Abstract

This thesis develops monitoring concepts for distributed systems that use remote procedure calls. The concepts are compared with regard to usability and feasibility. The implementation of one concept as part of the program ExplorViz is presented. The implementation extends ExplorViz to monitor RPC calls that are sent via HTTP connections and processed by Java Servlets. To evaluate the implementation, it has been tested with JPetStore, the Lobo Browser, and a small test program. The evaluation handles the full monitoring of the test system which creates a monitoring overhead of factor 3.6.

The thesis is part of the students bachelor/master project “Remote Procedure Call Monitoring”. In the project the program PubFlow, which is designed as a distributed system, has been monitored. PubFlow uses RPC calls based on connections of the technologies ActiveMQ, JAX-WS, and HSQLDB. The monitoring of these technologies has been implemented as part of the project.
Contents

1 Introduction ................................................................. 1
  1.1 Motivation .............................................................. 1
  1.2 Goals ........................................................................... 1
    1.2.1 G1: Analysis of the Servlet Interface ...................... 1
    1.2.2 G2: Development of a Monitoring Concept ............. 2
    1.2.3 G3: Implementation of the Servlet Monitoring ........ 2
    1.2.4 G4: Evaluation of a Software System with Full Monitoring 2
  1.3 Document Structure ..................................................... 2

2 Foundations and Technologies ........................................... 3
  2.1 Dynamic Analysis ....................................................... 3
  2.2 Distributed Systems .................................................... 3
  2.3 Remote Procedure Call ............................................... 4
  2.4 Java Servlet ............................................................. 5
  2.5 JPetStore ..................................................................... 5
  2.6 Java URLConnection .................................................... 5
  2.7 Lobo Browser ............................................................. 6
  2.8 AspectJ ....................................................................... 6
  2.9 Kieker ........................................................................ 7
  2.10 High-throughput Tuned Kieker Version ....................... 7
  2.11 ExplorViz ................................................................. 7

3 Approach ....................................................................... 9
  3.1 S1: Analysis of JPetStore .............................................. 9
  3.2 S2: Analysis of a Servlet Container ............................... 9
  3.3 S3: Development of Monitoring Concepts .................... 10
    3.3.1 Global Trace ID ..................................................... 11
    3.3.2 Local Trace ID ..................................................... 11
    3.3.3 Concept Evaluation ............................................. 14
    3.3.4 Bridge Record ..................................................... 15
    3.3.5 Servlets ............................................................. 16
  3.4 S4: Test System .......................................................... 16
    3.4.1 Server .................................................................. 18
    3.4.2 Client .................................................................. 18
    3.4.3 ExplorViz .......................................................... 18
## Contents

4 Evaluation 19
  4.1 Implementation .................................................. 19
    4.1.1 Infrastructure ................................................. 20
    4.1.2 Servlet Probe ............................................... 20
    4.1.3 HttpURLConnection Probe ..................................... 21
    4.1.4 Usage ......................................................... 21
  4.2 Quality Assurance .................................................. 23
    4.2.1 Tests ......................................................... 23
    4.2.2 Problems .................................................... 23
  4.3 Performance .......................................................... 24
    4.3.1 Response Time ................................................ 24
    4.3.2 Created Records ............................................. 27
    4.3.3 Page Load Time ............................................... 27

5 Related Work 29

6 Conclusion and Future Work 31
  6.1 Future Work ...................................................... 31
  6.2 Conclusion .......................................................... 32

Bibliography 33

A Data Disk 35
  A.1 Content ............................................................. 35
Chapter 1

Introduction

1.1 Motivation

Modern software systems become more and more difficult to understand, because of increasing complexity. Especially when the implementation is poorly documented or the documentation has not been transferred to current versions, the orientation in the software can be pretty hard for developers. The monitoring of the software system during its runtime can help to understand the relations of the program parts, and to regain some of the information that were lost by the lack of documentation. The programs developed for this purpose need to gather and prepare the information that were collected in the monitoring, and display them to the user.

Nowadays when cloud computing establishes, software systems gain another source of complexity which is the distribution of the system. The distribution makes it necessary that the parts communicate via network interfaces. To be able to include information about the network communication into the monitoring results, the communication has to be monitored as well.

A full monitoring of distributed software systems hopefully does not only give the ability to maintain the systems more easily, but also simplifies the development. Therefore the monitoring can help to do performance analysis and identify slow components that need to be improved. This becomes especially important for the processing of big data, where a good performing program can save a lot of time and money.

