Towards Adaptive Monitoring of Java EE Applications

Dušan Okanović[#1], André van Hoorn[*2], Zora Konjović[#3], and Milan Vidaković[#4]

[#]*Faculty of Technical Sciences, University of Novi Sad*
Fruškogorska 11, 21000 Novi Sad, Serbia
{[1]oki,[3]ftn_zora,[4]minja}@uns.ac.rs

[*]*Software Engineering Group, University of Kiel*
Christian-Albrechts-Platz 4, 24098 Kiel, Germany
[2]avh@informatik.uni-kiel.de

***Abstract*—Continuous monitoring of software systems under production workload provides valuable data about application runtime behavior and usage. An adaptive monitoring infrastructure allows to control, for instance, the overhead as well as the granularity and quality of collected data at runtime. Focusing on application-level monitoring, this paper presents how we extended the monitoring framework Kieker by reconfiguration capabilities based on JMX technology. The extension allows to change the instrumentation of software operations in monitored distributed Java EE applications. As a proof-of-concept, we demonstrate the adaptive monitoring of a distributed sample Java EE application deployed to a JBoss application server.**

***Keywords*-Continuous monitoring, adaptive monitoring, aspect-oriented programming, Java EE, JMX**

I. Introduction

Software testing, debugging, and profiling in development environments hardly allow to detect errors and unpredicted events that can occur after the software is deployed and used in its production environment. While new, previously unknown, errors can show up, it is a common phenomenon for software performance and quality of service to degrade over time [1], too. To determine whether the quality of service and service level agreements are on a satisfactory level, it is necessary to monitor software in its operational stage and environment.

In the development phase, software developers usually utilize tools such as debuggers and profilers. Although they provide a picture of the software behavior, they typically induce a significant performance overhead—something which is unacceptable for production use. Dynamic behavior of the software can, for instance, be analyzed by reading source code or UML diagrams [2] that are produced in the design phase. The reading of source code is inconvenient because of its complexity. Also, architectural diagrams are often incomplete or missing.

To determine how software behaves under production workload, continuous monitoring of that software is a valuable option. Continuous monitoring of software is a technique that provides a picture of the dynamic behavior of software under real usage, but often results in a large amount of data. In the process of the analysis, the obtained data can be used to reconstruct architectural models and perform their visualization (e.g., employing UML).

One important issue regarding monitoring is the induced performance overhead. During monitoring, resources in the system are shared among the components of the monitored systems as well as those of the monitoring system. Thus, the monitored system has to fulfill its functionality with fewer resources. This resource contention eventually affects the performance of the monitored system, e.g., in terms of increased response times. Monitoring overhead can be controlled using adaptive techniques. These techniques allow a reduction of overhead by shutting down monitoring in parts of the system, if those parts are not of particular interest at the moment.

The main contribution of this paper is that it presents how the open-source monitoring and analysis framework Kieker [1] can be used for continuous and adaptive monitoring of enterprise Java EE (JEE) applications. In order to achieve this, we created additional JMX [3] components that allow to reconfigure monitoring parameters, e.g., the set of instrumented software operations, during the monitoring process. This combination of Kieker and JMX enables adaptive monitoring. During operation, we can disable and enable monitoring of parts of the application, particularly the instrumentation of software operations, to reduce overhead or to obtain more information, respectively. In this paper, we present the integration of the Kieker framework into the JBoss application server [4] and our implementation of additional JMX components required to achieve reconfigurable and adaptive monitoring. We discuss the instrumentation of a sample JEE application using AspectJ-based [5] probes from the Kieker framework. Also, results of monitoring this JEE application are shown.

The remainder of this paper is structured as follows. Section II provides an overview of related work in the field of performance monitoring. Section III introduces the basic concepts of the Kieker framework. Our extension enabling adaptive monitoring is described in Section IV. Monitoring of a sample application deployed to a JBoss application server using the Kieker framework and the developed extensions is shown in Section V. Section VI draws the conclusions and outlines future work.

