir> Directory for the generated file(s)

−p, −−output−filename−prefix <prefix> Prefix for output filenames

−−plot−Deployment−Sequence−Diagrams Generate and store deployment−level sequence diagrams (.pic files)

−−plot−Assembly−Sequence−Diagrams Generate and store assembly−level sequence diagrams (.pic files)

−−plot−Deployment−Component−Dependency−Graph Generate and store a deployment−level component dependency graph (.dot file)

−−plot−Assembly−Component−Dependency−Graph Generate and store an assembly−level component dependency graph (.dot file)

−−plot−Container−Dependency−Graph Generate and store a container dependency graph (.dot file)

−−plot−Deployment−Operation−Dependency−Graph Generate and store a deployment−level operation dependency graph (.dot file)

−−plot−Assembly−Operation−Dependency−Graph Generate and store an assembly−level operation dependency graph (.dot file)

−−plot−Aggregated−Deployment−Call−Tree Generate and store an aggregated deployment−level call tree (.dot files)

−−plot−Aggregated−Assembly−Call−Tree Generate and store an aggregated assembly−level call tree
--- plot -- Call -- Trees
(. dot files)
Generate and store call trees for the selected traces (. dot files)

--- print -- Message -- Traces
Save message trace representations of valid traces (. txt files)

--- print -- Execution -- Traces
Save execution trace representations of valid traces (. txt files)

--- print -- invalid -- Execution -- Traces
Save a execution trace representations of invalid trace artifacts (. txt files)

--- print -- Deployment -- Equivalence -- Classes
Output an overview about the deployment--level trace equivalence classes

--- print -- Assembly -- Equivalence -- Classes
Output an overview about the assembly--level trace equivalence classes

--- select -- traces <id0 ... idn>
Consider only the traces identified by the list of trace IDs. Defaults to all traces.

--- ignore -- invalid -- traces
If selected, the execution aborts on the occurrence of an invalid trace.

--- max -- trace -- duration <duration in ms>
Threshold (in milliseconds) after which an incomplete trace becomes invalid. Defaults to infinity.

--- ignore -- executions -- before -- date <yyyyMMdd--HHmmss>
Executions starting before this date (UTC timezone) are ignored.

--- ignore -- executions -- after -- date <yyyyMMdd--HHmmss>
Executions ending after this date (UTC timezone) are ignored.

--- short -- labels
If selected, abbreviated labels (e.g., package names) are used in the visualizations.

Example
The following commands generate a deployment-level operation dependency graph and convert it to pdf format:

```
bin/trace-analysis.sh
  -inputdirs examples/userguide/ch5--trace-monitoring-aspectj/testdata/kieker-20100830-082255522-UTC
  -outputdir .
  -plot -- Deployment -- Operation -- Dependency -- Graph
```

Add additional pdf deploymentOperationDependencyGraph.dot > deploymentOperationDependencyGraph.pdf

Additional examples can be found in Chapter 5.
A.4 Script dotPic-fileConverter.sh|bat

Converts each .dot and .pic file, e.g., diagrams generated by Kieker.TraceAnalysis (Section 5), located in a directory into desired graphic output formats. This script simply calls the Graphviz and PlotUtils tools dot and pic2plot.

Usage

Usage: dotPic−fileConverter.sh <output−directory> <file−type−1 ... file−type−N>

Example

The following command converts each .dot and .pic file located in the directory out/ to files in .pdf and .png format:

```
$ bin/dotPic-fileConverter.sh out/ pdf png
```
B Kieker.Monitoring Default Configuration

This is the file `kieker.monitoring.properties` from the binary release and constitutes Kieker.Monitoring’s default configuration. Section 3.2 describes how to use a custom configuration.

```plaintext
## In order to use a custom Kieker.Monitoring configuration, create a copy of
## this file and modify it according to your needs.

## The location of the file is passed to Kieker.Monitoring via the JVM parameter
## kieker.monitoring.configuration. For example, with a configuration file named
## my.kieker.monitoring.properties in the folder META-INF you would pass this location
## to the JVM when starting your application:
##
## java -Dkieker.monitoring.configuration=META-INF/my.kieker.monitoring.properties [...] 
##
## If no configuration file is passed, Kieker tries to use a configuration file in
## META-INF/kieker.monitoring.properties
##
## If this also fails, a default configuration is being used according to the values in
## this default file.

# The name of the Kieker instance.
kieker.monitoring.name=KIEKER

# The name of the VM running Kieker. If empty the name will be determined
# automatically, else it will be set to the given value.
kieker.monitoring.hostname=

# The initial ID associated with all experiments.
kieker.monitoring.initialExperimentId=1

# Whether the MonitoringController will be available as an MBean.
kieker.monitoring.MBean=false

# These two properties are only evaluated if the MBean is activated.
# They define the ObjectName used to access the MBean (usually you
# don’t have to change them).
kieker.monitoring.MBean.domain=kieker.monitoring
kieker.monitoring.MBean.name=MonitoringController

# Enable/disable monitoring after startup (true|false; default: true)
# If monitoring is disabled, the MonitoringController simply pauses.
# Furthermore, probes should stop collecting new data and monitoring
```
# writers stop should stop writing existing data.
kieker.monitoring.enabled=true