1.2 Goals

1.2.1 G1: Analysis of the Servlet Interface

The first goal is the analysis of the servlet interface with regard to monitoring. This should give us an idea how the monitoring of the servlet could be implemented, and what maybe has to be considered.
1. Introduction

1.2.2 G2: Development of a Monitoring Concept
The second goal is the development of a general monitoring concept that can be applied to every, or at least a lot of message technologies. The usability of the concept with as many message technologies as possible is important to make the monitoring easily extensible.

1.2.3 G3: Implementation of the Servlet Monitoring
The third goal is the implementation of the monitoring concept for Java Servlets. The servlets are used as a test system to show that the developed concept can be implemented. Therefore the implementation will be tested with JPetStore.

1.2.4 G4: Evaluation of a Software System with Full Monitoring
The fourth goal of this thesis is the evaluation of a full monitoring of a test system. The full monitoring includes a general monitoring of method calls and the remote monitoring. The test system is build of the JPetStore servlet and the Lobo browser and a small test application that create requests for the JPetStore. The evaluation should show, if a full monitoring is applicable on a productive system.

1.3 Document Structure
Chapter 2 introduces the programs and technologies that are used in this thesis. Chapter 3 presents four steps which show our approach to reach the first three goals that were presented in this chapter. Afterwards we discuss the implementation and the evaluation of the monitoring in Chapter 4. Related work that also deals with remote monitoring or monitoring of servlets is presented in Chapter 5. Finally a conclusion and some ideas for future work can be found in Chapter 6.
Chapter 2

Foundations and Technologies

This chapter gives an overview of foundations and technologies which are used in this thesis.

2.1 Dynamic Analysis

The main goal of this thesis is the improvement of dynamic program analysis. A dynamic analysis deals with the behavior of a program during its execution. The analysis collects data about concerns like memory usage, execution time, internal or external communication, method calls and other usage statistics of the program. Therefore records about events inside of the monitored application are created and put together to traces. These traces are stored or sent to an external program. Afterwards the traces can be visualized to show the behavior of the monitored program.

There are different possibilities how to process and prepare the traces to make them easily interpretable by humans. What kind of visualization should be chosen, depends on the goals of the monitoring. A possible goal of the monitoring is to identify components with bad performance, for example methods with a long execution time, or components with high memory consumption. Another possible goal of monitoring is to gain overview over the application. We use ExplorViz to display our monitoring results, as described later in this chapter.

2.2 Distributed Systems

A distributed system is a software system that consists of multiple running instances of the same or different programs that communicate over network interfaces. This provides the ability to distribute the instances over multiple physical devices. Distributed systems can be used to process higher amounts of data by assigning sub tasks to different computers which perform these sub tasks simultaneously. In the end the individual results have to be put together to get the result for the primary task. Depending on the parallelizability of the task, parallelizing can reduce the run-time significantly. The parallelizability depends especially on the complexity of separating tasks and putting the results together.

An example where the distribution of a system decreases the execution time of a task dramatically is web search. The web consisted already in 2011 of more than 63 billion...
2. Foundations and Technologies

![Diagram of RMI](image)

**Figure 2.1.** Overview of RMI (Taken from Oracle 2014)

pages, including multimedia sources and book pages. For a request at a search engine like Google, the search results have to be found in less than a second from these 63 billion sources. Therefore Google has created an infrastructure with a large number of networked computers. These computers store preprocessed information about all web pages in indexes. So a search request does not only require a large amount of computing power to search the indexes, but also a lot of fast accessible memory where the indexes are stored. Without this complex distributed system there would be no possibility to access the search results in such a short period of time. [Coulouris et al. 2011]

### 2.3 Remote Procedure Call

The concept of remote procedure calls is the ability to access functions that are located on a remote device. The most common implementations for remote procedure calls in Java is Remote Method Invocation or RMI. The RMI architecture consists of four elements. The first element is the RMI registry. The registry stores information about the objects that can be accessed, and who provides them. The provider of an object is a RMI server. The server can associate a name with a local object by calling the registry. Afterwards the object and its methods can be accessed from an RMI client. The client calls the registry to get the stored information. With these information the client can invoke methods on the objects of the server. The fourth element of RMI is optional. It is a web server that stores the classes of published objects. The client can load unknown classes from the web server to be able to operate on them.