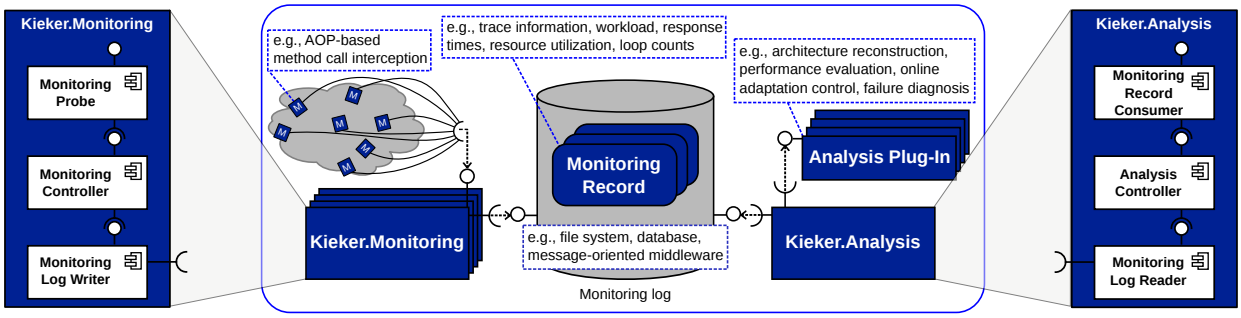


Fig. 1. Kieker framework architecture

II. Related Work

A recent study presented by Snatzke [6] indicates that performance is considered critical in practice, but developers rarely use monitoring tools. In general, application-level monitoring tools, and especially open-source tools, are rarely used. Typical reasons for this are time pressure (during development) and resource constraints during application use.

Apart from Kieker, which is used in this paper and described in Section III, there are several other systems that allow to profile and monitor JEE applications. For example, JBossProfiler [7] is a tool based on the JVMTI [8]/JVMPI [9] API. It can be used to monitor applications deployed to the JBoss application server. The use of this API provides precise results but induces significant overhead. COMPAS JEEM [10] inserts software probes during the application startup. The probes can be inserted into any layer of JEE applications (EJB, Servlet etc.). The advantage of this approach is that there is no need for application source code changes. However, a drawback of this approach is the fact that different probes must be defined for each application layer. The HotWave tool [11] allows runtime re-weaving, similar to AspectJ [5]. It can be used for the implementation of different profiling tools. The downside is that, due to some restrictions, it does not support *around* advices. The advice, however, can be replaced by combining *before* and *after* advices and the use of inter-advice communication. The tool is still in development. The approach introduced by Briand et al. [12] is used for reconstructing UML sequence diagrams from JEE applications. The instrumentation is performed employing AspectJ, as is the case for Kieker. The system is limited to diagram generation. It is not suitable for continuous monitoring and it is not able to monitor web services, only RMI. SAMEtech [13] conducts monitoring in a similar way to Kieker. It uses numeration of executions within a trace, but cannot monitor concurrent executions within an application. Similar to the work presented in this paper, Ehlers and Hasselbring [14] present an approach for Kieker allowing to activate and deactivate monitoring probes at runtime.

DynaTrace [15] and JXInsight [16] are examples of commercially available application monitoring tools. One of the open-source tools in use is Nagios [17] which is not intended for application level monitoring, but to monitor system infrastructure.

This overview shows the lack of tools (especially non-commercial open-source tools) that allow continuous and reconfigurable monitoring of JEE applications with low overhead. The Kieker framework [1] in combination with JMX [3] can be used for adaptive and reconfigurable continuous monitoring of JEE applications, as presented in this paper. Among other technologies, Kieker uses AspectJ for instrumentation which provides a separation of monitoring code from the application code. JMX, which is in the core of the JEE application server infrastructure, can be used for controlling the monitoring process at runtime. Changing of monitoring parameters during the monitoring process allows to implement an infrastructure for adaptive and reconfigurable monitoring, as presented in this paper.