# The Timer used by Kieker. You usually don’t want to change the value.
kieker.monitoring.timer=kieker.monitoring.timer.DefaultSystemTimer
# You can specify additional parameters send to the Timer, e.g.
# kieker.monitoring.timer.DefaultSystemTimer.KEY=VALUE

# The size of the thread pool used to execute registered periodic sensor jobs.
kieker.monitoring.periodicSensorsExecutorPoolSize=1

# Enables/disable the automatic assignment of each record’s logging timestamp
# (true/false; default: true)
kieker.monitoring.setLoggingTimestamp=true

###############################
####### WRITER #######
###############################
## Selection of monitoring data writer (classname)
## The value must be a fully-qualified classname of a class implementing
## kieker.monitoring.IMonitoringWriter and providing a constructor that
## accepts an IMonitoringController and a single Configuration.
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter

###
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.SyncFsWriter
#
### Configuration Properties of the DummyWriter
kieker.monitoring.writer.DummyWriter.key=value

###
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
#
### In order to use the default temporary directory, set the property value of
### storeInJavaIoTmpdir to true.
kieker.monitoring.writer.filesystem.SyncFsWriter.storeInJavaIoTmpdir=true
#
### In order to use a custom directory, set storeInJavaIoTmpdir=false
### and set customStoragePath as desired. Examples:
kieker.monitoring.writer.filesystem.SyncFsWriter.customStoragePath=

###
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
#
### In order to use the default temporary directory, set the property value of
### storeInJavaIoTmpdir to true.
kieker.monitoring.writer.filesystem.AsyncFsWriter.storeInJavaIoTmpdir=true
In order to use a custom directory, set storeInJavaIoTmpdir=false and set customStoragePath as desired. Examples:

```java
kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=
```

Asynchronous writers need to store monitoring records in an internal buffer. This parameter defines its capacity in terms of the number of records.

```java
kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize=10000
```

Behavior of the asynchronous writer when the internal queue is full:

0: terminate Monitoring with an error (default)
1: writer blocks until queue capacity is available
2: writer discards new records until space is available

Be careful when using the value ‘1’ since then, the asynchronous writer is no longer decoupled from the monitored application.

```java
kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior=0
```

---

The name of the pipe used (must not be empty).

```java
kieker.monitoring.writer.namedRecordPipe.PipeWriter.pipeName=kieker−pipe
```

The url of the jndi provider that knows the jms service

```java
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=tcp://127.0.0.1:3035/
```

The topic at the jms server which is used in the publisher/subscribe communication.

```java
kieker.monitoring.writer.jms.AsyncJMSWriter.Topic=queue1
```

The type of the jms factory implementation, e.g. "org.exolab.jms.jndi.InitialContextFactory" for openjms 0.7.7

```java
```

The service name for the jms connection factory.

```java
kieker.monitoring.writer.jms.AsyncJMSWriter.FactoryLookupName=ConnectionFactory
```

The time that a jms message will be kept alive at the jms server before it is automatically deleted.

```java
kieker.monitoring.writer.jms.AsyncJMSWriter.MessageTimeToLive=10000
```

Asynchronous writers need to store monitoring records in an internal buffer. This parameter defines its capacity in terms of the number of records.

```java
kieker.monitoring.writer.jms.AsyncJMSWriter.QueueSize=10000
```
### Behavior of the asynchronous writer when the internal queue is full:

- **0**: terminate Monitoring with an error (default)
- **1**: writer blocks until queue capacity is available
- **2**: writer discards new records until space is available

Be careful when using the value ‘1’ since then, the asynchronous writer is no longer decoupled from the monitored application.

```
kieker.monitoring.writer.jms.AsyncJMSWriter.QueueFullBehavior=0
```

##### kieker.monitoring.writer=kieker.monitoring.writer.database.SyncDbWriter

- **Database driver class name**
  - Examples: `MySQL` -> `com.mysql.jdbc.Driver`

```
```

- **Connection string**
  - Examples:
    - `MySQL`: `jdbc:mysql://HOSTNAME/DBNAME?user=DBUSER&password=DBPASS`
    - `DerbyDB`: `jdbc:derby:DBNAME;user=DBUSER;password=DBPASS`

```
```

- **Name of the database table**
  - (can be generated using the file `table-for-monitoring.sql`)

```
kieker.monitoring.writer.database.SyncDbWriter.TableName=kiekerdata
```

##### kieker.monitoring.writer=kieker.monitoring.writer.database.AsyncDbWriter

- **Database driver class name**
  - Examples: `MySQL` -> `com.mysql.jdbc.Driver`

```
```

- **Connection string**
  - Examples:
    - `MySQL`: `jdbc:mysql://HOSTNAME/DBNAME?user=DBUSER&password=DBPASS`
    - `DerbyDB`: `jdbc:derby:DBNAME;user=DBUSER;password=DBPASS`

```
```

