INSTITUT FÜR INFORMATIK

Continuous Monitoring of Software Services: Design and Application of the Kieker Framework

André van Hoorn, Matthias Rohr, Wilhelm Hasselbring,
Jan Waller, Jens Ehlers, Sören Frey, Dennis Kieselhorst

Bericht Nr. 0921
November 2009

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Continuous Monitoring of Software Services: Design and Application of the Kieker Framework

André van Hoorn[1,3], Matthias Rohr[1,2], Wilhelm Hasselbring[1,3], Jan Waller[3], Jens Ehlers[3], Sören Frey[3], and Dennis Kieselhorst[4]

[1] Graduate School TrustSoft, University of Oldenburg, D-26111 Oldenburg. Germany
[2] BTC AG – Business Technology Consulting AG, D-26121 Oldenburg, Germany
[3] Software Engineering Group, University of Kiel, D-24098 Kiel, Germany
[4] EWE TEL GmbH, D-26133 Oldenburg, Germany

***Abstract*— In addition to studying the construction and evolution of software services, the software engineering discipline needs to address the operation of continuously running software services. A requirement for its robust operation are means for effective monitoring of software runtime behavior. In contrast to profiling for construction activities, monitoring of operational services should only impose a small performance overhead. Furthermore, instrumentation should be non-intrusive to the business logic, as far as possible.**

We present the Kieker framework for monitoring software runtime behavior, e.g., internal performance or (distributed) trace data. The flexible architecture allows to replace or add framework components, including monitoring probes, analysis components, and monitoring record types shared by logging and analysis. As a non-intrusive instrumentation technique, Kieker currently employs, but is not restricted to, aspect-oriented programming. An extensive lab study evaluates and quantifies the low overhead caused by the framework components. Qualitative evaluations provided by industrial case studies demonstrate the practicality of the approach with a telecommunication customer self service and a digital photo submission service. Kieker is available as open-source software, where both the academic and industrial partners contribute to the code. Our experiment data is publicly available, allowing interested researchers to repeat and extend our lab experiments.

Keywords: D.2.2.c Distributed/Internet based software engineering tools and techniques; D.2.7.m Restructuring, reverse engineering, and reengineering; D.2.8.b Performance measures; D.4.8.a Measurements; D.4.8.c Monitors

I. Introduction

Traditionally, software engineering primarily addresses the *construction* and *evolution* of software. In addition, means for efficient and robust *operation* are critical for software services. Continuous monitoring may enable early detection of quality-of-service problems, such as performance degradation, and may deliver usage data for resource management. Such monitoring information is also required to check the fulfillment of service level agreements (SLAs). Therefore, a system's runtime behavior should be monitored and analyzed continuously.

Monitoring can be performed on all layers of a software system, e.g., regarding hardware, operating system, middleware, and application state. In this paper, we focus on monitoring of software runtime behavior, particularly addressing middleware and application-level. This can be the basis for runtime failure detection and diagnosis for software services [1], reverse engineering of design models from program execution [2, 3], and approaches for self-adaptive software [4, 5]. Moreover, it provides valuable data for dynamic analysis [6] and model calibration [7].

In this paper, we present the object-oriented Kieker monitoring framework, which has been designed for continuous monitoring of software services. The framework components for software instrumentation, logging, and analysis/visualization are extensible and may easily be replaced to fulfill the requirements of specific project contexts. For instance, as a non-intrusive instrumentation technique, we employ aspect-oriented programming (AOP) [8]. Kieker currently provides sample instrumentations utilizing the popular AOP interception frameworks AspectJ [9], Spring AOP [10] and Apache CXF [11], which may be extended or replaced by other framework users.

Kieker uses a common data structure for monitoring records in all components that produce or consume monitoring data. For analysis of monitoring data, Kieker provides several visualizations of a system's runtime behavior, such as UML sequence diagrams, dependency graphs, and Markov chains. These models are extracted from recorded application-internal traces originating from system-provided services. The analysis may be performed online or offline. Kieker supports distributed request tracing, since the service-providing components of large-scale software systems are usually distributed across several execution containers on physical or virtual server nodes.

We employ Kieker for various research purposes, e.g., fault localization based on timing behavior anomaly detection [12], architecture-based runtime adaptation/reconfiguration [13, 14], visualization of software runtime behavior [15], application-level intrusion detection [16], and trace-based performance analysis [17, 18].

Continuous monitoring has to be distinguished from profiling activities, which are employed when a software service is under development in a test environment. Profiling offers advanced debugging functions, whereby considerable runtime overhead is usually accepted for obtaining detailed data for program analysis. Such an overhead is not acceptable for the continuous operation of software services.

In addition to the presentation of Kieker's architecture, this paper includes a quantitative and qualitative evaluation of the framework and the monitoring methodology. Lab experiments, employing micro benchmarks, quantify the small overhead that is introduced per activated monitoring probe and break down the cause of overhead to the framework components. In order to investigate the applicability to real environments, we conducted industrial case studies with operational software services of a telecommunication customer self service and a digital photo submission service. Kieker is available as open-source software,[1] where both the academic and industrial partners contribute to the code. Along with the code, our experiment data is available such that interested researchers can repeat and extend our lab experiments.

[1] http://kieker.sourceforge.net/

To summarize, the original contributions of this paper are

1) the presentation of Kieker's flexible framework architecture,[2]
2) a quantitative evaluation of Kieker's runtime overhead in extensive lab experiments,
3) a qualitative evaluation of Kieker's runtime overhead and the practicality of fine-grained (distributed) monitoring in industrial case studies, and
4) a summary of design recommendations for the *construction* and *evolution* of software services, which emerged from our lab experiments and industrial case studies, to achieve an efficient and robust *operation* of these software services.

The remainder of this paper is structured as follows: Section II describes the architecture of the Kieker framework. Section III presents the methodology to monitor and reconstruct (distributed) traces, and shows analysis and visualization examples. The systematic assessment of the overhead using benchmarks in lab experiments is presented in Section IV. In Section V, we report on the application of the framework in industrial systems. Our experience and the emerged design knowledge from using Kieker are discussed in Section VI. Related work follows in Section VII. Section VIII draws our conclusions and indicates areas for future work.

II. Framework Architecture

This section provides an overview of Kieker's framework architecture. The outstanding features of Kieker may be summarized as follows:

- A common, extensible monitoring record model that is shared among the logging and analysis component: in this paper, we use record types for storing response times (in this section), and (distributed) trace information (Sections III and V).
- An extensible reader/writer model: the monitoring records may be written to and being read from relational databases, to the file system or to JMS queues; additional data sinks may be added.

[2] An initial version of Kieker has been presented in [15]; meanwhile major revisions for enhanced flexibility have been added.

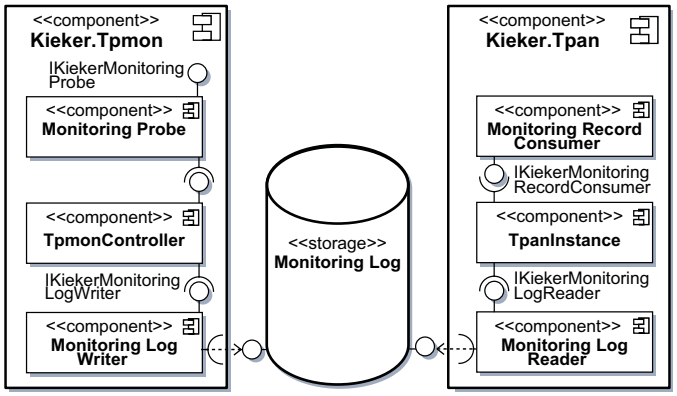


Fig. 1
TOP-LEVEL VIEW ON KIEKER'S ARCHITECTURE

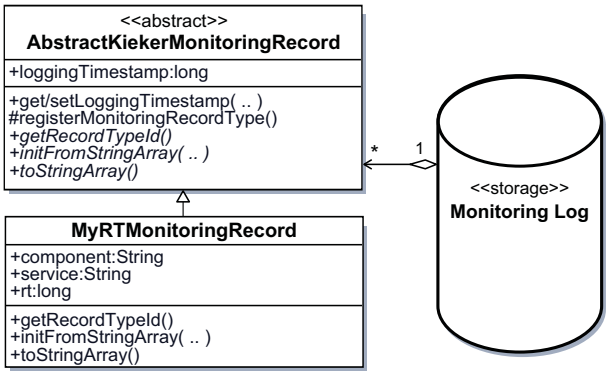


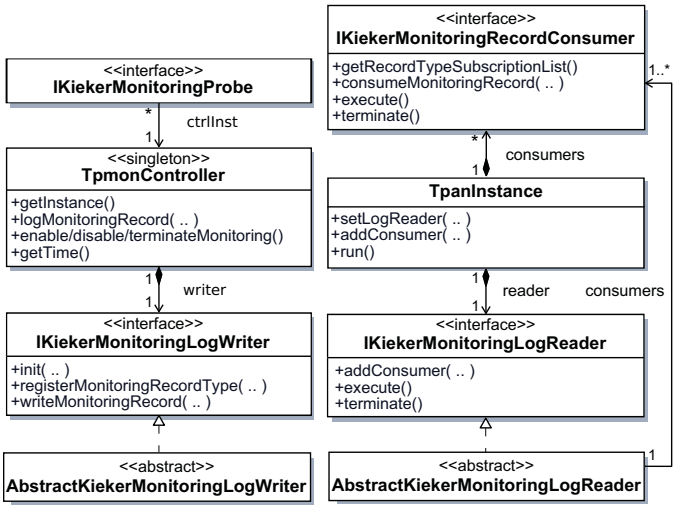
Fig. 2
ABSTRACT KIEKER MONITORING RECORD STORED IN THE MONITORING LOG, WITH A CONCRETE MONITORING RECORD AS EXAMPLE.

- An extensible monitoring record consumer model: the monitoring records may be analyzed and visualized for various purposes. If the monitoring log is based on a messaging middleware, continuous online analysis is supported.

The Unified Modeling Language (UML) [19] is employed for describing the architecture, independent of the programming language (which is Java for Kieker).

At the top-level, Kieker is partitioned into the two components Kieker.Tpmon and Kieker.Tpan with the Monitoring Log in between, as illustrated in Figure 1. Kieker.Tpmon provides a reusable infrastructure for collecting application-level monitoring data in Monitoring Probes and writing this monitoring data to the Monitoring Log, e.g., the local file system, a database, or a messaging queue, using a Monitoring Log Writer. The TpmonController is responsible for initializing and controlling a Kieker.Tpmon instance. The Monitoring Log contains Monitoring Records, as defined in Figure 2. Each record holds the monitoring data of a single measurement created by the Monitoring Probes. Kieker.Tpan provides the infrastructure for analyzing the Monitoring Log: a Monitoring Log Reader (Figure 1) reads Monitoring Records from the Monitoring Log and delivers these to registered Monitoring Record Consumers, according to the observer design pattern [20]. Monitoring Record Consumers perform the actual analysis or visualization functionality. A Kieker.Tpan instance is initialized and controlled by a TpanInstance instance (Figure 1).

Figure 3 shows the core Kieker framework classes and interfaces with their associations in a UML Class Diagram notation. Kieker offers different implementations for the Monitoring Record, Monitoring Probe, Monitoring Log Writer, Monitoring Log Reader, and Monitoring Record Consumer components and allows to use customized components created by implementing or extending the interfaces or abstract classes of the framework corresponding to these components, as described below.



(a) Kieker.Tpmon (b) Kieker.Tpan

Fig. 3
CLASSES AND INTERFACES OF THE FRAMEWORK COMPONENTS (A) KIEKER.TPMON AND (B) KIEKER.TPMON.

Figure 4 illustrates a Monitoring Record's life cycle. It shows the sequence of interactions among instances of the Kieker components in a UML Communication Diagram notation, whereby the numbers at the operation calls, which are attached to the

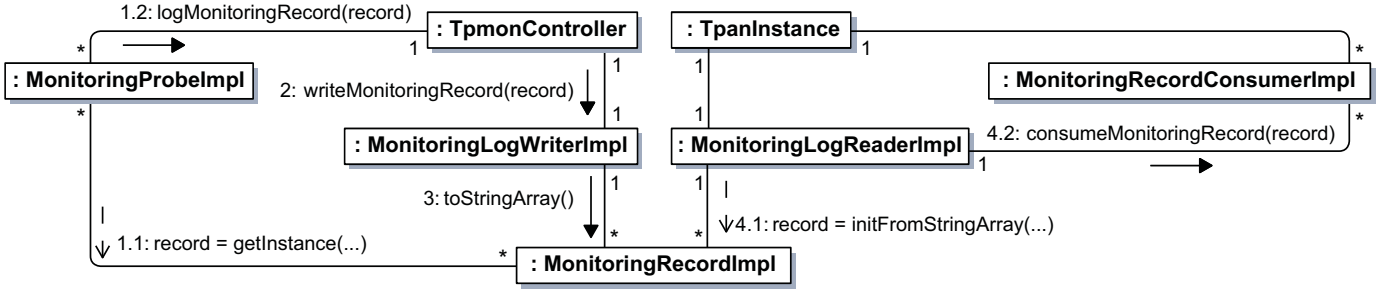


Fig. 4

COMMUNICATION AMONG FRAMEWORK COMPONENTS FOR CREATING AND WRITING A MONITORING RECORD IN KIEKER.TPMON (1.1–3) AS WELL AS READING AND USING THIS MONITORING RECORD IN KIEKER.TPAN (4.1, 4.2).

links, define the order. The multiplicities at the links indicate the possible number of these links among respective object instances.