To access the remote objects, the RMI client creates local stubs of referenced objects. These stubs can be accessed from a Java program as like local objects. But a method call is processed from the RMI implementation and sent to the RMI server. The server executes the method call on the local object and sends the return value to the client. The client
receives this value and returns it.

### 2.4 Java Servlet

A Java Servlet is an usual Java Program that is built with the purpose of responding to HTTP requests. To be not required to implement an own web server in every program, a servlet is built on a communication interface. Running a servlet requires an servlet container like Jetty or Apache Tomcat, which implements the communication interface. So a servlet is the equivalent to a CGI-Script on conventional web-servers and used for dynamic creation of websites with Java. The servlet container creates a new internal thread for every incoming connection and does not fork a new process, like a CGI call does. This makes servlets in general more efficient than CGI. [Chan and Mordani 2013]

An other approach to RPC can be done with servlets. The servlet publishes functions over URLs. These can be accessed via HTTP from a remote caller. The caller sends the functions arguments in the HTTP request. The function is executed on the server and the return value is sent in the HTTP response. This model is a lot more basic than RMI. It only provides the access to functions and not to objects, there is no mechanism to add functions at run time, there is no registry that provides information about the available functions and functions can not be accessed as if they were local.

### 2.5 JPetStore

JPetStore is an example for a complex Java Servlet. It uses the Stripes, Spring, and MyBatis framework. It was written in order to demonstrate the simple management of servlets using these frameworks. It implements a web store for an imaginary company that sells pets over the internet. We use this servlet as an example for our tests. Therefore, we have tried to do an analysis of the servlet, that is presented in Chapter 3.

### 2.6 Java URLConnection

A remote procedure call requires caller and callee. But the servlet interface is only designed for receiving requests, not for sending them. In our test system, we use the Java interface URLConnection from the java.net standard library to call the remote procedures. An URLConnection can easily be created by calling the method `java.net.URL.openConnection()` on an URL object of the URL that should be loaded. When using a HttpURLConnection, which is a subclass of URLConnection, the HTTP header can be set with `URLConnection.setRequestProperty(String key, String Value)`. A connection to the specified URL can be opened with the method `URLConnection.connect()`. After connecting it is not possible to change the header information, because they have already been put into the OutputStream. The OutputStream can be accessed with the method
2. Foundations and Technologies

![Image of Lobo browser compared to Mozilla Firefox.](image)

Figure 2.2. Comparison of the Lobo browser (left) to Mozilla Firefox (right).

URLConnection.getOutputStream() to send a request body. After closing the OutputStream, the response can be accessed by URLConnection.getHeaderFields for the request headers and URLConnection.getInputStream for the response body.

### 2.7 Lobo Browser

The Lobo browser is a graphical web browser implementation that is written solely in Java, which makes it possible to apply the ExplorViz monitoring. It has a user interface that is comparable to common browsers like Mozilla Firefox, but with less functionality. The browser can render HTML and CSS2, and has a JavaScript engine. The active development of Lobo has stopped in 2009, and the rendering speed cannot keep up with current browsers. This makes the browser not applicable in productive environments but it is sufficient for our test system. Figure 2.2 shows the Lobo browser compared to Mozilla Firefox. [community 2009]

### 2.8 AspectJ

AspectJ is an extension for Java that allows aspect-oriented programming. Aspect-orientated programming gives the ability to react on events inside of a program, without adjusting the program code at the methods where the event is triggered. Therefore AspectJ provides a concept that is called weaving. Weaving can be done with any kind of Java byte code. It means that an additional method call will be integrated in the byte code, and according to this, will be executed when the program is run. A new method call can be inserted at a joinpoint. A joinpoint is in short every method call in the program. Pointcuts are used to describe a set of joinpoints, for example every method with determined name, or every method that belongs to a determined class. The reaction to a pointcut can be defined in advices. Advices are like usual methods that can be applied before, after, and instead of
2.9 Kieker

Kieker is a framework for application performance monitoring and dynamic software analysis for software systems written in Java. Kieker consists of a monitoring and an analysis part. The monitoring part has to be included into the monitored software. It consists of probes, that collect information from the software, and passes the information to the analysis part. The analysis part consists of plugins that process and display the collected information. Therefore Kieker is easily extensible and can be used with custom components. [van Hoorn et al. 2012]