- **Name of the database table**
  - (can be generated using the file `table-for-monitoring.sql`)

```
kieker.monitoring.writer.database.AsyncDbWriter.TableName=kiekerdata
```

- **The number of concurrent Database connections.**

```
kieker.monitoring.writer.database.AsyncDbWriter.numberOfConnections=4
```
### Load the initialExperimentId from the DB and increase it by 1
### instead of using the value from the configuration.
### (Currently not implemented!)

```java
kieker.monitoring.writer.database.AsyncDbWriter.loadInitialExperimentId = false
```

Listing B.1: kieker.monitoring.properties
C Additional Source Code Listings

C.1 MyNamedPipeManager and MyPipe

```java
package bookstoreApplication;

import java.util.HashMap;

public class MyNamedPipeManager {

    private static final MyNamedPipeManager PIPE_MGR_INSTANCE = new MyNamedPipeManager();

    /* Not synchronized! */
    private final HashMap<String, MyPipe> pipeMap = new HashMap<String, MyPipe>();

    public static MyNamedPipeManager getInstance() {
        return MyNamedPipeManager.PIPE_MGR_INSTANCE;
    }

    /*
     * Returns a pipe with name pipeName. If a pipe with this name does not exist prior to the call, it will be created.
     *
     * @param pipeName name of the (new) pipe.
     * @return the pipe
     * @throws IllegalArgumentException if the given name is null or has length zero.
     */
    public synchronized MyPipe acquirePipe(final String pipeName) throws IllegalArgumentException {
        MyPipe conn = this.pipeMap.get(pipeName);
        if (conn == null) {
            conn = new MyPipe(pipeName);
            this.pipeMap.put(pipeName, conn);
        }
        return conn;
    }

    Listing C.1: MyNamedPipeManager.java
```

Kieker 1.3 User Guide
package bookstoreApplication;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class MyPipe {
    private final String pipeName;
    private final LinkedBlockingQueue<PipeData> buffer =
            new LinkedBlockingQueue<PipeData>();

    public MyPipe(final String pipeName) {
        this.pipeName = pipeName;
    }

    public String getPipeName() {
        return this.pipeName;
    }

    public void put(final PipeData data) throws InterruptedException {
        this.buffer.put(data);
    }

    public PipeData poll(final long timeout) throws InterruptedException {
        return this.buffer.poll(timeout, TimeUnit.SECONDS);
    }
}

Listing C.2: MyPipe.java

package bookstoreApplication;

public class PipeData {
    private final long loggingTimestamp;
    private final Object[] recordData;

    public PipeData(final long loggingTimestamp, final Object[] recordData) {
        this.loggingTimestamp = loggingTimestamp;
        this.recordData = recordData;
    }

    public final long getLoggingTimestamp() {
        return this.loggingTimestamp;
    }

    public final Object[] getRecordData() {
        return this.recordData;
    }
}

Listing C.3: PipeData.java
D Example Console Outputs

D.1 Quick Start Example (Chapter 2)

Listing D.1: Execution of the manually instrumented Bookstore application (Section 2.3)
Listing D.2: Execution of the example analysis (Section 2.4)
D.2 Trace Monitoring, Analysis & Visualization (Chapter 5)

Bookstore.main: Starting request 0
Apr 28, 2011 4:28:29 PM kieker.monitoring.core.configuration.Configuration createSingletonConfiguration
INFO: Loading properties from properties file in classpath: 'META-INF/kieker.monitoring.properties'
Apr 28, 2011 4:28:29 PM kieker.monitoring.core.controller.MonitoringController createInstance
INFO: Current State of kieker.monitoring (1.3−20110427) Status: 'enabled'
  Name: 'KIEKER', Hostname: 'Kaapstad', experimentID: '1'
WriterController:
  Number of Inserts: '0'
  Automatic assignment of logging timestamps: 'true'
Writer: 'kieker.monitoring.writer.filesystem.AsyncFsWriter'
  Configuration:
    kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior='0'
    kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize='10000'
    kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=''
    kieker.monitoring.writer.filesystem.AsyncFsWriter.storeInJavaIoTmpdir='true'
Writer Threads (1):
  Finished: 'false'; Writing to Directory: '/tmp/kieker−20110428−142829399−UTC−Kaapstad−KIEKER'
Sampling Controller: Periodic Sensor available: Current Poolsize: '0'; Scheduled Tasks: '0'
Apr 28, 2011 4:28:29 PM kieker.monitoring.core.registry.ControlFlowRegistry <init>
INFO: First threadId will be 7752665283541598209
INFO: Registered monitoring record type with id '1': kieker.common.record.OperationExecutionRecord

Listing D.3: Execution of the Bookstore with AspectJ trace instrumentation (Section 5.1.1)
E Ant Scripts

E.1 Quick Start Example (Chapter 2)

The following build.xml and build.properties files can be used for compiling and executing the manually instrumentated Bookstore application and the analysis, as described in Chapter 2. The files are included in the directory examples/userguide/ch2-manual-instrumentation/.