A. Monitoring Record

A Monitoring Record holds the measurement data collected in a single measurement. Different types of Monitoring Records can be used together in one Kieker.Tpmon instance. Figure 2 shows a concrete Monitoring Record type *MyRTMonitoringRecord* which can be used to store response times of services provided by software components. As required by all Monitoring Record types, it extends the abstract framework class *AbstractKiekerMonitoringRecord*.

B. Monitoring Probe

A Monitoring Probe (Figure 1) contains the measurement logic which collects and possibly preprocesses measurement data from the application. A Monitoring Probe creates an instance of a Monitoring Record and sends this Monitoring Record to the TpmonController by calling the method *logMonitoringRecord(..)* of the TpmonController, as shown in Figure 4. All Monitoring Probe types must implement the interface *IKiekerMonitoringProbe* (Figure 3).

Monitoring Probes of different types can be used together in a single Kieker.Tpmon instance and are typically tightly bound to middleware underlying the application. For example, the interception APIs of Java middleware technologies provided by the Spring framework, the Java Servlet specification, and the Apache CXF Web service framework can be used to implement and integrate Monitoring Probes into an application. In Section V, we demonstrate how different technology-specific types of Monitoring Probes can be used in combination to monitor distributed traces.

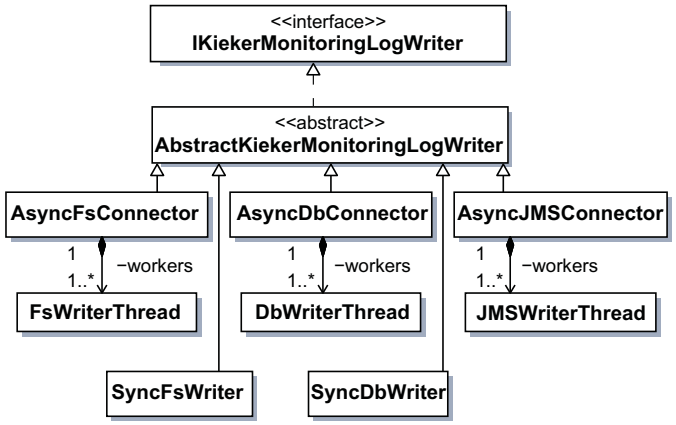
Listing 1

EXAMPLE ASPECTJ RESPONSE TIME MONITORING PROBE

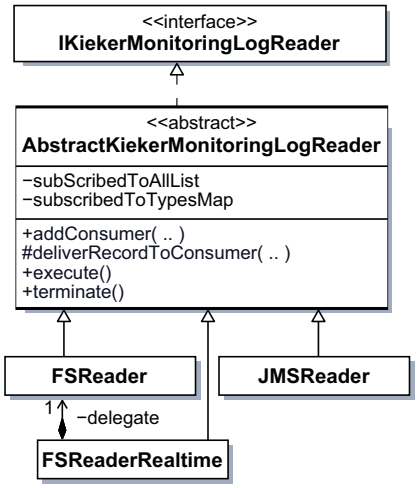
```
1  @Aspect
2  public class MyRTMonitoringProbe
3         implements IKiekerMonitoringProbe {
4
5   static final TpmonController CTRL =
6          TpmonController.getInstance();
7
8   @Around
9      (value="execution(@MyRTProbe * *.*(..))")
10  public Object probe(ProceedingJoinPoint j)
11         throws Throwable {
12    MyRTMonitoringRecord record =
13                  new MyRTMonitoringRecord();
14    record.component = j.getSignature()
15                        .getDeclaringTypeName();
16    record.service = j.getSignature().getName();
17    Object retval;
18    long tin = CTRL.getTime();
19    try { retval = j.proceed(); }
20    catch (Exception e) { throw e; }
21    finally {
22     record.rt = CTRL.getTime() - tin;
23     CTRL.logMonitoringRecord(record);
24    }
25    return retval;
26  }
27 }
```

As an example, Listing 1 shows an AspectJ-based Monitoring Probe which measures response times of Java methods and stores this data as Monitoring Records of the previously introduced type *MyRTMonitoringRecord* (Figure 2). In this example, all Java methods annotated with *MyRTProbe* are instrumented, as follows for the method *searchBook()* via Java annotation:

```
@MyRTProbe()
public static void searchBook() { .. }
```



(a) Implemented writers



(b) Implemented readers

Fig. 5

IMPLEMENTED MONITORING LOG WRITERS AND READERS. THE ABSTRACT CLASSES HAVE BEEN INTRODUCED IN FIGURE 3.

Calls to an instrumented method are intercepted and the execution proceeds with the method *probe(..)* (lines 10–26) containing the response time measurement logic of the Monitoring Probe. Before the execution is delegated to the intercepted method (line 19), a Monitoring Record instance is created and initialized (lines 13–16) and the timestamp before the execution is taken (line 18). After this execution returns, the response time is calculated and stored in the Monitoring Record (line 22) which is then passed to the TpmonController instance (line 23).

C. Monitoring Log Writer

A Monitoring Log Writer is responsible for writing/serializing the Monitoring Records to the Monitoring Log. For each Monitoring Record to be logged, the writer is invoked by the TpmonController and writes the data contained in the retrieved Monitoring Record by calling the Monitoring Record's *toStringArray()* method, as illustrated in Figure 4. Figure 5(a) shows the Monitoring Log Writers which are already included in the Kieker framework, supporting the file system, relational databases, and JMS queues. The prefixes *Sync/Async* in Figure 5(a) indicate whether the I/O operation required to log the Monitoring Record is performed within the Monitoring Probe's thread of control (synchronously) or by one or more asynchronous writer thread(s) using an internal buffer. The filesystem writer stores Monitoring Records represented as comma-separated values (CSV). Listing 2 shows a sample filesystem Monitoring Log containing entries of the *MyRTMonitoringRecord* Monitoring Record type. Each line contains the Monitoring Record type identifier, as well as the values of the record fields *loggingTimestamp* (cropped in the listing), *component*, *service*, and *rt* (response time in nanoseconds). A mapping file is used to store the mapping between a Monitoring Record type identifier and its implementing class, as shown in Listing 3.

Listing 2

FILESYSTEM MONITORING LOG WITH MONITORING RECORDS OF TYPE MYRTMONITORINGRECORD

```
$1;..267737726;Catalog;getBook;2104283
$1;..268321753;Catalog;getBook;2679347
$1;..324713919;Catalog;getBook;20082302
$1;..324780416;CRM;getOffers;20164491
$1;..324787610;Bookstore;searchBook;93795571
$1;..325166071;Catalog;getBook;20568496
$1;..325180972;CRM;getOffers;20612072
$1;..325186824;Bookstore;searchBook;94195055
```

Listing 3

MAPPING OF MONITORING RECORD TYPE IDENTIFIER TO IMPLEMENTING CLASS

```
$1=MyRTMonitoringRecord
```

D. Monitoring Log Reader

A Monitoring Log Reader is used to create Monitoring Record instances from a Monitoring Log written by a corresponding Monitoring Log Writer (by calling a Monitoring Record's *initFromStringArray(..)* method), see Figure 2. As shown in Figure 5(b), Kieker includes Monitoring Log Readers corresponding to the included writers for the filesystem and for JMS queues. Monitoring Log Readers extend the

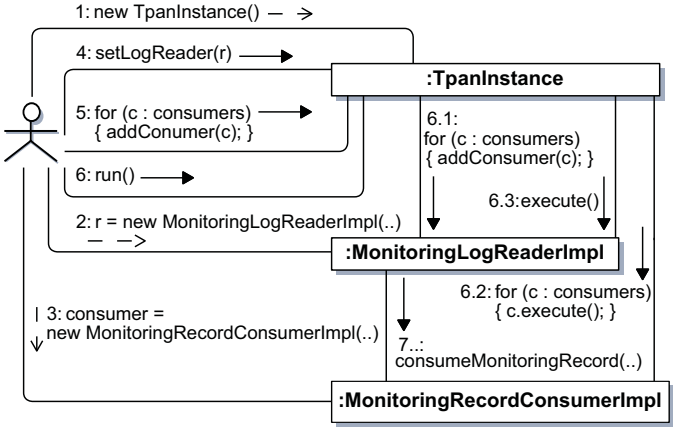


Fig. 6
KIEKER.TPAN COMMUNICATION DIAGRAM

abstract class *AbstractKiekerMonitoringLogReader*, which already provides convenient functionality for handling the subscription of Monitoring Record Consumers to Monitoring Records of specific types as well as the delivery of records to the subscribers.

Additionally, Kieker includes the Monitoring Log Reader *FSReaderRealtime* which can be used to replay Monitoring Records from a filesystem-based Monitoring Log in the original timescale. This has proven to be very helpful for debugging purposes and for simulating continuously incoming monitoring data while developing online analysis components.

E. Monitoring Record Consumer

Analysis or visualization components are integrated into Kieker.Tpan by implementing the *IKiekerRecordConsumer* interface (see Figure 3). A Monitoring Record Consumer is registered to the TpanInstance as a subscriber for Monitoring Records of selected or all Monitoring Record types (as returned by the *getRecordTypeSubscriptionList()* method, see Figure 3). The TpanInstance delegates the subscription list to the Monitoring Log Reader, which directly delivers newly incoming records of interest to the consumers by calling their *consumeMonitoringRecord(..)* method, as illustrated in Figure 6.

Listing 4 shows an example response time monitor for *MyRTMonitoringRecord* that we use as an example in the present section. The monitor compares (line 19) incoming response time values with a threshold value stored in the object's variable *rtSlo* (response time service level objective),

Listing 4
EXAMPLE RESPONSE TIME MONITOR

```
1  public class RTMonitor
2         implements IKiekerRecordConsumer {
3
4   private final long rtSlo;
5   public RTMonitor(long rtSlo) {
6    this.rtSlo = rtSlo;
7   }
8
9   public String[] getRecordTypeSubscriptionList(){
10   return new String[] {
11    MyRTMonitoringRecord.class.getName()
12   };
13  }
14
15  public void consumeMonitoringRecord
16         (AbstractKiekerMonitoringRecord r) {
17   MyRTMonitoringRecord rtRec =
18                       (MyRTMonitoringRecord) r;
19   if (rtRec.rt > this.rtSlo) {
20     /* SLO violation! */
21   }
22  }
23
24  public boolean execute() { return true; }
25  public void terminate() { }
26 }
```

which is specified on object creation (line 6). By returning the classname of the Monitoring Record type *MyRTMonitoringRecord* in line 10, the monitor only receives Monitoring Records of this type. Please remind that Kieker allows to use the same data structures (Monitoring Records) in analysis components as used in the Monitoring Probes, in this case the Monitoring Probe measuring the response times (Listing 1) and the Monitoring Record Consumer (Listing 4) implementing the response time monitor.

When a TpanInstance is started by calling its *run()* method, the *execute()* methods of all registered Monitoring Record Consumers are called. This allows the implementation of an asynchronous event-based architecture, which is particularly useful for online analysis components. Such an online analysis component spawns a thread in its *execute()* method implementing the analysis tasks based on asyncronously incoming Monitoring Records.

III. DYNAMIC TRACE ANALYSIS

For dynamic trace analysis, Kieker records information about operation executions and about control flow traces. We first introduce Kieker's approach to logging and reconstructing trace information before example analyses and visualizations are presented in Section III-B.