2.10 High-throughput Tuned Kieker Version

The high-throughput tuned Kieker version is a partial rewrite of Kieker that is designed to be able to process about 250 times more traces per second than the original Kieker. It uses AspectJ to patch trace creation methods into the software systems. These trace creation methods are called probes. The probes are integrated into the monitored software system during run time, therefore the monitored software needs not to be altered. Kieker creates thread-local trace identifiers inside of the systems to be able to connect the records of method calls to traces. The trace records are sent via network interface to worker units that collect, preprocess, and reduce the traces. The Kieker version has the ability to dynamically cluster these workers in order to provide support for different amounts of produced traces. [Fittkau et al. 2013]

2.11 ExplorViz

ExplorViz is a trace visualization software that is used to display traces created by the high-throughput tuned Kieker version. It is designed as a web server and prepares the traces to be rendered in browsers with WebGL. The traces can be displayed in a 2D landscape model that gives an overview over the distributed components of a software system. For the display of a software component it provides a 3D model based on a city metaphor that represents components, classes, and internal communication as districts, buildings, and connections between the buildings. An example of the 3D view is shown in Figure 2.3, the 2D view is shown later in Chapter 4.
2. Foundations and Technologies

Figure 2.3. 3D view in ExplorViz showing JPetStore during a request. The colored markers were added afterwards. Legend: **Black**: Tomcat servlet container; **Blue**: Spring framework; **Red**: iBatis framework; **Cyan**: Stripes framework; **Magenta**: actual JPetStore implementation.
Chapter 3

Approach

Our approach is separated in four steps.

3.1 S1: Analysis of JPetStore

As mentioned in Chapter 2 JPetStore is a servlet that is build using the Stripes, Spring, and MyBatis (formerly iBatis) framework. Without knowing the fundamental structure of these frameworks it is a lot of work to find out how the HTTP requests are processed. An overview of the used components is shown in Figure 2.3. The servlet interface is not used in the JPetStore code, but is hidden inside of the Stripes framework. The Stripes framework is used for the presentation of the page contents. Therefore JPetStore defines accessible pages in JSP files, that are interpreted by Stripes. The data access is done with the MyBatis framework, which requires the Spring framework for some advanced database transactions. The data itself is stored in a HyperSQL database that is only accessed from the MyBatis framework. The program logic that defines, which data will be accessed for a request is nearly the only thing that is implemented in the JPetStore source code.

JPetStore seems obviously to be designed with the model-view-controller pattern, and it extensively uses frameworks to only be required to implement the business logic. This is the only knowledge that we gained from the analysis. That did not help a lot concerning the possibility of monitoring the software. Therefore the next step is the analysis of a servlet container and the servlet interface.

3.2 S2: Analysis of a Servlet Container

There are a few different servlet containers, the most commonly used are Apache Tomcat and Jetty. Every servlet container uses the same interface to communicate with the servlet. The servlet interface is defined in `javax.servlet.Servlet`. Its method `service(ServletRequest, ServletResponse)` is directly accessed by the servlet container for every request that is made. That means a servlet has to implement this method, to be able to receive requests. The servlet container can spawn multiple instances of servlet to respond to multiple requests simultaneously. Therefore a servlet has a so called life-cycle with three stages. After creation of a servlet object, the servlet is still uninitialized. The servlet must not be used to create a response, in this stage. The initialization is done with the `init(ServletConfig)` method
3. Approach

which passes some start parameters to the instance. Afterwards the servlet is initialized
and the `service (...)` method can be called from the servlet container. One instance can
be used for multiple independent requests. Therefore in general, no information should
be saved locally in the instance, between the requests. When the instance is no longer
required, the `destroy()` method is called. This instructs the servlet to free its allocated
resources. Finally the instance can no longer be used to respond to requests and it must
not be initialized again. For a new instance, a new object has to be created and initialized.

3.3 S3: Development of Monitoring Concepts

The general monitoring concepts presented in this section have been developed in the
bachelor/master project 2013/14.

To be able to display the connection between remote programs, the monitoring has to
inform ExplorViz about messages that are sent between the programs. Neither ExplorViz
nor Kieker have already implemented methods to display remote connections. Therefore we
had to develop a concept that defines how to extract the required information and how to
communicate them to ExplorViz. The implementation of the handling of these information
in ExplorViz has not been part of the project but has been done by the ExplorViz developer.