In order to run the analysis of the application, it is necessary to pass the location of the monitoring log directory. This is done via the parameter analysis.directory, e.g.:

```
> ant run -analysis -Danalysis.directory /tmp/kieker–20110428–142829399–UTC–Kaapstad/
```

Listing E.1: Command to compile and run the instrumented Bookstore via ant

```
<project name="Bookstore Application" basedir="." default="run–monitoring">

  <property file="build.properties" />

  <path id="classpath">
    <filesset dir="${lib.dir}" includes="**/*.jar" />
  </path>

</project>
```

Listing E.2: build.properties

```bash
dir=build
src.dir=src
lib.dir=lib

jar.file.monitoring=BookstoreMonitoring.jar
jar.file.analysis=BookstoreAnalysis.jar
jar.file.commons–logging=commons–logging–1.1.1.jar
jar.file.kieker=kieker–1.3.jar

main=monitoring=bookstoreApplication.BookstoreStarter
main=analysis=bookstoreApplication.BookstoreAnalysisStarter

meta.dir=META–INF

msg.filesNotFound=One or more of the required libraries could not be found. Please add the following files to the ${lib.dir} directory: ${jar.file.commons–logging}, ${jar.file.kieker}.
```

Kieker 1.3 User Guide
<condition property="not—all—files—available">
  <not><and>
    <available file="${lib.dir}/${jar.file.commons.logging}" />
    <available file="${lib.dir}/${jar.file.kieker}" />
  </and></not>
</condition>

<target name="check—files" if="not—all—files—available">
  <fail message="${msg.filesNotFound}" />
</target>

<target name="run—monitoring" depends="check—files, build—jar—monitoring">
  <java fork="true" classname="${main.class—monitoring}">
    <classpath>
      <path refid="classpath" />
      <path location="${jar.file.monitoring}" />
    </classpath>
  </java>
</target>

<target name="run—analysis" depends="check—files, build—jar—analysis">
  <java fork="true" classname="${main.class—analysis}">
    <arg line="${analysis.directory}" />
    <classpath>
      <path refid="classpath" />
      <path location="${jar.file.analysis}" />
    </classpath>
  </java>
</target>

<target name="build—jar—monitoring" depends="compile">
  <jar destfile="${jar.file.monitoring}" basedir="${build.dir}" >
    <manifest>
      <attribute name="Main-Class" value="${main.class—monitoring}" />
    </manifest>
  </jar>
  <delete dir="${build.dir}" />
</target>

<target name="build—jar—analysis" depends="compile">
  <jar destfile="${jar.file.analysis}" basedir="${build.dir}" >
    <manifest>
      <attribute name="Main-Class" value="${main.class—analysis}" />
    </manifest>
  </jar>
  <delete dir="${build.dir}" />
</target>

<target name="compile" depends="init">
  <javac srcdir="${src.dir}" destdir="${build.dir}" classpathref="classpath" />
</target>
Listing E.3: build.xml
E.2 Custom Components (Chapters 3 and 4)

The following build.xml and build.properties files can be used for compiling and executing the manually instrumentated Bookstore application with the custom components, as described in Chapters 3 and 4. The files are included in the directory examples/userguide/ch3-4-custom-components/.

```xml
build.dir = build
src.dir = src
lib.dir = lib
meta.dir = META-INF
jar.file = BookstoreApplication.jar
properties.file = kieker.monitoring.properties
jar.file.commons-logging = commons-logging-1.1.1.jar
jar.file.kieker = kieker-1.3.jar
main.class = bookstoreApplication.Starter
meta.dir = META-INF
msg.filesNotFound = One or more of the required libraries could not be found. Please add the following files to the ${lib.dir} directory: ${jar.file.commons-logging}, ${jar.file.kieker}

Listing E.4: build.properties

```
<classpath>
  <path refid="classpath" /></path>
  <path location="${jar.file}" /></path>
</classpath>
<jvmarg value="-Dkieker.monitoring.configuration=${meta.dir}/${properties.file}" /></jvmarg>
</java>
</target>

<target name="-build-jar" depends="-compile">
  <jar destfile="${jar.file}" basedir="${build.dir}"
  <manifest>
    <attribute name="Main-Class" value="${main-class}" />
  </manifest>
  <delete dir="${build.dir}" />
</target>

<target name="-compile" depends="-init">
  <javac srcdir="${src.dir}" destdir="${build.dir}" classpathref="classpath" />
  <mkdir dir="${build.dir}/${meta.dir}" />
  <copy file="${meta.dir}/${properties.file}" tofile="${build.dir}/${meta.dir}/${properties.file}" />
</target>

<target name="-init">
  <delete dir="${build.dir}" />
  <mkdir dir="${build.dir}" />
</target>
</project>

Listing E.5: build.xml
E.3 AspectJ-based Trace Monitoring (Chapter 5)

The following build.xml and build.properties files can be used for compiling and executing the Bookstore application instrumentated with AspectJ, as described in Chapter 5.