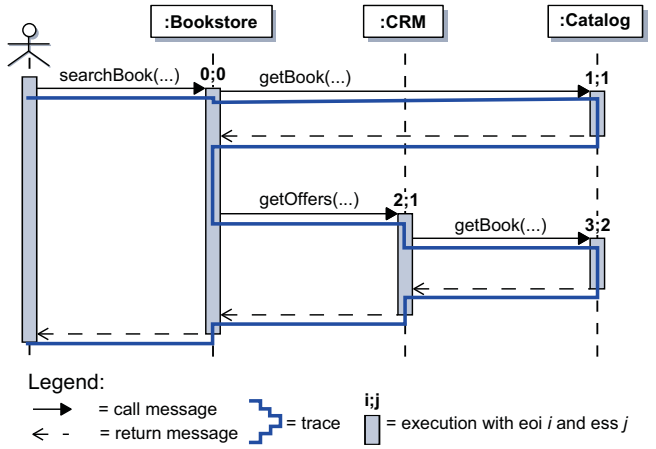


Fig. 7

UML SEQUENCE DIAGRAM ILLUSTRATING OUR TRACE-RELATED TERMINOLOGY

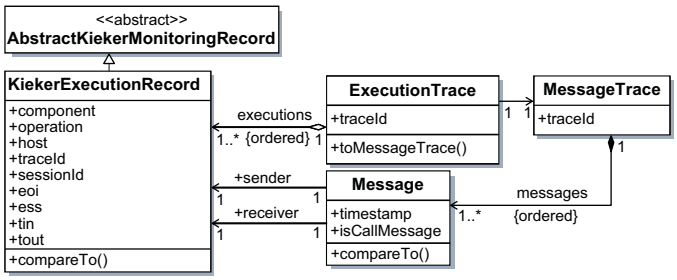


Fig. 8

UML CLASS DIAGRAM DEFINING THE TRACE-RELATED INFORMATION IN KIEKER.TPAN

A. Logging and Reconstructing Trace Information

According to the UML [19], an *operation* is a behavioral feature of a classifier. Examples are methods associated to classes in object-oriented applications and services provided by components. In our terminology, operations are features of components that implement provided services. An *execution* of such an operation denotes the execution of the associated behavior by the corresponding component instance at runtime. The UML Sequence Diagram in Figure 7 includes four executions of three different operations (*getBook()* is called twice).

Kieker includes the Monitoring Record type *KiekerExecutionRecord* which can be used to write execution information into the Monitoring Log. As shown in Figure 8, a *KiekerExecutionRecord* contains information about the executed operation, the corresponding component, the hostname on which the execution was performed, as well as timestamps, typically with nanosecond resolution, for the start (*tin*) and end (*tout*) of an execution. Kieker includes different Monitoring Probe types for logging *KiekerExecutionRecords* within an application, which look similar to the Monitoring Probe in Listing 1.

A request to a system-provided service results in a nested control flow of corresponding executions, referred to as a *trace*. Kieker provides efficient facilities for attaching a unique trace identifier to the thread executing a service request, which is then contained in any *KiekerExecutionRecord* of that trace (see Figure 8).

If the reconstruction of traces from a Monitoring Log containing *KiekerExecutionRecords* would only include the trace information presented so far, we would require the following assumptions: (a) no two execution start or end time events (*tin*/*tout*) within the same trace occur at the same time; and (b) clocks in a distributed system are perfectly synchronized (both with respect to the respective time resolution). Since both assumptions cannot be guaranteed in realistic environments, Kieker includes efficient facilities to attach two additional parameters to any *KiekerExecutionRecord* in order to log the information needed to reconstruct (distributed) traces from the Monitoring Log reliably: an *execution order index eoi* and an *execution stack size ess* (see Figure 8).

- eoi: An execution with an execution order index value i denotes the i-th execution started within a trace (starting with the value 0).
- ess: An execution with an execution stack size value j denotes an execution that was started when the depth of the calling stack for the corresponding trace was j.

The executions shown in the example trace in Figure 7 are annotated with the corresponding execution order index (eoi) and execution stack size (ess) values. Note, that while an execution order index is unique within a trace, an execution stack size value can, and usually does, occur more than once.

In Kieker.Tpan, two equivalent representations of traces are used internally: execution traces and message traces. An execution trace representation of a trace is simply the ordered (by execution order index values) sequence of executions (stored as *KiekerExecutionRecords*, see Figure 8). A message trace describes a trace in terms of an ordered sequence of messages instead of executions. Each execution can be described by a corresponding call message,

Listing 5

KIEKER.TPAN OUTPUT OF EXECUTION TRACE REPRESENTATION

```
TraceId 8430034814995791873 (NOSESSION):
<[0,0] 1257440666759412388-1257440666841265860 srv0::Bookstore.searchBook(..)>
<[1,1] 1257440666805601818-1257440666807695902 srv0::Catalog.getBook(..)>
<[2,1] 1257440666820790063-1257440666841169272 srv0::CRM.getOffers(..)>
<[3,2] 1257440666820839575-1257440666840922990 srv0::Catalog.getBook(..)>
```

Listing 6

KIEKER.TPAN OUTPUT OF MESSAGE TRACE REPRESENTATION

```
TraceId 8430034814995791873 (NOSESSION):
<SND 1257440666759412388 $-->srv0::Bookstore.searchBook(..)[0,0]>
<SND 1257440666805601818 srv0::Bookstore.searchBook(..)[0,0]-->srv0::Catalog.getBook(..)[1,1]>
<RVC 1257440666807695902 srv0::Catalog.getBook(..)[1,1]-->srv0::Bookstore.searchBook(..)[0,0]>
<SND 1257440666820790063 srv0::Bookstore.searchBook(..)[0,0]-->srv0::CRM.getOffers(..)[2,1]>
<SND 1257440666820839575 srv0::CRM.getOffers(..)[2,1]-->srv0::Catalog.getBook(..)[3,2]>
<RVC 1257440666840922990 srv0::Catalog.getBook(..)[3,2]-->srv0::CRM.getOffers(..)[2,1]>
<RVC 1257440666841169272 srv0::CRM.getOffers(..)[2,1]-->srv0::Bookstore.searchBook(..)[0,0]>
<RVC 1257440666841265860 srv0::Bookstore.searchBook(..)[0,0]-->$>
```

representing an operation call which starts the execution, and a return message, representing the end of an execution returning the control flow to the calling execution, as illustrated in Figure 7. For more details, refer to the associations sender and receiver in Figure 8. Listings 5 and 6 show text representations of the corresponding execution trace and a message trace stored by Kieker.Tpan, respectively.

Figure 8 shows the relations among executions, messages, execution traces, and message traces. While it is straightforward to derive execution traces from the Monitoring Log, for later analysis it is usually easier to derive analysis models from message traces. In Kieker.Tpan, execution traces are derived from the Monitoring Log and then transformed into equivalent message trace representations from which analysis models and diagrams, as described in the following Section III-B, are created.

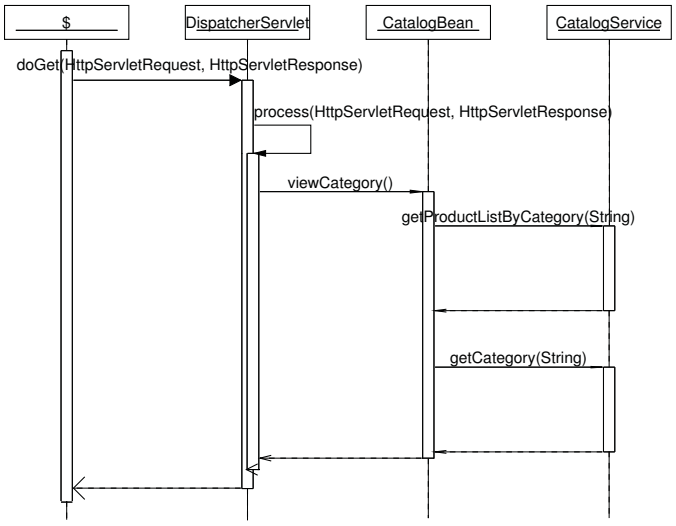


Fig. 9

UML SEQUENCE DIAGRAM GENERATED BY KIEKER.TPAN

B. Analysis and Visualization of Trace Information

As described in the previous Section III-A, Kieker.Tpmon allows to log (distributed) trace information to the Monitoring Log which can then be transformed into two equivalent trace representations using Kieker.Tpan, i.e., execution traces and message traces. These representations constitute the basis for trace-based analysis and visualization functionality which can be integrated into the Kieker.Tpan component. This section gives an overview of some of the trace-based analysis and visualization functionality which have been integrated into Kieker.Tpan so far: sequence diagrams, dynamic call trees, dependency diagrams, and Markov chains. By implementing a Monitoring Record Consumer, as described in Section II, it is easy to implement custom components for analyzing and visualizing trace information employing the Kieker framework.

1) UML sequence diagrams: UML sequence diagrams provide a dynamic architectural viewpoint in terms of interactions among runtime objects implementing software services. In Figure 7 of the previous section, we used a sequence diagram to illustrate the trace-related terminology needed to define

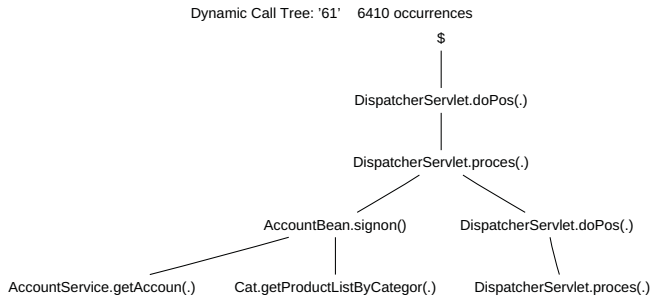


Fig. 10
CALL TREE CORRESPONDING TO A TRACE EQUIVALENCE CLASS GENERATED BY KIEKER.TPAN

execution traces and message traces. Message traces can be transformed to UML sequence diagrams in a straightforward way. Figure 9 shows such a UML sequence diagram generated by a Kieker visualization component for the iBATIS JPetStore which is a demo Java Web application implementing an online store scenario.[3] We employ this case study to give examples for visualizations in the present section. Given the timing information included in the execution traces, the sequence diagrams could easily be augmented with this additional data, e.g., observed response times. The UML specification [19] and the UML profiles for performance [21, 22] suggest appropriate notations for performance annotations.

2) Dynamic call trees: Figure 10 shows an alternative representation of a trace, called *dynamic call tree* [23], generated by Kieker.Tpan. A dynamic call tree contains the calling relations (call messages) among operations, in contrast to a sequence diagram which also includes the corresponding return messages. A dynamic call tree is an ordered graph, where the order (left to right) corresponds to the execution order of the nodes. In [17], we use the control flow information contained in a dynamic call tree to analyze the recorded response times of operations based on the call tree position of the corresponding executions.

When analyzing traces from the Monitoring Log, a valuable initial analysis is to determine the trace *equivalence classes*. Informally, a trace equivalence class contains all traces which are equal in terms of the control flow, i.e., the sequence diagrams of all traces in an equivalence class are identical. Based on the message traces this analysis can be implemented efficiently. Figure 10 shows the dynamic call tree common to all 6 410 traces in a trace equivalence class extracted from the monitoring data of the JPetStore example.

3) Dependency graphs: While sequence diagrams, or execution traces, provide a view on the *sequence* of interactions among objects in a single trace (or scenario/use case), it is often desirable to analyze this information in an aggregated form. Interactions among objects constitute runtime dependencies among these system entities, which can be described using weighted directed dependency graphs: each entity is assigned a node and each dependency relation an edge; the edge is directed from an entity using a particular service to the entity providing that service; the edges are augmented with the total number of call actions among the respective entities observed in the considered set of traces.

We implemented a Kieker.Tpan component which computes dependency graphs, represented in adjacency matrices, from a set of message traces. These dependency graphs are then available for further analysis or visualization. Figure 11 shows a dependency graph generated by Kieker.Tpan, visualizing calling dependencies among classes of the partly-instrumented JPetStore application. The figure provides an aggregated view of the runtime dependencies observed in 236 719 traces, resulting from 250 concurrent users simulated by probabilistic workload generation [24].

Dependency graphs are employed by some approaches to runtime reconfiguration [14] or failure diagnosis [12]. Figure 12 shows a dependency graph that has been enhanced with anomaly score information to support failure diagnosis. In this visualization, three architectural levels (operation, component, and deployment context) are displayed and small histograms show the distribution of the anomaly scores for each operation. Please refer to [12] for a detailed discussion; in the present paper this figure just serves as an illustration of possible visualizations.

4) Markov chains: Markov chains are a common formalism used in reliability and performance theory (see e.g. [25, 26]) to describe and analyze random processes, such as system and user behavior. A discrete-time Markov chain describes a process in terms of a discrete number of states and probabilis-

[3]The case study iBATIS JPetStore from http://ibatis.apache.org/ is available as an instrumented version on the Kieker web page http://kieker.sourceforge.net/.