III. Kieker Framework

Kieker [1] is a framework for continuous monitoring and analysis of software systems, currently focusing on Java-based applications. It consists of the Kieker.Monitoring and the Kieker.Analysis components. The Kieker.Monitoring component collects and stores monitoring data. The Kieker.Analysis component performs analysis and visualization of this monitoring data. The architecture of the Kieker framework is depicted in Fig. 1.

The Kieker.Monitoring component is executed on the same computer where the monitored application is being run. This component collects data during the execution of the monitored applications. A Monitoring Probe is a software sensor that is inserted into the observed application and takes various measurements. For example, Kieker includes probes to monitor control-flow and timing information of method executions. Monitoring Log Writers store the collected data, in the form of Monitoring Records, in a Monitoring Log. The framework is distributed with Monitoring Log Writers that can store Monitoring Records in file systems, databases, or JMS queues. Additionally, users can implement and use their own writers. A Monitoring Controller controls the work of this part of the framework.

The data in the Monitoring Log is analyzed by the Kieker.Analysis component. A Monitoring Log Reader reads records from the Monitoring Log and forwards them to Analysis Plugins. Analysis Plugins may, for example, analyze and visualize gathered data. Control of all components in this part of the Kieker framework is performed by the Analysis Controller component.

Both components of the Kieker framework work completely independently. This approach allows a cluster of servers to run the monitored software, to store monitoring data in a file system or database on another server, and to perform data visualization and analysis—possibly on yet another server.

Program instrumentation in the Kieker framework is usually performed using aspect-oriented programming (AOP) [18]. This way, developers can separate program logic from monitoring logic (separation of concerns). Instrumentation consists of writing aspect classes and weaving them with application classes. These aspects intercept the execution of program logic at defined points (so-called join points) and add additional behavior (defined in advices).

Kieker can monitor each method in every class or only designated ones. Developers can use aspects that monitor all methods or only methods annotated with annotations. The `@OperationExecutionMonitoringProbe` annotation and several different aspects are distributed with the framework which allows for the creation of different monitoring scenarios. Users can also use their own aspects and annotations.

Probes distributed with the framework, intended for monitoring executions of methods, generate Monitoring Records that are instances of the `OperationExecutionRecord` class. Every instance contains the name of a component for which it is generated, as well as method name, session id, trace id, execution start and end time.

Regardless of the chosen scenario, the aspect intercepts the executed method, takes necessary measurements, lets the method execute, creates a Monitoring Record and stores data into the Monitoring Log using the Monitoring Controller. Within one application there can be multiple annotations and aspects, so that they can perform complementary measurements.

IV. Kieker Extension for Adaptive Monitoring

The framework extension for adaptive monitoring and its configuration for use with the JBoss server was done by implementing a new Monitoring Log Writer and by adding new JMX components. A deployment diagram for this system is depicted in Fig. 2.

DProfWriter is a new writer which stores all records into a special buffer—the ResultBuffer. Kieker's Monitoring Controller is configured to use the DProfWriter. The ResultBuffer is implemented as a JMX MBean and relies on the JBoss microkernel infrastructure. The DProfWriter sends records to the ResultBuffer through the MBeanServer. The buffer sends data to a Record Receiver service running on a remote server. Data can be sent periodically (in bulks) or as soon as they arrive into the buffer. This remote service stores records into a relational database for further analysis. Essentially, in this case, the combination of the buffer, the service and the database constitutes Kieker's Monitoring Log.

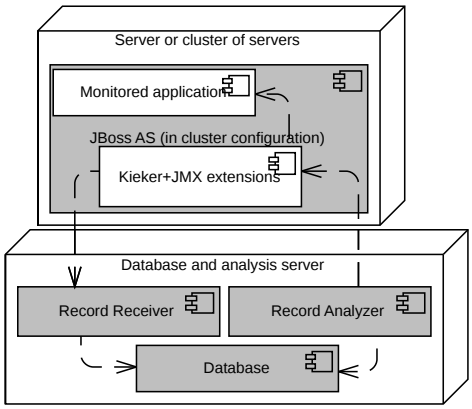


Fig. 2. Deployment diagram of the system. DProfWriter, ResultBuffer, AspectController and DProfManager are omitted; they are parts of the Kieker+JMX extensions component