The files are included in the directory examples/userguide/ch5-trace-monitoring-aspectj/.

```xml
<project name="Tutorial-Example" basedir="." default="run">
  <property file="build.properties" />
  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar" />
  </path>
  <condition property="not-all-files-available">
    <not>
      <and>
        <available file="${lib.dir}/${jar.file.commonslogging}" />
        <available file="${lib.dir}/${jar.file.aspectjweaver}" />
        <available file="${lib.dir}/${jar.file.kieker}" />
      </and>
    </not>
  </condition>
  <target name="-check-files" if="not-all-files-available">
```

Listing E.6: build.properties

```xml
build.dir=build
src.dir=src
lib.dir=lib
meta.dir=META-INF
jar.file=BookstoreApplication.jar
properties.file=kieker.monitoring.properties
jar.file.commonslogging=commons-logging-1.1.1.jar
jar.file.aspectjweaver=aspectjweaver-1.6.11.jar
jar.file.kieker=kieker-1.3.jar
main-class=bookstoreTracing.BookstoreStarter
main-class=hostname-rewriter=bookstoreTracing.BookstoreHostnameRewriter
meta.dir=META-INF
msg.filesNotFound=One or more of the required libraries could not be found. Please add the following files to the ${lib.dir} directory: ${jar.file.commonslogging}, ${jar.file.aspectjweaver}, ${jar.file.kieker}
```

Kieker 1.3 User Guide
<fail message="${msg.filesNotFound}" />
</target>

<target name='run' depends='--check-files, --build-jar'>
<java fork='true' classname='${main-class}'>
<classpath>
  <path refid='classpath' />
  <path location='${jar.file}' />
</classpath>
  <arg line='${num.requests}' />
</java>
</target>

<target name='run-hostname-rewriter' depends='--check-files, --build-jar'>
<java fork='true' classname='${main-class-hostname-rewriter}'>
<arg line='${analysis.directory}' />
<classpath>
  <path refid='classpath' />
  <path location='${jar.file}' />
</classpath>
</java>
</target>

<target name='--build-jar' depends='--compile'>
<copy file='META-INF/aop.xml' tofile='${build.dir}/META-INF/aop.xml' />
<jar destfile='${jar.file}' basedir='${build.dir}'>
  <manifest>
    <attribute name='Main-Class' value='${main-class}' />
  </manifest>
</jar>
<delete dir='${build.dir}' />
</target>

<target name='--compile' depends='--init'>
<javac srcdir='${src.dir}' destdir='${build.dir}' classpathref='classpath' />
</target>

<target name='--init'>
<delete dir='${build.dir}' />
<mkdir dir='${build.dir}' />
</target>
</project>

Listing E.7: build.xml

Kieker 1.3 User Guide 71
F Java EE Servlet Container Example

The Kieker download site includes an additional example JavaEEServletContainerExample. Using the sample Java Web application iBATIS JPetStore, this example demonstrates how to employ Kieker.Monitoring for monitoring a Java application running in a Java EE container—in this case, Apache Tomcat. Monitoring probes based on the Java EE Servlet API and AspectJ are used to monitor execution, trace, and session data (see Section 5).

The example is currently only prepared for UNIX-like systems. However, based on the descriptions below, it shouldn’t be too difficult to run it under Windows.

F.1 Preparation of the Tomcat Servlet Container

1. Copy the files kieker-1.3.jar, commons-logging-1.1.1.jar, and aspectjweaver-1.6.11.jar from Kieker’s binary distribution to the Tomcat’s lib/ directory.
2. Copy the file kieker-monitoring-servlet-1.3.war from Kieker’s binary distribution to the Tomcat’s webapps/ directory.
4. Tomcat’s bin/catalina.sh file was modified to add the location of the kieker.monitoring.properties and the AspectJ agent to the argument list of the JVM call:

```
JAVA_OPTS="-javaagent:lib/aspectjweaver-1.6.11.jar -Dorg.aspectj.weaver.showWeaveInfo=false -Daj.weaving.verbose=false"
JAVA_OPTS="$JAVA_OPTS -Dkieker.monitoring.configuration=$(dirname $0)/../lib/META-INF/kieker.monitoring.properties"
```

Listing F.1: Excerpt from catalina.sh

5. A prepared jpetstore.war file is already located in the Tomcat’s webapps/ directory. If you want to rebuild the sources, for example to modify the instrumentation, see Section F.3.

F.2 JPetStore and Kieker.Monitoring Control Servlet

We will now start the Tomcat server and generate some monitoring data by manually accessing the JPetStore web application.

1. Start the Tomcat server using the bin/startup.sh script in the Tomcat’s bin/ directory (you may have to execute "chmod +x bin/*.sh" in order to set the executable flags of the shell scripts). You should make sure, that the Tomcat started properly, by taking a look at the logs/catalina.out file. On error, the file logs/localhost.<date>.log may contain details to resolve the issue.