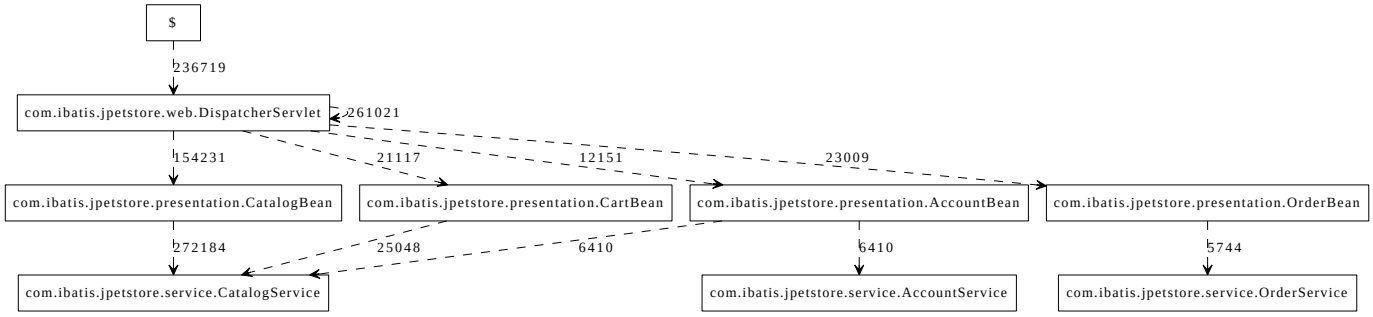


Fig. 11

COMPONENT DEPENDENCY GRAPH VISUALIZATION GENERATED BY KIEKER.TPAN

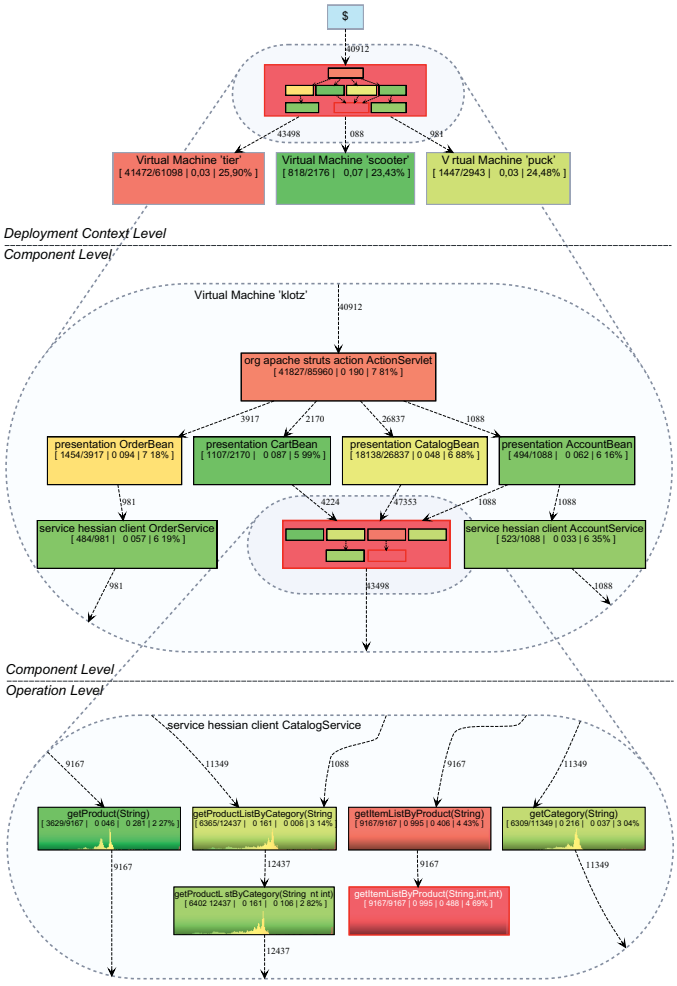


Fig. 12

VISUALIZED HIERARCHICAL DEPENDENCY GRAPH [12]

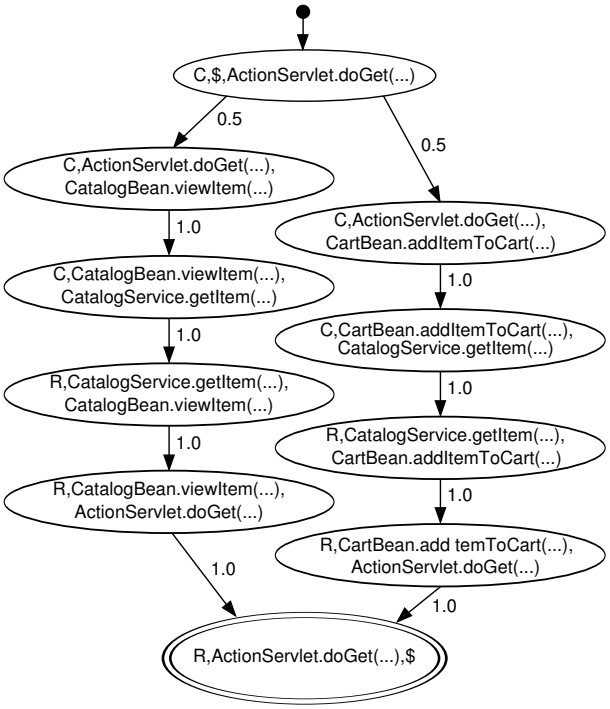


Fig. 13

OPERATION-LEVEL MARKOV CHAIN FOR TWO MESSAGE TRACES [15]

tic transitions–in discrete time steps–among these. The characteristic property of a Markov chain is that the next state of a process solely depends on the process's current state. An often used representation of Markov chains are (finite) state machines.

As illustrated in Figure 13, Kieker can generate finite state machine representations of Markov chains derived from a set of execution traces, where the states represent the creation of a message within a message trace and the edges connect subsequent messages. The edges are labeled with the relative

frequencies, derived from the monitoring data, that messages follow each other. A missing edge between two messages expresses that the monitoring data analyzed contains no trace with a sub-sequence containing only these two messages. Figure 19 in Appendix I shows a single Markov chain generated from the 236 719 traces of the JPetStore example.

Our method used to compute Markov chains from execution traces is described in [16]. In that work, we present an application-level intrusion detection system by comparing execution traces with normal behavior learned from monitoring data and modeled as Markov chains.

Listing 7
SINGLE CLASS APPLICATION

```
1  public class MonitoredClass {
2   public void monitoredOperation() {
3    /* spend 100 microseconds on computations */
4   }
5
6   public void monitoredRecursiveOperation
7                                (int recDepth) {
8     if (recDepth > 1) {
9      monitoredRecursiveOperation(recDepth - 1);
10    } else {
11     monitoredOperation()
12    }
13  }
14 }
```

IV. QUANTITATIVE OVERHEAD EVALUATION

This section provides a detailed quantitative assessment of Kieker's monitoring overhead using two micro benchmarks. Based on the framework's architectural description (see Section II), the goals are to quantify (1) the overhead caused by Kieker.Tpmon's components and to quantify (2) the framework's scalability with respect to monitoring traces.

First, a short overview of possible causes for overhead is given (IV-A), then the experiment design (IV-B) and the results of the two micro benchmarks are presented (IV-C).

A. Causes of Overhead

Figure 14 presents a UML sequence diagram of the typical framework-internal control flow for an operation measurement with Kieker. Before any of the actual operation code of the *monitoredOperation* in the *MonitoredClass* object is executed, the *triggerProbeBefore* part of the *KiekerMonitoringProbeImpl* is activated. Inside this probe, Kieker collects data, such as the current time and operation signature, before proceeding with the real operation code of the *monitoredOperation*. After the actual operation is finished, the *triggerProbeAfter* part of the *KiekerMonitoringProbeImpl* is activated. There, Kieker collects some additional data, such as the response time of the execution or the return values of the monitored operation. Finally, a Monitoring Record (see Section II-A) is prepared and added to an internal buffer by the *KiekerMonitoringWriterImpl*. This buffer is processed asynchronously by one or more *AsyncWriterThreads*. To keep the sequence diagram simple, the buffer is omitted in Figure 14.

Furthermore, the sequence diagram is annotated at the bottom with four different execution times, i.e., the time spent executing the actual code of the *monitoredOperation* (Δ_A); the time spent while triggering the probe before and after the actual operation (Δ_B); the time spent on collecting data about the monitored operation (Δ_C); and the time spent writing the data (Δ_D). The actual writing of data by the asynchronous writer thread will also have some impact on the overhead of Kieker and will be accounted for in Δ_D.

Thus, there may be three causes for overhead introduced by Kieker in addition to the execution time of the monitored service: (1) triggering of the probe (Δ_B), (2) collecting the data (Δ_C), and (3) writing the data (Δ_D). Section IV-C describes two lab experiments to quantify this overhead.

B. Experiment Design

The monitored application for the experiments consists of a single Java class *MonitoredClass*, as displayed in Listing 7, with two operations *monitoredOperation* and *monitoredRecursiveOperation*.

The experiments are performed on a modern enterprise server machine in our Software Performance Engineering Lab, in this case a X6270 Blade Server with two Intel Xeon 2.53GHz E5540 Quadcore processors and 24GB RAM running Solaris 10 and a SUN Java 64-bit Server VM in version 1.6.0_16-b01. Furthermore, AspectJ 1.6.6 with load-time weaving is utilized to insert the particular Monitoring Probes into the Java bytecode. Aside from the experiment, the server machine is held idle and not utilized.

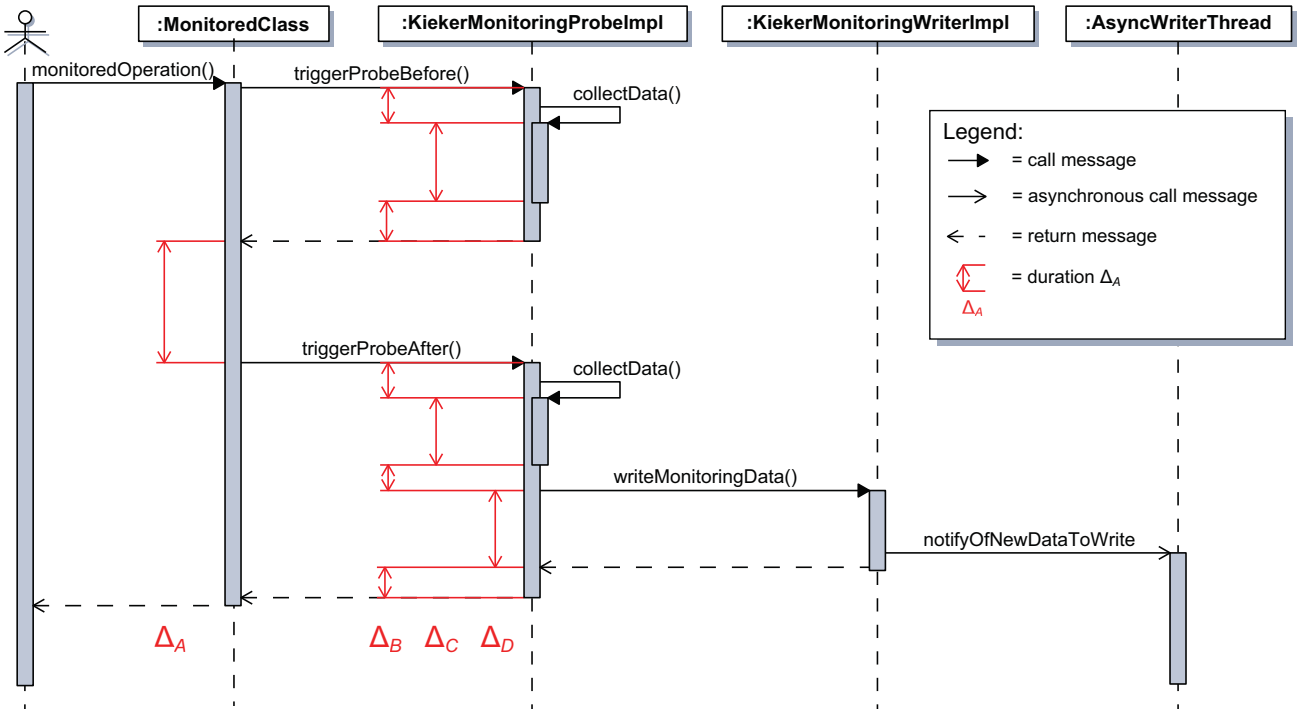


Fig. 14
UML SEQUENCE DIAGRAM FOR OPERATION MONITORING WITH KIEKER

Each experiment consists of four runs, one run for each measurement configuration. Thus, in the first experiment run only the execution time of the monitored operation (or chain of operations) is determined (Δ_A). In the second run, each monitored operation is instrumented with an empty AspectJ-based Monitoring Probe, thus the durations $\Delta_A + \Delta_B$ are measured. The third run adds the collection of data with an AspectJ-based Monitoring Probe capable of monitoring traces (see Section III-A) $(\Delta_A + \Delta_B + \Delta_C)$, while the fourth run finally represents the measurement of full monitoring with the addition of the asynchronous filesystem Monitoring Log Writer (*AsyncFSConnector*) (see Section II-C) $(\Delta_A + \Delta_B + \Delta_C + \Delta_D)$. With this experiment design, we can incrementally measure the three causes for monitoring overhead.

Each experiment run consists of two phases, a warm-up phase and a measurement phase. Both phases are, in principle, identical, but only measurements of the measurement phase are recorded for later analysis, since measurements taken during the warm-up phase are subject to significant variance in the timing behavior due to class loading, just-in-time compilation, etc. After each run, there is a short idle time to allow the system to calm down and to finish any open tasks.