The DProfManager component, implemented as a JMX MBean, is used to control the monitoring process. It controls the ResultBuffer and an AspectController component. The AspectController component is used to change weaving parameters defined in the AspectJ configuration file (*aop.xml*). It is implemented as a JMX MBean, too. The AspectController can access the monitored application's *aop.xml* file, parse it, change parameters and save changes in the application archive (jar/war/ear) file. This will change the timestamp of the archive file, which will cause the application server to redeploy the application, causing the re-weaving of the application with the Kieker aspects. Loss of session and breaking of running transactions can occur, but these are not within the scope of this paper. Also, if there is no *aop.xml* file inside the application archive, the AspectController can create a new one.

The communication through the MBeanServer may seem to cause increased performance lag and overhead. But since all these actions are performed within a single Java virtual machine [19], this overhead is lower than the overhead caused by, for example, storing records into the database.

On the receiving side, the Record Analyzer component analyzes the records contained in the database. Depending on its configuration, it chooses new monitoring parameters. These parameters are then sent to the DProfManager for a reconfiguration of monitoring. It is important to state that, additionally, users can manually change monitoring parameters using any JMX console application.

V. Evaluation

The use of the Kieker framework for monitoring of JEE applications will be demonstrated using the software configuration management (SCM) application described in [20]. SCM is a JEE application responsible for tracking applications and application versions. We deployed SCM on a JBoss 5.1.0 server. As described in Section IV, the Kieker framework has been extended with a new Monitoring Log Writer and additional functionality enabling reconfigurable monitoring scenarios.

A. SCM Application

The application is implemented using Enterprise JavaBean (EJB) [21] technology. Entity EJBs are used in the O/R mapping layer. They are accessed through stateless session EJBs (SLSB), modeled according to the façade design pattern [22]. SLSBs are annotated to work as JAX-WS [23] web services as well. The application client is a Java Swing [24] application which uses web services to access the application. Listing 1 shows an excerpt of the `OrganizationFacade` class. The `createOrganization(...)` method invokes the `checkOrgName(...)` method and creates a new entity EJB for the organization in the defined city. This method is annotated with `@OperationExecutionMonitoringProbe`. Other method definitions from this class are omitted from this listing, but are also annotated with `@OperationExecutionMonitoringProbe`. The `OrganizationFacadeService` remote interface is omitted since it contains only method declarations. Listing 2 shows an excerpt of the `Organization` entity EJB class. Other entity EJBs in this system are similar.

```
1  @Stateless
2  public class OrganizationFacade
3    implements OrganizationFacadeService {
4
5  ...
6
7    @OperationExecutionMonitoringProbe
8    public Organization createOrganization(String orgName,
9          String address, String eMail, long cityId) {
10     checkOrgName(orgName);
11     City c = em.find(City.class, cityId);
12     Organization org =
13       new Organization(orgName, address, eMail, c);
14     em.persist(org);
15     return org;
16   }
17 }
```

Listing 1. Instrumented OrganizationFacade SLSB class (excerpt)

```
1  @Entity
2  public class Organization {
3    long id;
4
5  ...
6
7    @Id
8    @OperationExecutionMonitoringProbe
9    public long getId() { return id; }
10 }
```

Listing 2. Instrumented Organization entity EJB class (excerpt)

The testing will be conducted by repeatedly invoking the `OrganizationFacade.createOrganization(...)` method. These invocations are supposed to generate data which will be used for diagram creation and program performance analysis. In a second scenario, using the JBoss JMX console, we will change weaving parameters so that executions of methods in entity EJBs are excluded from monitoring. This causes a redeployment of the application. The testing process is then repeated with the changed monitoring configuration.