2. Now, you can access the JPetStore application by opening the URL http://localhost:8080/jpetstore/(Figure F.1(a)). Kieker initialization messages should appear in logs/catalina.out. The output includes the information where the monitoring data is written to (should be Tomcat’s temp/kieker-<DATE-TIME>/ directory).

3. Browse through the application to generate some monitoring data. This data can be analyzed and visualized using Kieker.TraceAnalysis, as described in Chapter 5.

4. Kieker includes a servlet to control the status of Kieker.Monitoring. It can be accessed via http://localhost:8080/kieker-monitoring-servlet-1.3/ (Figure F.1(b)).

(a) iBATIS JPetStore  (b) Kieker.Monitoring control servlet

Figure F.1
In order to rebuild the JPetStore sources (located in JPetStore-5.0-instrumented/), the following steps are required:

1. Copy the kieker-1.3.jar from Kieker’s binary distribution to the JPetStore’s devlib/ directory. It is required for the annotation-based instrumentation (@OperationExecutionMonitoringProbe), as described in Chapter 5.
2. Build the JPetStore with the build.xml by calling ant from within build/ directory.
3. You’ll find the packaged JPetStore .war-file in build/wars/.
4. Copy the file to the Tomcat’s webapps/ directory.
G Using the JMS Writer and Reader

This chapter gives a brief description on how to use the AsyncJMSWriter and JMSReader classes. The directory examples/userguide/appendix-JMS/ contains the sources, ant scripts etc. used in this example. It is based on the Bookstore application with manual instrumentation presented in Chapter 2.

1. Copy the files kieker-1.3.jar and commons-logging-1.1.1.jar from the binary distribution to the example’s lib/ directory.

2. The file examples/userguide/appendix-JMS/META-INF/kieker.monitoring.properties is already configured to use the AsyncJMSWriter:

```
###
kieker.monitoring.writer=kieker.monitoring.writer.jms.AsyncJMSWriter
#
## The url of the jndi provider that knows the jms service
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=tcp://127.0.0.1:3035/
#
## The topic at the jms server which is used in the publisher/subscribe communication.
kieker.monitoring.writer.jms.AsyncJMSWriter.Topic=queue1
#
## The type of the jms factory implementation, e.g.
InitialContextFactory
#
## The service name for the jms connection factory.
kieker.monitoring.writer.jms.AsyncJMSWriter.FactoryLookupName=ConnectionFactory
#
## The time that a jms message will be kept alive at the jms server before
## it is automatically deleted.
kieker.monitoring.writer.jms.AsyncJMSWriter.MessageTimeToLive=10000
#
## Asynchronous writers need to store monitoring records in an internal buffer.
## This parameter defines its capacity in terms of the number of records.
kieker.monitoring.writer.jms.AsyncJMSWriter.QueueSize=10000
#
## Behavior of the asynchronous writer when the internal queue is full:
## 0: terminate Monitoring with an error (default)
## 1: writer blocks until queue capacity is available
## 2: writer discards new records until space is available
## Be careful when using the value '1' since then, the asynchronous writer
## is no longer decoupled from the monitored application.
kieker.monitoring.writer.jms.AsyncJMSWriter.QueueFullBehavior=0
```

Listing G.1: Excerpt from kieker.monitoring.properties configuring the JMS writer
3. Download an OpenJMS install archive from http://openjms.sourceforge.net and decompress it to the root directory of the example.

4. Copy the following files from the OpenJMS lib/ folder to the lib/ directory of this example:
   a) openjms-<version>.jar
   b) openjms-common-<version>.jar
   c) openjms-net-<version>.jar
   d) jms-<version>.jar
   e) concurrent-<version>.jar
   f) spice-jndikit-<version>.jar

The execution of the example is performed by the following three steps:

1. Start the JMS server (you may have to set your JAVA_HOME variable first):
   ⊲ openjms-<version>/bin/startup.sh

2. Start the analysis part (in a new terminal):
   ⊲ ant run−analysis

3. Start the instrumented Bookstore (in a new terminal):
   ⊲ ant run−monitoring
H Sigar-Based Samplers for System-Level Monitoring

This chapter gives a brief description on how to use the included periodic samplers (Section 3.1.4) for monitoring CPU utilization and memory/swap usage. The directory examples/userguide/appendix-Sigar/ contains the sources, ant scripts etc. used in this example. These samplers employ the Sigar API [1].

H.1 Preparation

1. Copy the files kieker-1.3.jar, commons-logging-1.1.1.jar, and sigar-1.6.3.jar from the binary distribution to the example’s lib/ directory.
2. Additionally, depending on the underlying system platform, corresponding Sigar native libraries need to be placed in the example’s lib/ directory. Kieker’s lib/ folder already includes the right libraries for x86 Linux/Windows platforms (libsigar-x86-linux.so and sigar-x86-winnt.[dll|lib]). Native libraries for other platforms can be downloaded from [1].