The experiment code is available on Kieker's web page, such that interested researchers may repeat and extend our experiments.

C. Experiment Results

We report on the experiment results concerning the overhead caused by Kieker's components and the scalability of monitoring traces.

1) Overhead of Kieker's Components: This first experiment is designed to determine the overhead caused by Kieker.Tpmon's components when monitoring individual operations, here a call to *monitoredOperation*. The normal execution time of this operation (Δ_A) is about 100 microseconds. In each run of this experiment, a total of 100 000 000 operation calls is performed, but only the last 1 000 000 (i.e., after the warm-up phase) are recorded and analyzed.

The results of this experiment are displayed in Tables I and II, and in Figure 15. In addition to the box-and-whisker plot [27] of the experiment results, the particular mean values with their 95% confidence intervals (CI) are included. Furthermore,

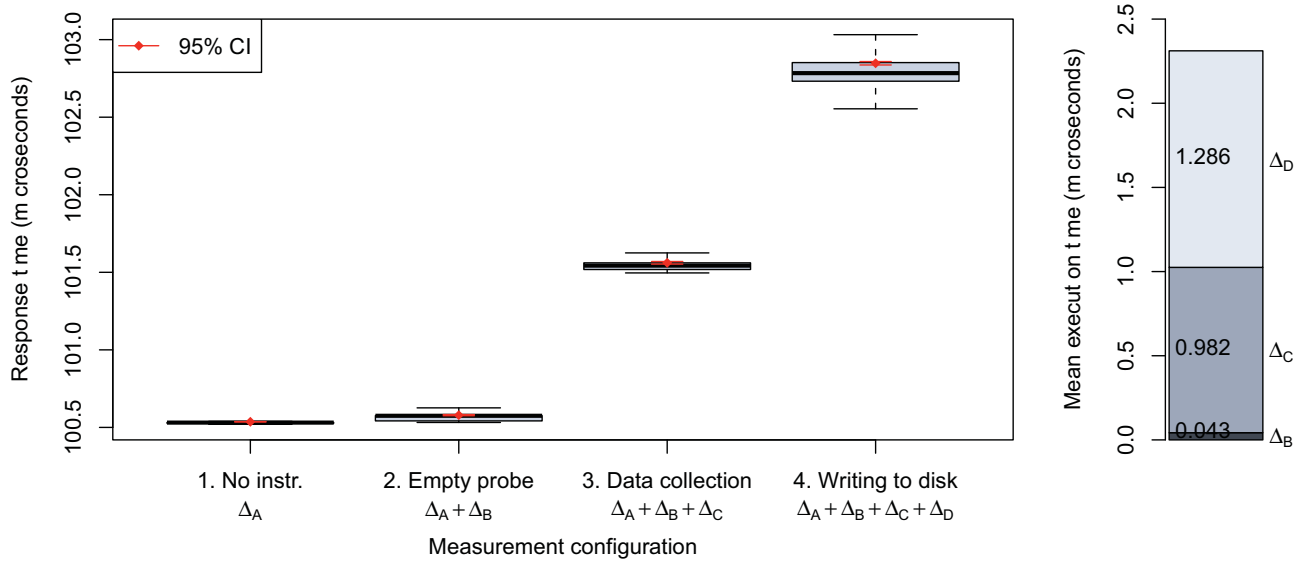


Fig. 15

RESULTS OF THE EXPERIMENT ON OVERHEAD OF KIEKER COMPONENTS ON A SINGLE MONITORED OPERATION

	1st run	2nd run	3rd run	4th run
1. Quartile	100.53	100.54	101.52	102.73
Median	100.53	100.58	101.54	102.78
3. Quartile	100.53	100.58	101.56	102.85
LCL Mean	100.54	100.58	101.55	102.84
Mean	100.54	100.58	101.56	102.85
UCL Mean	100.54	100.58	101.57	102.86
Std. dev.	0.59	1.85	4.26	5.65

TABLE I

RESULTS OF THE EXPERIMENT ON OVERHEAD OF KIEKER COMPONENTS ON A SINGLE MONITORED OPERATION (ALL VALUES ARE RESPONSE TIMES IN MICROSECONDS)

	Δ_B	Δ_C	Δ_D	Σ
Mean	0.043	0.982	1.286	2.311
Median	0.044	0.968	1.241	2.253

TABLE II

MEAN AND MEDIAN VALUES (IN MICROSECONDS) OF Δ_B, Δ_C, AND Δ_D AS RESULTS OF THE EXPERIMENT ON OVERHEAD OF KIEKER COMPONENTS ON A SINGLE MONITORED OPERATION

a staple diagram of the mean values of Δ_B, Δ_C, and Δ_D illustrates on the right hand side in Figure 15 the proportions of the overhead caused by the particular Kieker.Tpmon components.

The measured execution time Δ_A is, as expected, just about 100 microseconds. The addition of the empty Monitoring Probe ($\Delta_A + \Delta_B$) has a negligible effect of just about 40 additional nanoseconds. The collection of data with the trace-capable Monitoring Probe ($\Delta_A + \Delta_B + \Delta_C$) adds another microsecond, while recording monitoring data to the disk with the Monitoring Log Writer ($\Delta_A + \Delta_B + \Delta_C + \Delta_D$) costs another 1.3 microseconds. The offset of the mean value of Δ_D over the median of Δ_D hints at a greater variability of measured execution times, particularly higher execution times, due to hard disk access. In summary, the total, constant overhead for monitoring an operation with Kieker is under 2.5 microseconds on the X6270 Blade Server.

2) Scalability of Monitoring Traces: The second experiment is designed to assure that the measured overhead scales linearly with additional operation calls. A recursive operation call (*monitoredRecursiveOperation* in Listing 7) is used to simulate a chain of nested operation calls, typically occurring in traces. Each execution of *monitoredOperation* takes about 100 microseconds and each additional recursion step in *monitoredRecursiveOperation* adds only a few additional nanoseconds. To simulate different depths of traces, the experiment is repeated with recursion depths from 1 to 10. In each run of each repetition, a total of 100 000 000 recursive operation calls is performed, but only the last 100 000 are recorded and analyzed, as before.

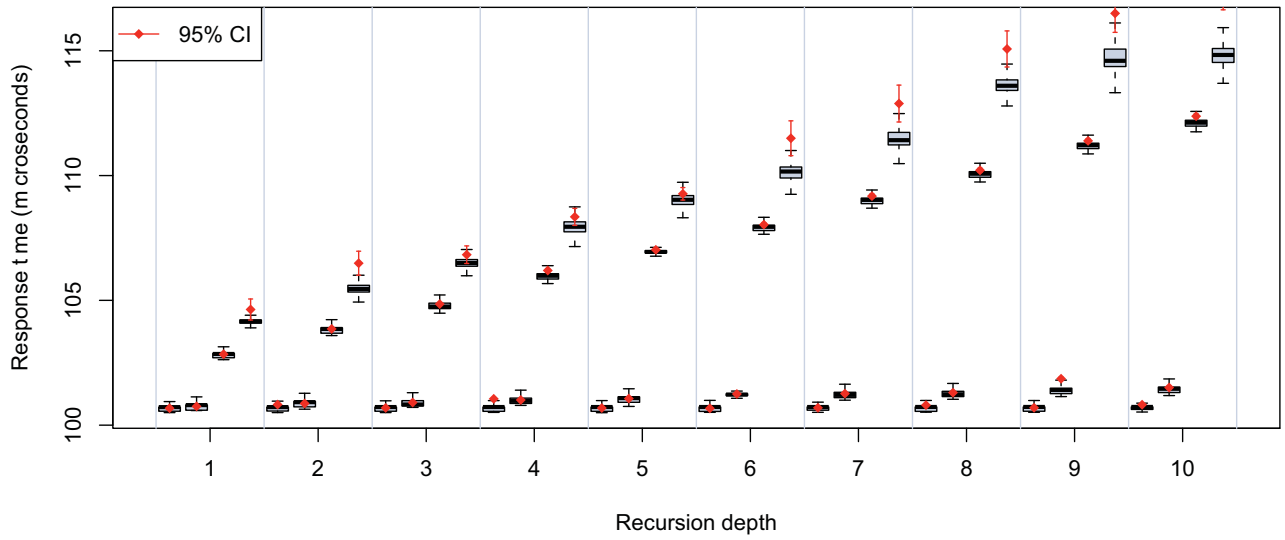


Fig. 16

RESULTS OF THE EXPERIMENT ON SCALABILITY OF MONITORING TRACES WITH KIEKER ON A `monitoredRecursiveOperation()` CALL. FOR EACH RECURSION DEPTH, THE RESULTS OF THE FOUR CONFIGURATIONS ARE SHOWN (ACCORDING TO FIGURE 15).

Recursion depth	1st run	2nd run	3rd run	4th run
1	100.68	100.75	102.84	104.63
2	100.82	100.87	103.86	106.49
3	100.69	100.91	104.85	106.83
4	101.05	101.01	106.20	108.35
5	100.69	101.07	107.02	109.26
6	100.67	101.25	108.03	111.49
7	100.69	101.26	109.18	112.89
8	100.80	101.30	110.21	115.07
9	100.70	101.86	111.39	116.50
10	100.82	101.51	112.38	117.86
correlation coefficient	-0.02	0.92	0.99	0.99

TABLE III

LINEAR DEPENDABILITY: CORRELATION OF RECURSION DEPTH AND RESPONSE TIME (ALL VALUES ARE MEAN RESPONSE TIMES IN MICROSECONDS) AS RESULTS OF THE EXPERIMENT ON SCALABILITY OF MONITORING TRACES

The recorded results of this experiment are displayed in Table III and Figure 16. Again, the measured execution time of Δ_A is just about 100 microseconds, the effect of additional recursion steps is negligible. The empty Monitoring Probe ($\Delta_A + \Delta_B$) and the collection of data with the trace-capable Monitoring Probe ($\Delta_A + \Delta_B + \Delta_C$) scale linearly (correlation coefficient is 92% and 99% respectively) with increasing recursion depths. Finally, the full instrumentation of the Kieker monitoring framework, including the recording of monitoring data to the hard disk with the Monitoring Log Writer ($\Delta_A + \Delta_B + \Delta_C + \Delta_D$), scales linearly as well (correlation coefficient is 99%), but there is a higher variance due to hard disk access, as can be seen in the box plot.

Furthermore, we were able to assure the linear scalability of monitoring traces with additional operation calls up to a recursion depth of 128, but we had to increase the *monitoredOperation*'s execution time to 1 millisecond. Additionally, the internal buffer used by the Monitoring Log Writer had to be increased significantly. Since monitoring a trace of operations up to a depth of 128 nested operations, taking a total response time of 1 millisecond, is not a realistic setting, we omitted the detailed results.

V. INDUSTRIAL EVALUATION

In the context of two case studies, a first with a digital photo service provider (CS-1) and a second with a telecommunication company (CS-2), the Kieker framework was evaluated by continuously monitoring operational software services in real-world systems. Both case study systems were enterprise-scale Web-based customer portals employing Java technology. This enabled an integration

of the Kieker.Tpmon component to monitor service-internal traces, as described in Section III. The goals of both case studies were to evaluate the practicality of such fine-grained monitoring under real workload conditions. In each case, the analysis of the filesystem Monitoring Log using Kieker.Tpan was performed offline.

In CS-1, Kieker.Tpmon was used during one week to log traces inside a single node of a load-balanced cluster of homogenous application servers. 161 operations were instrumented, covering a subset of the provided software services for ordering digital photo prints and other photo products. No observable overhead was reported comparing the performance of the instrumented node with the performance of the other cluster nodes which were exposed to the same workload conditions. CS-1 is not discussed in the present paper. Performance evaluation results of this case study can be found in [18].

The remainder of this section describes how Kieker.Tpmon is being used in the context of CS-2 to monitor distributed traces in a customer portal provided by the EWE TEL GmbH, one of the largest regional telecommunication providers in the north of Germany. The customer portal provides services such as account configuration to the customers. Section V-A describes the case study system's architecture; Section V-B describes how the system was instrumented using Kieker.Tpmon to log distributed trace information; and Section V-C presents some results of the analysis employing Kieker.Tpan.

A. System Architecture

Figure 17 illustrates the four-tier system architecture consisting of client, web server, application server and database sources. Front-end server nodes in the web presentation tier handle Web-based service requests. The front-end nodes request services from two application server nodes in the business-tier via SOAP Web service calls. The business-tier nodes access a replicated cluster of database servers as well as various services from the back-end tier, e.g., via SOAP or EJB calls.