B. Monitoring Configuration

In order to use the Kieker framework under JBoss AS, the Kieker libraries must be in the classpath, i.e., in the *lib/* directory of the server. Application monitoring is, in this case, conducted through the use of annotations and aspects already present in the framework. Listing 3 shows a part of the AspectJ configuration file (*aop.xml*) for this test case.

```
1  <aspectj>
2    <weaver><include within="gint.scm..*"/></weaver>
3    <aspects>
4      <aspect name="kieker.monitoring.probe.aspectJ.executions.
           OperationExecutionAspectAnnotationServlet"/>
5    </aspects>
6  </aspectj>
```

Listing 3. AspectJ configuration file

In the *aop.xml* we defined the monitoring aspects and the application packages to be monitored. We used the aspect that intercepts executions of methods annotated with `@OperationExecutionMonitoringProbe`. Upon application server startup, the selected aspects are woven into the monitored application, according to the *aop.xml*, and the application is deployed. Listing 4 shows the line added in order to exclude entity executions from monitoring in scenario 2.

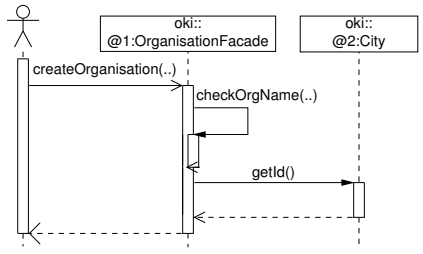
```
1  <exclude within="gint.scm.ws.entity..*" />
```

Listing 4. AspectJ directive for the changed monitoring configuration

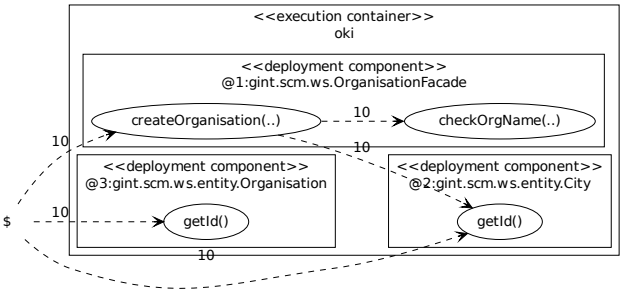
C. Data Analysis

Several diagrams are generated based on the data recorded and stored in the database during application execution. Diagrams, execution trace listings, and message trace listings which can be used in the analysis can be generated using the Kieker.TraceAnalysis tool.

Fig. 3 shows one of the generated sequence diagrams and an aggregated dependency diagram for the first test case, where entity EJBs are monitored. After we excluded monitoring of entity EJBs, we obtained a new set of monitoring data, which was used for the generation of the diagrams in Fig. 4.



(a) Sequence diagram



(b) calling dependency graph

Fig. 3. Diagrams reconstructed from the monitoring data of scenario 1

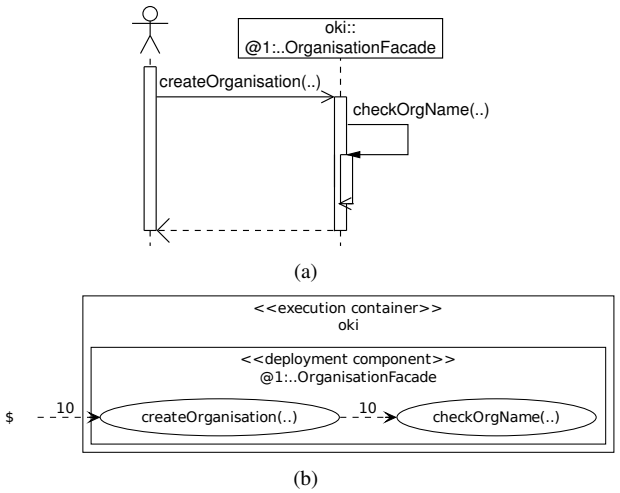


Fig. 4. Diagrams reconstructed from scenario 2

VI. Conclusion

This paper presented the use of the Kieker framework for continuous and adaptive monitoring of distributed Java EE applications. Based on a description of the Kieker framework, we presented our extension allowing to change monitoring parameters at runtime, particularly the instrumentation of software operations. For this purpose, we implemented a new Monitoring Log Writer as well as additional components. The use of JMX technology allows for remote monitoring and adaptation setups.