H.2 Using the Sigar-Based Samplers

The Sigar API [1] provides access to a number of system-level inventory and monitoring data, e.g., regarding memory, swap, cpu, file system, and network devices. Kieker includes Sigar-based samplers for monitoring CPU utilization (CPUsDetailedPercSampler, CPUsCombinedPercSampler) and memory/swap usage (MemSwapUsageSampler). When registered as a periodic sampler (Section 3.1.4), these samplers collect the data of interest employing the Sigar API, and write monitoring records of types CPUUtilizationRecord, ResourceUtilizationRecord, and MemSwapUsageRecord respectively to the configured monitoring log.

Listing H.1 shows an excerpt from this example’s MonitoringStarter which creates and registers two Sigar-based periodic samplers. For reasons of performance and thread-safety, the SigarSamplerFactory should be used to create instances of the Sigar-based Samplers.
Based on the existing samplers, users can easily create custom Sigar-based samplers by extending the class `AbstractSigarSampler`. For example, Listing 3.1 in Section 3.1.4 shows the `MemSwapUsageSampler`'s `sample` method. Typically, it is also required to define a corresponding monitoring record type, as explained in Section 3.3. When implementing custom Sigar-based samplers, the `SigarSamplerFactory`'s `getSigar` method should be used to retrieve a `Sigar` instance.

This example uses a stand-alone Java application to set up a Sigar-based monitoring process. When using servlet containers, users may consider implementing this routine as a `ServletContextListener`, which are executed when the container is started and shutdown. As an example, Kieker includes a `CPUMemUsageServletContextListener`.

### H.3 Executing the Example

The execution of the example is performed by the following two steps:

1. Monitoring CPU utilization and memory usage for 30 seconds (class `MonitoringStarter`):
   - `ant run--monitoring`

Kieker’s console output lists the location of the directory containing the file system monitoring log. The following listing shows an excerpt:

```
writer: 'kieker.monitoring.writer.filesystem.AsyncFsWriter'
configuration:
  kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior = '0'
  kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize = '10000'
  kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath = ''
  kieker.monitoring.writer.filesystem.AsyncFsWriter.storeInJavaIoTmpdir = 'true'

writer threads (1):
  finished: 'false'; writing to directory: '/tmp/kieker-20110511-10095928-UTC-avanhoorn-thinkpad-KIEKER-SINGLETON'
```
A sample monitoring log can be found in the directory examples/userguide/appendix-Sigar/testdata/kieker-20110511-10095928-UTC-avanhoorn-thinkpad-KIEKER-SINGLETON/.

2. Analyzing the monitoring data (class AnalysisStarter):

```
ant run-analysis -Danalysis.directory=</path/to/monitoring/log/>
```

You need to replace `</path/to/monitoring/log/>` by the location of the file system monitoring log. You can also use the above-mentioned monitoring log included in the example.

The `AnalysisStarter` produces a simple console output for each monitoring record, as shown in the following excerpt:

```
Wed, 11 May 2011 10:10:01 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ; utilization: 0.00 %
Wed, 11 May 2011 10:10:01 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ; utilization: 0.00 %
Wed, 11 May 2011 10:10:01 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 722.0 MB ; swap usage: 0.0 MB
Wed, 11 May 2011 10:10:06 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ; utilization: 5.35 %
Wed, 11 May 2011 10:10:06 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ; utilization: 1.31 %
Wed, 11 May 2011 10:10:06 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap usage: 0.0 MB
Wed, 11 May 2011 10:10:11 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ; utilization: 1.80 %
Wed, 11 May 2011 10:10:11 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ; utilization: 0.20 %
Wed, 11 May 2011 10:10:11 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap usage: 0.0 MB
Wed, 11 May 2011 10:10:16 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ; utilization: 0.79 %
Wed, 11 May 2011 10:10:16 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap usage: 0.0 MB
Wed, 11 May 2011 10:10:21 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ; utilization: 1.80 %
Wed, 11 May 2011 10:10:21 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ; utilization: 0.79 %
Wed, 11 May 2011 10:10:21 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap usage: 0.0 MB
Wed, 11 May 2011 10:10:26 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ; utilization: 0.40 %
Wed, 11 May 2011 10:10:26 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ; utilization: 0.59 %
Wed, 11 May 2011 10:10:26 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap usage: 0.0 MB
```
# Libraries

The following table shows all libraries which are used by Kieker and explains them briefly. These libraries are included in the `lib/` directory of both the Kieker binary and source distributions.