The dashed rectangle in Figure 17 highlights the four server nodes that have been instrumented using Kieker.Tpmon: two front-end server nodes, referred to as FE0 and FE1 in this paper, as well as the two business-tier application server nodes, referred to AS0 and AS1. A hardware load balancer (H/W LB) distributes incoming HTTP requests to the homogeneous (in terms of hardware and software) replicated front-end nodes FE0 and FE1. Both FE0 and FE1 host three parallel customer portal instances and serve static and dynamic Web content. The portal instances are Java EE Web applications jointly deployed into an Apache Web Server/Java EE container installation. Both front-end nodes distribute the Web service calls to the homogeneous business-tier nodes AS0 and AS1 in a round-robin fashion (using Apache's *mod_proxy_balancer* module[4]). The Java application logic on all four nodes is implemented using the Spring framework[5] and employing Apache CXF[6] for calling and, respectively, providing Web services.

B. Instrumentation

Six Kieker.Tpmon instances on the front-end nodes (one for each portal) in addition to one Kieker.Tpmon instance per business-tier node, result in a total number of eight concurrent Kieker.Tpmon instances distributed over the four server nodes. Each instance is configured to use an asynchronous file system writer, as described in Section II, writing the Monitoring Log to the the local file system. Six different Monitoring Probe types are integrated on three layers of the software architecture to jointly monitor distributed execution traces across the server nodes of the front-end and business-tier. *KiekerExecutionRecords* are used as the Monitoring Record type representing the data of an execution, as described in Section III,

1) A Monitoring Probe on the front-end nodes intercepts incoming HTTP service requests and initializes the trace and session information for this request, including a unique trace identifier, as well as the initialization of the execution order index and execution stack size values (see Section III). After the execution of the actual request, the Monitoring Probe resets the trace information and logs the execution on the HTTP request level.
2) Two CXF Monitoring Probes on the front-end nodes are used to intercept outgoing Web service calls as well as the corresponding responses to and from the business-tier. For

[4]http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html
[5]http://www.springsource.org/
[6]http://cxf.apache.org/

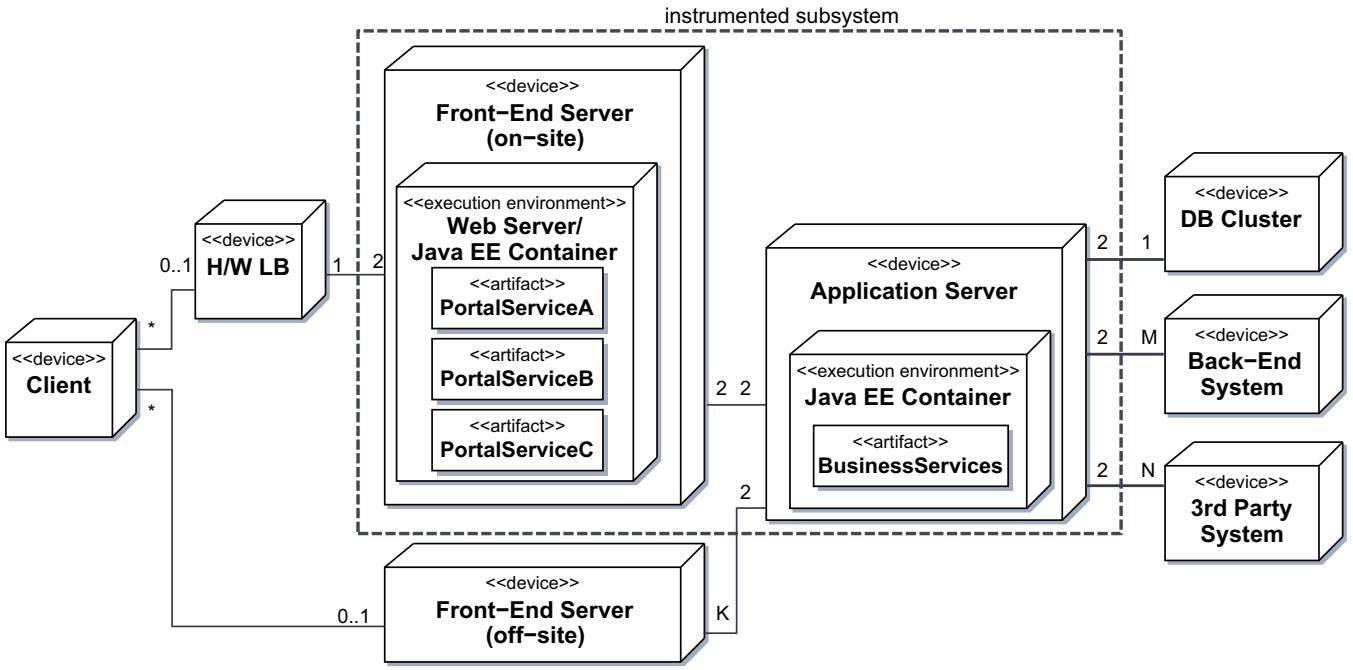


Fig. 17

FOUR-TIER ARCHITECTURE OF THE INDUSTRIAL CASE STUDY SYSTEM (CS-2).

each outgoing Web service call, the trace and session identifiers, as well as the execution order index value are integrated into the corresponding SOAP message. A trace's execution order index value is updated according to the value contained in the response message sent by the business-tier node.

Two corresponding CXF Monitoring Probes are integrated into the business-tier nodes to manage the trace and session information of incoming and outgoing SOAP messages.

A Web service call results in a logged execution on the calling front-end node and a logged execution on the called business-tier node.

3) A Spring-based Monitoring Probe intercepts and logs executions of the business service implementations on the business-tier nodes.

C. Trace Reconstruction Results

Figure 18 shows a sequence diagram of a distributed trace from the case study system. It was reconstructed and visualized by Kieker.Tpan, according to Section III. Here, it has been slightly shortened to fit on the page. It was possible to perform the analyses and to create the visualizations presented in Section III, since we used the same Monitoring Record type *KiekerExecutionRecord* to record the executions including the trace information. The trace in Figure 18 originates from a portal request dispatched by the load balancer to front-end node FE0. The trace is initiated by a call message to the operation *doFilter(..)* which is the Monitoring Probe intercepting the incoming HTTP request and initializing the trace and session information for this trace in the Kieker.Tpan instance on node FE0. The visualized part of the trace contains eight recorded remote calls from the front-end to the application server nodes AS0 and AS1. Each of these calls results in the same nesting of executions contained in the Monitoring Log: the recorded Web service call on the front-end and the corresponding Web service request on the application servers, followed by the execution of the requested business service on the respective application server.

Due to Kieker's distributed tracing functionality it is now possible to perform distributed tracing among front-end and business-tier nodes in the case study system, which is helpful for failure diagnosis. Moreover, the session information which was only available on the front-end nodes before, is now available to analyses regarding the business-tier.

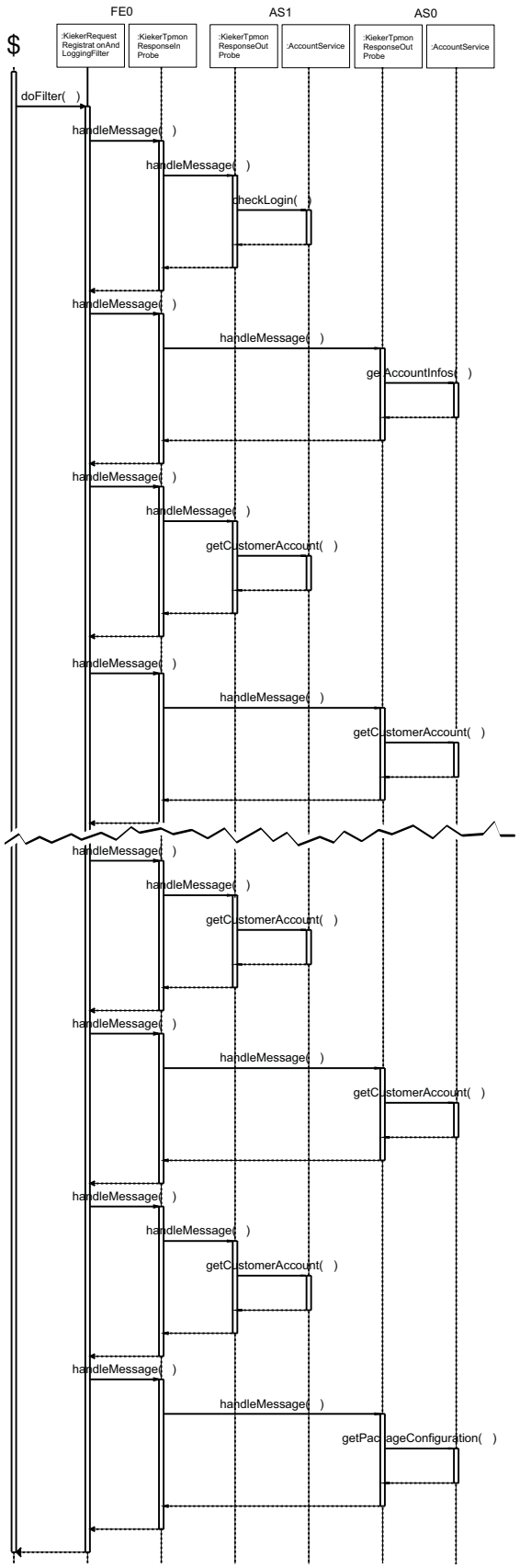


Fig. 18
RECONSTRUCTED SEQUENCE DIAGRAM OF A DISTRIBUTED TRACE SPANNING THREE SERVER NODES

In the previous Section IV, we present a detailed quantitative overhead evaluation of monitoring with Kieker in the controlled environment of our Software Performance Engineering Lab. In the industrial case studies, such controlled, quantitative measurements were not possible, since the load of these systems is out of our control. However, as a qualitative evaluation we can report that our industrial partners in both case studies could not observe any perceivable runtime overhead.

VI. REQUIRED DESIGN DECISIONS FOR MONITORING

From our experience with engineering of lab experiments and, particularly, industrial case studies, we observed that the integration of monitoring features into software services should be considered early in the engineering process [28]. In the following paragraphs we discuss several required design decisions concerning continuous monitoring.

Selection of Monitoring Probes: A monitoring probe contains the logic which collects and possibly pre-processes the data of interest of the application. Probes can measure externally visible behavior, e.g., service response times, but also application-internal behavior, such as calling dependencies between components. In practice, new application-internal monitoring probes are often only introduced in an ad-hoc manner, as a result of a system failure. For example, a probe for monitoring the number of available database connections in a connection pool may have been introduced. The selection of the types of monitoring probes must be driven by the goal to be achieved by the gathered monitoring data and depends on the analysis goals.

Number and Position of Monitoring Points: In addition to the above-mentioned decision of what types of monitoring probes are integrated into the system, important and highly challenging decisions concern the number and the exact locations of monitoring points. This decision requires a trade-off between the information quality available to the analysis tasks and the overhead introduced by possibly too fine-grained instrumentation yielding an extensive size of the monitoring log. The number and position of monitoring points also depend on the goal of monitoring. Additionally, the different usage scenarios of the application must be considered, since an equally distributed coverage of activated

monitoring points during operation is desirable. For instance, for failure diagnosis, the instrumentation is guided by the desired granularity of fault localization.

Intrusiveness of Instrumentation: A major maintainability aspect of application-level monitoring is how monitoring logic is integrated into the business logic. Maintainability is reduced if the monitoring code is mixed with the source code of the business logic, because this reduces source code readability. We consider the use of the AOP (Aspect-Oriented Programming) paradigm [8] as an extremely suitable means to integrate monitoring probes into an application. A popular Java-based AOP implementation is AspectJ. Many middleware technologies provide similar concepts, often based on interception. Examples are the definition of filters for incoming Web requests in the Java Servlet API, the method invocation interceptors in the Spring framework, or handlers for incoming and outgoing SOAP messages in different Web service frameworks. Kieker currently supports AOP-based monitoring probes with AspectJ, Spring, Servlets, and SOAP.

Physical Location of the Monitoring Log: The monitoring data collected within the monitoring probes is written to the so-called monitoring log. The monitoring log is typically located in the filesystem or in a database. The decision which medium to use depends on the amount of monitoring data generated at runtime, the required timeliness of analysis, and possibly restricted access rights or policies. A filesystem-based monitoring log is fast, since usually no network communication is required. The drawback is that online analysis of the monitoring data is not possible or at least complicated in a distributed setting. A database brings the benefit of integrated and centralized analysis support such as convenient queries but is slower than a file system log due to network latencies and the overhead introduced by the DBMS. Moreover, the monitoring data should not be written to the monitoring log synchronously since this has a considerable impact on the timing behavior of the executing business service. For online analysis, communication of monitoring data via messaging queues is required. Kieker includes different synchronous and asynchronous monitoring log writers for file system, database, and for message queues. Customized writers can be integrated into Kieker.