As shown in Figure 3.1 there are a few different concepts for monitoring remote calls.
We separated these concepts in two categories. The first category deals with concepts
where one trace represents one execution. The execution may go through multiple nodes of
the remote system. Hence the category is called `global trace ID`. The second category deals
with concepts where one trace represents the execution inside of one thread that is why
this category is called `local trace ID`. The connections between the traces are represented by
a new kind of record.

![Monitoring Concepts Diagram](image-url)
3.3. S3: Development of Monitoring Concepts

3.3.1 Global Trace ID

The simplest approach that can be included into ExplorViz is to pass over the trace ID from the caller to the callee. This creates one long trace over all the engaged parts of the distributed system. In a synchronous remote call the caller passes its trace ID and order ID to the callee. The callee adopts this trace and proceeds the execution with the adopted trace ID. When the callee has finished its execution it passes the current order ID, paired with the result of the execution to the caller. The caller has to adopt the order ID that was received with the response. This ensures that every order ID has been assigned to exactly one record. Figure 3.2 shows the concept as sequence diagram.

This concept only works properly with synchronous remote calls. At asynchronous calls the caller and callee proceed with their execution and would both create records with the same order IDs. This problem could be solved by assigning the order ID globally. Therefore every part of the system that has the same trace ID needs to communicate with each other. But this creates another problem: the order of the order ID does not represent the call stack anymore, because the order IDs are mixed from both parts of the trace. We could find no solution that arranges the global trace ID with asynchronous remote calls.

3.3.2 Local Trace ID

Instead of adopting the trace ID of the caller, it is probably easier to sustain the concept that a trace ID represents a thread, as it does without considering remote calls. To be able to connect two traces with each other we need a new kind of record. Instead of influencing the diagram of the call tree of the programs, the new record should create a reference between two method calls. Initially we will call this record BridgeRecord. The BridgeRecord should contain following information:

---

Figure 3.2. Concept of a global trace ID.
3. Approach

- Trace ID of the caller
- Order ID of the caller
- Trace ID of the callee
- Order ID of the callee

The information are sufficient to see that a remote call occurred and which components where engaged. The Record does not have information if the message is a request or response. Every message is considered to be asynchronous to fit to messages of arbitrary technologies. For example ActiveMQ does only define asynchronous messages that can not be applied to a request or response. The record will be discussed further in this chapter. We have developed four concepts, how the information can be collected that are required to create the record.

Native Message ID

Native message ids are identifiers, that already existing in a message. These ids depend on the used connection interface of an protocol. By intercepting such an ID at the sender and the receiver, the information required for the record can be reconstructed by matching them.

Internet and Transport Layer  
A TCP connection defines a sequence number that is unique for a certain period of time, when paired with the remote IP and port number. The big pro of this concept is, that it is almost independent from the used message technology. It would only be necessary to identify the sequence number with the Java method that handles the TCP packet. That is the problem with this concept, the processing of TCP packets is done by the operating system and not accessible from Java. Furthermore can a TCP connection used by multiple threads, and messages might be larger than the maximum size of a TCP packet.

Message Protocol  
Some message protocols, like ActiveMQ already provide a unique message ID. But the problem is, that such an ID is not present in every message protocol.

Comparing the Messages

An other possibility is to connect traces by the content of sent messages. In order to do so, sender and receiver have to send a copy or hash value of the message to a bridge instance. This instance is responsible for creating remote records by matching the messages. As you can see in Figure 3.3 the bridge instance is an external program that matches the sent and received messages and creates the remote record.
3.3. S3: Development of Monitoring Concepts

![Diagram of S3 monitoring concepts](image)

Figure 3.3. Bridge instance that is responsible for creating the bridge records.

This concept also has the big advantage that it is almost independent from the used message technology. The only thing that is required to match the messages is a copy of the message and the trace and order IDs from sender and receiver. But there are also problems with this concept. There is no possibility to distinguish messages with equal content, for example in a program that sends acknowledgments to multiple addresses. The trace that created the message and the trace that received the message are not relatable without any doubt. This is important because the sent acknowledgments originate from different contexts and cause different reactions, which should be attributed to each other. Another problem is that the message may not be changed. If only a time-stamp, a check-sum, or the order of header attributes is changed, the bridge record can not be created. A change of these information may occur because they probably have no relevance, for the interpretation of the message. These changes could maybe detected, when sending not only a hash, but the whole message to the bridge instance. Sending the whole message is probably not a good idea because it would triple the data that is sent. This could easily overload the network when a lot of large messages are sent.