The Apache Commons [7] library (`commons-logging-1.1.1.jar`) is the only third-party library always needed when using Kieker. The need to provide the additional libraries in the classpath depends on the specific configuration. For example, the AspectJ libraries are only required when using AspectJ-based monitoring probes.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aspectjrt-1.6.11.jar</td>
<td>This jar-file contains the runtime library for AspectJ programs.</td>
</tr>
<tr>
<td>aspectjtools-1.6.11.jar</td>
<td>This package contains the tools (the AspectJ Compiler and Browser) for AspectJ.</td>
</tr>
<tr>
<td>aspectjweaver-1.6.11.jar</td>
<td>This jar contains the weaver-agent for the aspect-oriented-extension for Java named AspectJ.</td>
</tr>
<tr>
<td>commons-cli-1.2.jar</td>
<td>Apache Commons CLI provides a simple API for working with the command line arguments and options.</td>
</tr>
<tr>
<td>commons-io-1.2.jar</td>
<td>Apache Commons-IO contains utility classes, stream implementations, file filters, and endian classes.</td>
</tr>
<tr>
<td>commons-logging-1.1.1.jar</td>
<td>Apache Commons Logging is a thin adapter allowing configurable bridging to other, well known logging systems.</td>
</tr>
<tr>
<td>commons-pool-1.2.jar</td>
<td>Apache Commons Pool is an Object-pooling API supplying different interfaces and classes to create modular object pools.</td>
</tr>
<tr>
<td>concurrent-1.3.4.jar</td>
<td>This library supplies different thread-safe classes for the enhanced development of multi-threaded Java applications.</td>
</tr>
<tr>
<td>cxf-api-2.2.10.jar</td>
<td>Apache CXF is an open source services framework.</td>
</tr>
<tr>
<td>Package Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>cxf-common-utilities-2.2.10.jar</td>
<td>This package contains different classes for Apache CXF.</td>
</tr>
<tr>
<td>cxf-rt-bindings-soap-2.2.10.jar</td>
<td>This package contains necessary files to use Apache CXF as well with the Simple Object Access Protocol (SOAP).</td>
</tr>
<tr>
<td>cxf-rt-core-2.2.10.jar</td>
<td>This library contains the Apache CXF Runtime Core.</td>
</tr>
<tr>
<td>derby.jar</td>
<td>Apache Derby is a lightweight database written in Java which can also be used as an embedded database. This library contains the necessary drivers for the database as well as the database management system itself.</td>
</tr>
<tr>
<td>jmc.jar</td>
<td>This library contains the Java Media Components which can be used for example for playing video content in Swing applications.</td>
</tr>
<tr>
<td>jms-1.1.jar</td>
<td>Java Message Service is an API to send and receive messages within a client and to control so called Message Oriented Middleware (MOM).</td>
</tr>
<tr>
<td>jndi-1.2.1.jar</td>
<td>The Java Naming and Directory Interface is an API which provides methods for multiple naming and directory services. It can be used for example to register disposed files in a network and to allow other part of a Java program to use them for RMI calls.</td>
</tr>
<tr>
<td>junit-4.5.jar</td>
<td>This jar-file contains the necessary classes for the JUnit-tests, which can be used to test automatically Java classes.</td>
</tr>
<tr>
<td>log4j-1.2.15.jar</td>
<td>Apache log4j is a framework for the logging of messages, errors and exceptions in Java applications.</td>
</tr>
<tr>
<td>mysql-connector-java-5.1.5-bin.jar</td>
<td>This library contains the drivers to connect from a Java application to a MySQL database system.</td>
</tr>
<tr>
<td>openjms-0.7.7-beta-1.jar</td>
<td>OpenJMS is an open source implementation of Sun Microsystems’s Java Message Service API 1.1 Specification</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>openjms-common-0.7.7-beta-1.jar</td>
<td>OpenJMS is an open source implementation of Sun Microsystems’s Java Message Service API 1.1 Specification</td>
</tr>
<tr>
<td>openjms-net-0.7.7-beta-1.jar</td>
<td>OpenJMS is an open source implementation of Sun Microsystems’s Java Message Service API 1.1 Specification</td>
</tr>
<tr>
<td>rabbitmq-client.jar</td>
<td>This library contains the client for the RabbitMQ messaging system.</td>
</tr>
<tr>
<td>Scenario.jar</td>
<td>This package provides scene graph functionality for Java.</td>
</tr>
<tr>
<td>servlet.jar</td>
<td>This package contains different classes for the work with servlets.</td>
</tr>
<tr>
<td>servlet-api.jar</td>
<td>The Java Servlet API supplies protocols to let applications respond for example to HTTP requests.</td>
</tr>
<tr>
<td>sigar-1.6.3.jar</td>
<td>Hyperic SIGAR (System Information Gath er) provides a Java API to system inventory and monitoring data (Memory, CPU etc.). In addition to the Jar file, it is required to add corresponding platform-specific native libraries to the classpath, which can be downloaded from [1]. Kieker’s lib/ folder already includes such libraries for Linux/Windows for the x86 architecture ([libsigar-x86-linux.so and sigar-x86-winnt.[dll</td>
</tr>
<tr>
<td>spice-jndikit-1.2.jar</td>
<td>The JNDI Kit is a toolkit for the easy use of the so called Java Naming and Directory Interface.</td>
</tr>
<tr>
<td>spring.jar</td>
<td>The spring framework delivers different methods and classes to make the handling with Java/Java EE easier.</td>
</tr>
<tr>
<td>spring-web.jar</td>
<td>This library contains the web application context, multipart resolver, Struts support, JSF support and web utilities for the spring framework.</td>
</tr>
</tbody>
</table>
Bibliography