Monitoring Overhead: It is clear that continuous monitoring introduces a certain overhead to the running system. Of course, the overall overhead depends on the number of activated monitoring points and its activation frequency. The overhead for a single activated monitoring point depends on the delays introduced by the resource demand and process synchronizations in the monitoring probes, the monitoring control logic, and particularly I/O access for writing the monitoring data into the monitoring log. It is a requirement that the monitoring framework is as efficient as possible and that the overhead increases only linearly with the number of activated monitoring points, i.e., each activation of a monitoring point should add the same constant overhead. Of course, this linear scaling is only possible up to a certain point and depends on the average number of activated monitoring points, i.e., the granularity of instrumentation. In Section IV, we provide a detailed evaluation of Kieker's runtime overhead.

VII. Related Work

Related work includes monitoring frameworks and tracing in distributed systems (VII-A), dynamic analysis and architecture discovery (VII-B), and approaches for instrumentation with monitoring probes (VII-C).

A. Monitoring Frameworks and Tracing in Distributed Systems

An integrated monitoring framework considers two aspects: (1) instrumentation and logging of monitoring data, and (2) its subsequent analysis. Such an integrated approach for tracing runtime paths in Java EE systems is provided by the COMPAS JEEM tool [29]. With COMPAS, probes are inserted at deployment time as a proxy layer to the target components (e.g., EJBs at the business tier, Servlets or JSPs at the Web tier). The approach is non-intrusive and portable across any Java EE application server, but interception is limited to a level specified by the component interfaces.

Magpie [30, 31] monitors request resource consumption and component interactions on the software component level in distributed systems. Early versions of Magpie [31] used unique tokens to distinguish requests from each other. These tokens are propagated from one component to the next

to reconstruct traces. A later implementation of Magpie [30] replaced the token passing to a method based on events and timestamps in order to distinguish concurrent requests. In distributed systems, Magpie uses synchronization events between transmitted and received packets to connect request data spanning multiple system nodes. For non-distributed systems, Kieker supports both timestamp-based and token-based trace distinction, and in distributed systems, Kieker depends on token-passing while Magpie uses cross-machine synchronization events. Magpie has been implemented for Microsoft technology, while Kieker has been implemented for Java technology.

Basic principles of message tracking in distributed systems based on Web Service calls are in [32]. A more general algorithm called SAMEtech for merging traces in distributed systems is described by Israr et al. [33]. Similar to Kieker, it can connect traces in distributed systems independently from local clocks. SAMEtech reconstructs component interaction trees by correlating recorded event messages of method calls and returns. A limitation is that component internal parallelism (forking) is not supported. To be able to distinguish concurrent processes across nodes, specific requirements on the timestamp mechanisms, on message delivery, or on the message inherent information (particularly caller and callee references) have to be fulfilled. SAMEtech can derive Layered Queueing Networks from monitoring data. Israr et al. [33] assume that a tool provides traces, and outline how trace monitoring has to be performed in distributed systems. Kieker could be used to monitor such traces for distributed and non-distributed systems.

The Rainbow project [5] utilizes model-based monitoring in the context of architecture-based adaptation of software systems. Low level system information such as load or response time is measured by probes and can be published or queried. The system's architectural model is kept up-to-date at runtime applying the concept of so called *gauges* [34]. A gauge aggregates low level system information delivered by probes and interprets them in the context of higher-level models. Appropriate self-adaptation steps are determined and executed once a constraint violation is detected.

The dynamic analysis approach Pinpoint and follow-up publications [35–37] for problem determination perform runtime monitoring for failure diagnosis. Pinpoint tags client requests moving through the system and uses data mining techniques to correlate probable failures of these requests to determine which components are most likely to be faulty. Anomaly detection is an actual application of Kieker as well [12]. Pinpoint has no focus on performance analysis, in contrast to Kieker.

A recent survey [38] indicates that software engineers consider performance as a highly critical requirement, but at the same time monitoring tools that may be applied at the application level are seldom used in practice; particularly no open-source monitoring tools. Nagios,[7] for instance, is a popular monitoring tool for infrastructure monitoring, but it is not intended for application-level monitoring. Example commercial products are DynaTrace,[8] and JXInsight.[9]

B. Dynamic Analysis and Architecture Discovery

Related work in this section covers approaches for reconstructing dynamic as well as structural architecture views from runtime information. Studies concerning the execution of software services, known as dynamic analysis, have recently been surveyed by Cornelissen et al. [6]. Ducasse and Pollet [39] survey software architecture reconstruction methodologies including approaches exploiting runtime information.

Briand et al. [3] present an approach for reverse engineering of UML sequence diagrams from distributed Java software. Similar to our approach, AspectJ is suggested for instrumentation. The generated sequence diagrams are more detailed than in our approach: different object instances of one class are distinguished from each other and, beyond method entry and exit points, conditions and loops are instrumented manually. Kieker could easily be extended to gain comparable results by adding suitable probe and record types. In [3], instrumentation of remote calls is limited to RMI, while Kieker also supports Web services.

The tool VET [40] allows the visualization of previously recorded execution traces. Traces and dependencies are depicted in sequence and so-called class association diagrams. The latter are basically matrix views of the Markov Chains which can be generated by Kieker.

[7] http://www.nagios.org

[8] http://www.dynatrace.com/en/

[9] http://jinspired.com/products/jxinsight

Another reverse engineering environment for Java software systems called Shimba has been published by Systä et al. [41]. In contrast to the previous and our approach, Shimba combines static and dynamic analysis to achieve better reverse engineering results. It requires to run the target software under a customized debugger and is consequently not intended for continuous operation. Shimba allows to synthesize statechart diagrams automatically from sequence diagrams, which is not yet a capability, but a possible extension of Kieker.

Different perspectives on an application's runtime behavior have been proposed by de Pauw et al. [42]. The implementation of their former tool Jinsight has influenced parts of Eclipse's Test & Performance Tools Platform.[10] Several prior approaches, providing similar visualizations as Kieker, are surveyed in [43].

Two novel trace visualization techniques, called massive sequence and circular bundle view, are introduced with the Extravis tool by Cornelissen et al. [44]. These visualization techniques emphasize scalability assuming that vast amounts of data are collected at runtime.

Besides deriving dynamic lower level system views, several approaches consider static system properties and interpretation of system events at more abstract architectural level. Schmerl et al. [45] propose the approach DiscoTect for bridging the abstraction gap. DiscoTect applies a language for mapping implementation conventions to architectural styles. The mapping is translated to state machines, which are processed at runtime. Monitored system events serve as state machine triggers and initiate the construction of architectural model elements. Architectures are represented using the Acme architecture description language [46]. Monitoring in DiscoTect employs JPDA [47], which causes the target system to run 10 times slower compared to the non-instrumented system (see [45]) and is therefore not suitable for continuous operation, as Kieker is.

Walker et al. [48] include a mapping step for discovering architectures. They propose mapping software system traces to architectural views. The focus lies on providing an efficient encoding technique for dynamic trace information and the flexibility for manipulating traces from a variety of different architectural viewpoints.

[10] http://www.eclipse.org/tptp/

C. Monitoring Instrumentation

To observe and analyze the runtime behavior of a software system during operation, it has to be instrumented with monitoring probes. The probes collect information about the system's control and data flow at application level which is used for analysis. As monitoring is a cross-cutting concern, techniques that do not require source code intrusion, but inject monitoring aspects during the building process, are appropriate. There exists several related work concerning probe instrumentation techniques at application level.

The open source projects Glassbox[11] and InfraRED[12] use AspectJ for aspect-oriented instrumentation, as Kieker does. Both tools do not aim at tracing of requests across distributed Java EE application servers. Govindraj et al. [49] report that the performance overhead with InfraRED is usually between 1-5% of response times for a variety of enterprise Web applications and up to 10% if call trees are traced. As these numbers are very vague, we decided to quantify the monitoring overhead concerning Kieker with high precision, see Section IV.

AspectJ utilizes BCEL [50] for bytecode instrumentation. The InsECTJ project[13] [51], which is also based on BCEL, provides a generic instrumentation framework for collecting application runtime information. InsECTJ is implemented as an Eclipse plug-in, through which probes from a delivered catalog can be re-used to instrument an existing application. Another AOP framework is GluonJ [52], which is built on the Javassist bytecode manipulation library [53] and claims to be more suitable for expressing inter-component dependencies than AspectJ.

Alternative instrumentation approaches besides AOP, which have usually to be disposed at design time, are performance management specifications like JMX [54] or ARM [55]. Both specify a method and provide APIs for integrating enterprise applications as manageable entities, including means to measure performance and usage metrics.

Several approaches address the application of monitoring by means of integrating software system instrumentation into a model-driven engineer-

[11] http://www.glassbox.com/

[12] http://infrared.sourceforge.net/

[13] http://insectj.sourceforge.net/

ing (MDE) process. These approaches automatically generate instrumentation code from application models. Early work in this field is contributed by Klar et al. [56] who propose separate program models such as a functional program model, a functional implementation model, and a monitoring model. A number of tools have been developed to support the construction of graph models and automatic instrumentation of programs written in C. For instance, Huang and Steigner [57] consider the utilization of separate measurement models for C as an approach to model-driven instrumentation of parallel programs.

Approaches to model-driven instrumentation for monitoring SLAs employing standard metamodels [58, 59]), have been proposed for the domains of component-based [60] and service-based systems [61].

A model-based approach for semi-automatically generating probes and enforcing policies on services in a SOA environment is presented by Bai et al. [62]. MoDePeMART [63] is an approach for model-driven performance measurement and assessment. It suggests a domain specific language (DSL) for metrics specification. Instrumented code for data collection and storage is generated from models annotated with the DSL. The DSL has been realized as a UML profile for class and state diagrams.

Schaefer et al. [64] present an approach for model-driven instrumentation of distributed systems. Several instrumentation patterns specify roles, positioning, and dependencies of monitoring points and have been implemented as a UML profile. It should be possible to integrate such model-driven approaches with Kieker, such that Kieker's instrumentation code would be generated from design models.

VIII. Conclusions

In this paper, we present the Kieker framework for continuous monitoring of software services. Kieker provides a common monitoring record model for data collection and subsequent analyses. Distributed environments are supported, such that service requests, which cross different execution containers, can be traced. The framework provides an extensible architecture and supports the construction of additional monitoring record types, probes, analyses, and visualizations.

We report on extensive lab studies as well as on industrial case studies with operational software services of a telecommunication customer self service and a digital photo submission service. The investigation of Kieker's runtime overhead has been a specific focus in our qualitative and quantitative evaluation. The experimental evaluation showed that Kieker imposes only a small linear overhead, and that it is applicable in industrial settings. Together with the source code, our experiment data is available such that interested researchers can repeat and extend our lab experiments.[14]

In addition to the presented micro benchmarks, we are currently analyzing the overhead of monitoring for the industrial standard SPECjAppServer2004 benchmark.[15] SPECjAppServer2004 is a Java EE benchmark, meant to measure the performance and scalability of Java EE application servers. Usually, two or more different application server configurations run the benchmark suite and are compared. Conversely, we intend to deploy a fixed configuration and modify the benchmark, first the normal benchmark is executed and its throughput performance is recorded, then we instrument the benchmark application to various degrees with monitoring probes and analyze the performance overhead in that setting.

As future work we also intend to address the following issues:

- Model-driven instrumentation: We will investigate the combination of Kieker with model-driven instrumentation [63].
- Architecture compliance checking: We intend to compare prescriptive architectural models, which are designed in forward engineering, with architectural models reconstructed based on monitored runtime information [65].
- Business activity monitoring: So-called key performance indicators (KPIs) are monitored for continuous business activity monitoring [66]. In collaboration with a local software company, we are currently investigating how monitoring probes for such KPIs may be integrated into Kieker for business activity monitoring.
- Interactive visualization: As existing visualizations, such as hierarchical dependency graphs, tend to become complex for real systems, we intend to extend their visual tools with interactive features.