As a proof-of-concept, Kieker and the extension enabling adaptability, were used for monitoring a software management application (SCM) based on EJB and web service technologies. Proper configuration, running of the framework, and monitoring the application, deployed to a JBoss application server, were shown.

The first part of our future work will focus on designing and implementing a control component for adaptive monitoring. This component analyzes recorded runtime data and determines a new monitoring configuration. Moreover, the component will send these parameters to the DProfController component. Also, there is a possibility of framework extension by adding new aspects which allow for more complex measurements, e.g. memory, network traffic etc.

Acknowledgment

The research presented in this paper was supported by the Ministry of Science and Technological Development of the Republic of Serbia, grant III-44010, Title: Intelligent Systems for Software Product Development and Business Support based on Models.

References

[1] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the Kieker framework," Department of Computer Science, University of Kiel, Germany, Tech. Rep. TR-0921, Nov. 2009. [Online]. Available: http://www.informatik.uni-kiel.de/uploads/tx_publication/vanhoorn_tr0921.pdf

[2] Object Management Group, Inc., "UML 2.3 Superstructure Specification. OMG document formal/2010-05-05," http://www.omg.org/spec/UML/2.3/Superstructure/PDF/, May 2010.

[3] M. Flury, J. Lindfors, and the JBoss Group, *JMX: Managing J2EE with Java Management Extensions*. Sams, 2002.

[4] JBoss Community team, "JBoss Application Server," http://www.jboss.org/jbossas.

[5] The Eclipse Foundation, "The AspectJ Project," http://www.eclipse.org/aspectj/.

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

[7] JBoss Community team, "JBoss Profiler," www.jboss.org/jbossprofiler.

[8] Oracle, "Java Virtual Machine Tool Interface (JVMTI)," http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/.

[9] ——, "Java Virtual Machine Profiler Interface (JVMPI)," http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html.

[10] T. Parsons, A. Mos, and J. Murphy, "Non-intrusive end-to-end runtime path tracing for J2EE systems," *IEE Proceedings – Software*, vol. 153, no. 4, pp. 149–161, Aug. 2006.

[11] A. Villazón, W. Binder, D. Ansaloni, and P. Moret, "HotWave: Creating adaptive tools with dynamic aspect-oriented programming in Java," in *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE '09)*. ACM, 2009, pp. 95–98.

[12] 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*, vol. 32, no. 9, pp. 642–663, Sep. 2006.

[13] A. Sahai, V. Machiraju, J. Ouyang, and K. Wurster, "Message tracking in SOAP-based web services," in *Proceedings of the 2002 IEEE/IFIP Network Operations and Management Symposium (NOMS 2002)*, 2002, pp. 33–47.

[14] J. Ehlers and W. Hasselbring, "Self-adaptive software performance monitoring," in *Proceedings of the Software Engineering 2011*, ser. GI-Edition – Lecture Notes in Informatics (LNI). Bonner Köllen Verlag, Mar. 2011, to appear.

[15] dynaTrace software Inc., "dynaTrace – Continuous application performance management," http://www.dynatrace.com/.

[16] JInspired, "JXInsight," http://www.jinspired.com/products/jxinsight/.

[17] Nagios Enterprises, "Nagios," http://www.nagios.org/.

[18] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, ser. LNCS. Springer, 1997, vol. 1241, pp. 220–242.

[19] JBoss Community team, "The JBoss JMX Microkernel," http://docs.jboss.org/jbossas/jboss4guide/r2/html/ch2.chapter.html.

[20] D. Okanović and M. Vidaković, "One implementation of the system for application version tracking and automatic updating," in *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*. ACTA Press, 2008, pp. 62–67.

[21] Oracle, "Enterprise JavaBeans Technology," http://www.oracle.com/technetwork/java/javaee/ejb/index.html.

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

[23] M. Kalin, *Java Web Services: Up and Running*. O'Reilly Media, 2009.

[24] Oracle, "Swing (Java Foundation Classes)," http://download.oracle.com/javase/6/docs/technotes/guides/swing/.