**Comparing Timestamps**

This concept is very primitive. It is similar to the previous one, but instead of comparing the whole messages, only the timestamps when messages are sent and received will be compared. This concept also requires a bridge instance as shown in Figure 3.3 and can easily be applied to every message technology. It could work pretty well, provided that a new message is sent not before the previous message is received. Another assumption that has to be made is, that the system time on every physical node has to be identical. So there is only one message at a time that is transferred through the system. If these two assumptions can not be made, it is impossible to identify which sent and received
3. Approach

messages belong to each other.

**Passing the Trace ID**

This concept is related to the concept adopting the trace ID from the global trace ID category. But instead of adopting the trace ID, the callee creates the remote record. Therefore the massage has to be modified at the caller and its trace ID and order ID have to be included in the message. The callee has to extract the ids from the message. So all the information required for the remote record are available at the callee. This makes the bridge instance that was required for most of the other concepts obsolete. Furthermore the sender of a message can be identified at the receiver without any doubt. This was not possible with a lot of the previous concepts. The problem with this concept is that additional information have to be included in every kind of message. This is no big deal with messages that have a message header where additional attributes can be stored. But some message technologies send their messages through a binary protocol that has to be altered, to include additional information. In this case this concept is harder to realize.

**Open a Parallel Connection**

Instead of including the monitoring information into the message, it is possible to create a parallel connection between the two monitored applications. The trace and order ID of the sender are passed through this parallel connection. Therefore the sender of the message has to open a socket, where he is waiting for a request of the receiver. When a message has been received, the receiver has to open the communication to the sender, and request the monitoring information. In this concept there is no problem that the message has to be altered. But an other problem has returned, this concept makes it very difficult to assign a sent message to a received message. When multiple messages are sent at the same time the sender may not resolve which incoming request belongs to which sent message.

3.3.3 Concept Evaluation

We came to the conclusion that the concept which includes the trace ID and order ID in the sent messages is the only one that is applicable on all message technologies and can assure that the monitoring information are correct.

All concepts in the category global trace ID are not usable with asynchronous messages, so they were excluded. Almost the same can be applied to the native message ids. These are not available in every message technology. Especially TCP is not required to be used, some messages can also be sent with UDP or in Java VM transport. Finally we could also exclude all concepts that can not assure that the created records hold correct information. These are comparing timestamps, comparing the messages, and open a parallel connection. Afterwards the only remaining concept that has not been excluded is passing the trace ID. Therefore we had to find a way, how to intercept the sent and received messages, and how to include the
monitoring information in the sent messages. Another advantage of local trace ids over global trace ids is that not every part of the system requires to be monitored.

3.3.4 Bridge Record

The bridge record should provide information about all the messages sent between the components of the distributed system. We said in the previous part, that the bridge record only has to be sent from the receiver of a message. In an environment where every part is monitored, this would be enough to identify every message. But if there is a participant that is not monitored, the information about the existence of the message could be lost. Therefore we replace the previous bridge record with three different kinds of records. An example with the records is shown in Figure 3.4 as sequence diagram.

**SentRecord**

A SentRecord holds the information, that a message has been sent. This includes the trace that has sent the message and at which execution point it has been sent. Additionally a timestamp of the sending time and the remote address of the receiver are stored in the record. Though the address of the receiver is optional because it may not be possible to extract this information from the message. The record is required in situations where the receiver of a message is not monitored. In this case it is the only record that is sent to ExplorViz. It conserves the information that a message has been sent. The address of the receiver is stored, to have an indication where the message was sent to.

Content of the record:

```java
long senderTraceId
int senderOrderId
int sentTimestamp
String destination
```

**ReceivedRecord**

The ReceivedRecord is created under the circumstance that sender and receiver of a message are monitored. The record includes information where the message has been sent and when and where is has been received. This record holds the most important information. It connects the sender with the receiver. It can only be created if both sides are monitored.

Content of the record:

```java
long senderTraceId
int senderOrderId
long receiverTraceId
int receiverOrderId
int receivedTimestamp
```
3. Approach

**UnknownReceivedRecord**

The UnknownReceivedRecord specifies that a message without monitoring information has been received. In this case it is not possible to match the sender with the receiver. But the record has optional fields to include information about the sender and the receiver as string. This information can be used to give an indication where the message came from.