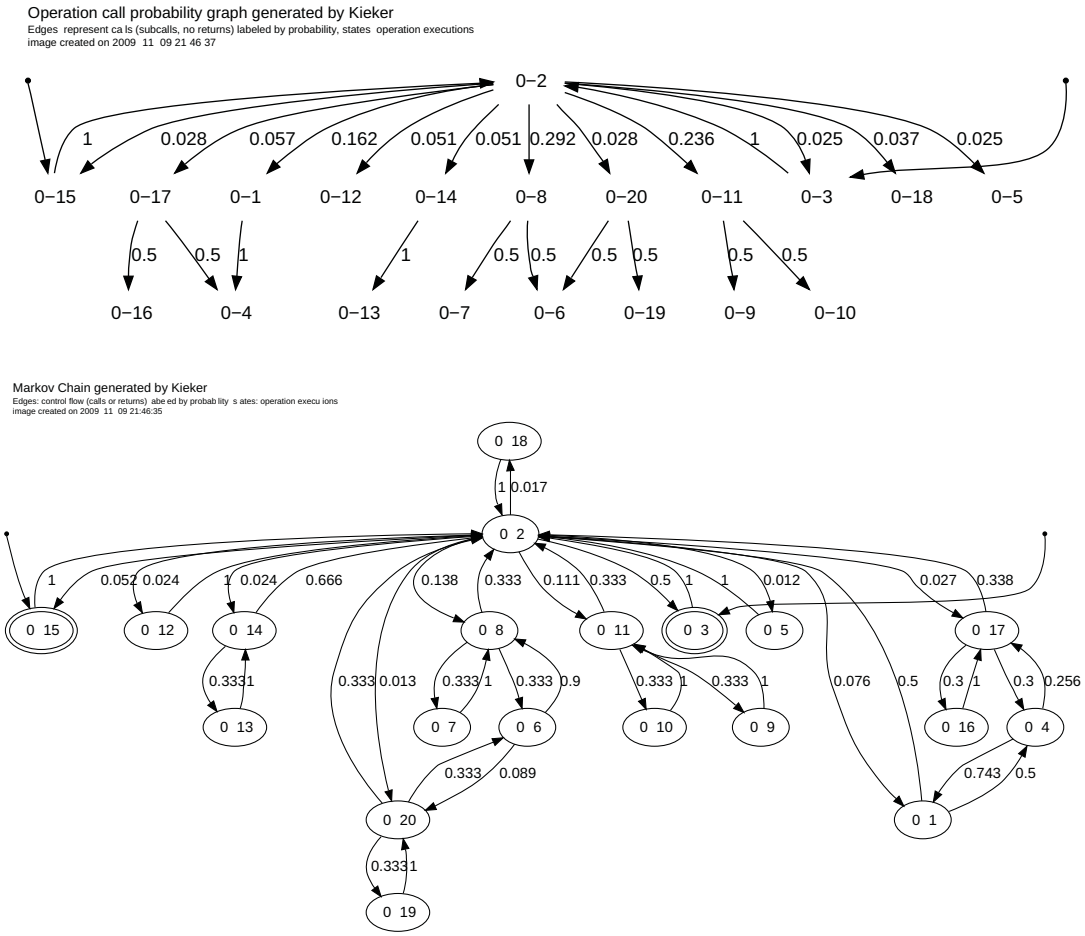
[14] http://kieker.sourceforge.net/

[15] http://www.spec.org/osg/jAppServer2004

ACKNOWLEDGMENT

This work is supported by the German Research Foundation (DFG), grant GRK 1076/1. In addition, the authors would like to thank Thilo Focke for implementing and deploying the first version of Kieker at EWE TEL, Marco Lübcke for deploying a later version of Kieker at CeWe Color, and Sergej Alekseev for providing us with monitoring data from Nokia Siemens Networks for analysis with Kieker. Thanks to Nils Sommer, Lena Stöver, Robert von Massow, and Nina Marwede for contributing various code parts to Kieker.

APPENDIX I
OPERATION CALL GRAPH AND MARKOV CHAIN




```
Op 0−1    com.ibatis.jpetstore.presentation.CatalogBean.viewItem()
Op 0−2    com.ibatis.jpetstore.web.DispatcherServlet.process(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Op 0−3    com.ibatis.jpetstore.web.DispatcherServlet.doGet(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Op 0−4    com.ibatis.jpetstore.service.CatalogService.getItem(java.lang.String)
Op 0−5    com.ibatis.jpetstore.presentation.AccountBean.signoff()
Op 0−6    com.ibatis.jpetstore.service.CatalogService.getProductListByCategory(java.lang.String)
Op 0−7    com.ibatis.jpetstore.service.CatalogService.getCategory(java.lang.String)
Op 0−8    com.ibatis.jpetstore.presentation.CatalogBean.viewCategory()
Op 0−9    com.ibatis.jpetstore.service.CatalogService.getItemListByProduct(java.lang.String)
Op 0−10   com.ibatis.jpetstore.service.CatalogService.getProduct(java.lang.String)
Op 0−11   com.ibatis.jpetstore.presentation.CatalogBean.viewProduct()
Op 0−12   com.ibatis.jpetstore.presentation.OrderBean.newOrderForm()
Op 0−13   com.ibatis.jpetstore.service.OrderService.insertOrder(com.ibatis.jpetstore.domain.Order)
Op 0−14   com.ibatis.jpetstore.presentation.OrderBean.newOrder()
Op 0−15   com.ibatis.jpetstore.web.DispatcherServlet.doPost(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
Op 0−16   com.ibatis.jpetstore.service.CatalogService.isItemInStock(java.lang.String)
Op 0−17   com.ibatis.jpetstore.presentation.CartBean.addItemToCart()
Op 0−18   com.ibatis.jpetstore.presentation.CartBean.viewCart()
Op 0−19   com.ibatis.jpetstore.service.AccountService.getAccount(java.lang.String,java.lang.String)
Op 0−20   com.ibatis.jpetstore.presentation.AccountBean.signon()
```

Fig. 19

OPERATION CALL GRAPH AND MARKOV CHAIN CONSTRUCTED FROM 236 719 TRACES OF AN INSTRUMENTED IBATIS JPETSTORE.

REFERENCES

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004)*, volume 3278 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2004.

[2] T. Parsons, A. Mos, M. Trofin, T. Gschwind, and J. Murphy. Extracting interactions in component-based systems. *IEEE Transactions on Software Engineering*, 34(6):783–799, Nov.-Dec. 2008.

[3] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9): 642–663, September 2006.

[4] A. Diaconescu and J. Murphy. Automating the performance management of component-based enterprise systems through the use of redundancy. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 44–53. ACM, 2005.

[5] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[7] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 111–121. IEEE Computer Society, 2009.

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, pages 327–353. Springer, 2001.

[10] M. Dessi. *Spring 2.5 Aspect Oriented Programming*. Packt Publishing, 2009.

[11] FUSE Open Source Community. FUSE Service Framework Documentation (Apache CXF Documentation). http://www.fusesource.com, 2009.

[12] N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, pages 47–57. IEEE Computer Society, March 2009.

[13] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In *Proceedings of the 2nd Warm-Up Workshop for ACM/IEEE ICSE 2010 (WUP '09)*, pages 41–44. ACM, April 2009.

[14] J. Matevska and W. Hasselbring. A scenario-based approach to increasing service availability at runtime reconfiguration of component-based systems. In *Proceedings of 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 137–144. IEEE Computer Society, August 2007.

[15] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stöver, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*, pages 80–85. ACTA Press, February 2008.

[16] I. A. Gul, N. Sommer, M. Rohr, A. van Hoorn, and W. Hasselbring. Evaluation of control flow traces in software applications for intrusion detection. In *Proceedings of the 12th IEEE International Multitopic Conference (IEEE INMIC 2008)*, pages 373–378. IEEE, 2008.

[17] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, and S. Alekseev. Trace-context sensitive performance profiling for en-

terprise software applications. In *Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*, volume 5119 of *Lecture Notes in Computer Science*, pages 283–302. Springer, June 2008.

[18] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In *1st Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW 2010)*. ACM, January 2010. To appear.

[19] Object Management Group (OMG). OMG Unified Modeling Language Superstructure Version 2.2. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/, February 2009.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] Object Management Group (OMG). UML Profile for Schedulability, Performance, and Time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02, January 2005.

[22] Object Management Group (OMG). UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), Beta 2. OMG adopted specification ptc/08-06-08. http://www.omg.org/cgi-bin/doc?ptc/2008-06-08, June 2008.

[23] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '97)*, pages 85–96. ACM, 1997.

[24] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for Web-based software systems. In *Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*, volume 5119 of *Lecture Notes in Computer Science*, pages 124–143. Springer, June 2008.

[25] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Wiley & Sons, 2nd edition, 2001.

[26] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall, 2004.

[27] D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers*. Wiley & Sons, 4th edition, 2006.

[28] A. van Hoorn, W. Hasselbring, and M. Rohr. Engineering and continuously operating self-adaptive software systems: Required design decisions. In *Proceedings of the 1st Workshop "Design for Future Workshop" (L2S2)*, volume 537 of *CEUR Workshop Proceedings*, pages 52–63, November 2009.

[29] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end to end run-time path tracing for J2EE systems. In *IEE Proceedings-Software*, 2006.

[30] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *6th Symposium On Operating Systems Design and Implementation (OSDI '04)*, pages 259–272, 2004.

[31] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HOTOS '03)*, pages 85–90. USENIX Association, 2003.

[32] A. Sahai, V. Machiraju, J. Ouyang, and K. Wurster. Message tracking in SOAP-based Web services. In *Network Operations and Management Symposium*, pages 33–47. IEEE, 2002.

[33] T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(5):474–492, 2007.

[34] D. Garlan and B. Schmerl. Using architectural models at runtime: Research challenges. In *Software Architecture – Proceedings of the 1st European Workshop on Software Architecture (EWSA 2004)*, volume 3047 of *Lecture Notes in Computer Science*, pages 200–205. Springer, May 2004.

[35] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Con-*

ference on Dependable Systems and Networks (DSN '02), pages 595–604. IEEE Computer Society, 2002.

[36] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks*, 16(5):1027–1041, September 2005.

[37] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(2):175–190, 2006.

[38] R. G. Snatzke. Performance survey 2008. http://www.codecentric.de/export/sites/www/_resources/pdf/performance-survey-2008-web.pdf, 3 2009.

[39] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[40] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (VET): An interactive plugin-based visualisation tool. In *Proceedings of the 7th Australasian User Interface Conference (AUIC '06)*, pages 153–160. Australian Computer Society, Inc., 2006.

[41] T. Systä, K. Koskimies, and H. Müller. Shimba – an environment for reverse engineering Java software systems. *Softw. Pract. Exper.*, 31(4): 371–394, 2001.

[42] W. de Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Revised Lectures on Software Visualization*, pages 151–162. Springer, 2002.

[43] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 42–55. IBM, 2004.

[44] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *The Journal of Systems and Software*, 81(12): 2252–2268, 2008.

[45] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, July 2006.

[46] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[47] Sun Microsystems, Inc. Java platform debugger architecture. http://java.sun.com/javase/technologies/core/toolsapis/jpda/, 2009.

[48] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '00)*, page 12. IBM Press, 2000.

[49] K. Govindraj, S. Narayanan, B. Thomas, P. Nair, and S. P. On using AOP for application performance management. In *Proceedings of the 5th Internation Conference on Aspect-Oriented Software Development (AOSD 2006), Industry Track*, pages 18–30, March 2006.

[50] M. Dahm. Byte code engineering. In *Java-Information-Tage (JIT '99)*, pages 267–277. Springer, 1999.

[51] A. Seesing and A. Orso. Insectj: A generic instrumentation framework for collecting dynamic information within Eclipse. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 45–49. ACM, 2005.

[52] S. Chiba and R. Ishikawa. Aspect-oriented programming beyond dependency injection. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, pages 121–143. Springer, 2005.

[53] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 313–336. Springer, 2000.

[54] Sun Microsystems, Inc. Java management extensions (JMX) technology. http://java.sun.com/products/JavaManagement/, 2009.

[55] M. W. Johnson. Monitoring and diagnosing application response time with ARM. In *Proceedings of the IEEE 3rd International Workshop on Systems Management (SMW '98)*, page 4. IEEE Computer Society, 1998.

[56] R. Klar, A. Quick, and F. Sötz. Tools for a model-driven instrumentation for monitoring.

In *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 165–180. Elsevier, 1991.

[57] X. Huang and C. Steigner. A model-driven tool for performance measurement and analysis of parallel programs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '95)*, pages 612–617, 1995.

[58] Object Management Group (OMG). UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. http://www.omg.org/spec/QFTP/, April 2008.

[59] Distributed Management Task Force. Common Information Model (CIM) standard. http://www.dmtf.org/standards/cim/, May 2009.

[60] K. Chan and I. Poernomo. QoS-aware model driven architecture through the UML and CIM. *Information Systems Frontiers*, 9(2-3): 209–224, 2007.

[61] C. Momm, T. Detsch, and S. Abeck. Model-driven instrumentation for monitoring the quality of web service compositions. In *Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW '08)*, pages 58–67. IEEE Computer Society, 2008.

[62] X. Bai, Y. Liu, L. Wang, and P. Zhong. Model-based monitoring and policy enforcement of services. *Simulation Modelling Practice and Theory*, 17(8):1399–1412, 2009.

[63] M. Boskovic and W. Hasselbring. Model driven performance measurement and assessment with MoDePeMART. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, volume 5795 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2009.

[64] J. Schaefer, J. Stynes, and R. Kroeger. Model-Based Performance Instrumentation of Distributed Applications. In *Proceedings of the 8th IFIP WG 6.1 International Conference Distributed Applications and Interoperable Systems (DAIS 2008)*, pages 210–223. Springer, June 2008.

[65] D. Ganesana, T. Keulerb, and Y. Nishimura. Architecture compliance checking at run-time. *Information and Software Technology*, 51(11): 1586–1600, November 2009.

[66] B. Wetzstein, P. Leitner, F. Rosenberg, I. Brandic, S. Dustdar, and F. Leymann. Monitoring and analyzing influential factors of business process performance. In *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference (EDOC '09)*, pages 141–150, 2009.