Content of the record:
- long receiverTraceId
- int receiverOrderId
- int receiverTimestamp
- String sender
- String destination

### 3.3.5 Servlets

As described earlier in the chapter a servlet does extend the servlet interface and every request and response is passed as parameter to its `service(..)` method. The request and response can be accessed as an object at this point. This gives us the ability to read the monitoring information from the request and include monitoring information into the response. The remote record will also be created at this time.

Adding the trace information to the message is quite simple. Servlets send and receive HTTP Messages. These messages are separated into a header and a body. The header is extensible and can store additional properties. The next lines give an example of a HTTP message with included monitoring information. The HTTP body is separated from the header by a blank line.

HTTP Message:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 26
ExplorVizTraceId: 206623741042419191
ExplorVizOrderId: 53482

This is the body of an HTTP message.
```

### 3.4 S4: Test System

The test system consists of three parts, a server, a client and ExplorViz. The complete test system is included in the Appendix A.
Figure 3.4. Example of created records during an RPC call, with current trace and order ids.
3. Approach

3.4.1 Server
The server is JPetStore\(^1\) running in a Tomcat 7 servlet container. JPetStore can be accessed as a Git repository, and built with Ant. The built war-file can be deployed with Tomcat. The monitoring will include the whole Tomcat server, therefore the Tomcat has to be started with the start options described in Chapter 4.

3.4.2 Client
The client can be any Java program that connects to the server with an implementation of the abstract class \texttt{java.net.HttpURLConnection}. We have used two different programs in our test system.

The first one is a very small program that loads a HTML page. The main part of the program is an loop that finds all local links in the page, and selects a random one that is loaded in the next iteration. We have implemented this test program on our own.

The second program is the Lobo browser. Because the browser is written completely in Java and uses a subclass of URLConnection to communicate with the server it can be used without modifications. The precompiled version of the browser that can be downloaded from sourceforge\(^2\) does not support Java 7. So we had to compile the program from the source code\(^3\). Probably because the browser has not been updated for four years there are a few problems displaying the JPetStore pages with correct formatting. But apart from this the pages are loading properly and accessing pages by clicking on hyperlinks does also work.

3.4.3 ExplorViz
ExplorViz is also running as a servlet in a Tomcat 7 servlet container. This has to be a different instance from the Tomcat that runs the JPetStore. Because otherwise the monitoring probes could not connect to ExplorViz during start-up. Additionally the whole Tomcat is monitored when running on the same instance, so ExplorViz would also monitor itself and run slower.

\[^1\]https://github.com/mybatis/jpetstore-6
\[^2\]http://sourceforge.net/projects/xamj/
\[^3\]http://xamj.cvs.sourceforge.net/viewvc/xamj/XAMJ_Project/
Chapter 4

Evaluation

4.1 Implementation

We wrote a probe for each of both technologies that we wanted to monitor. The probe intercepts incoming and outgoing messages and sends appropriate records to ExplorViz. Additionally we created an infrastructure for the probes, based on the monitored-application project that is part of ExplorViz. Figure 4.1 shows an overview of our implementation.

![Figure 4.1. Class diagram of the monitoring implementation.](image)

Additionally we created an infrastructure for the probes, based on the project that is part of ExplorViz. Figure 4.1 shows an overview of our implementation.
4. Evaluation

4.1.1 Infrastructure

The infrastructure for our monitoring probes has been developed in the bachelor/master-project 2013/14.

We used the existing monitored-application project as a basis. This project provides a probe for general monitoring. The probe creates BeforeOperationEvent, AfterOperationEvent, and AfterFailedOperationEvent records for every method call. The project does also provide a MonitoringController that is responsible for sending the record buffers to ExplorViz. We created the following three classes that are required for our monitoring, additionally to the probes.

Serialization This class is used to create the remote records and to put them into the output buffer.

ProbeController We require the ProbeController to exclude events from the monitoring that are caused by the monitoring, or in other words: it prevents the monitoring from monitoring itself. This is a problem that did not occur in the monitored-application project because the general monitoring probe does not access methods from the monitored program. Our Probes have to do so, because they need to modify the messages inside of the monitored program.

ExtendedAspect This class extends the AbstractAspect from the monitored-application project that is responsible for the general monitoring. So we can check if the monitoring is enabled in the ProbeController. If the call should be monitored the method delegates the call to the AbstractAspect.

4.1.2 Servlet Probe

The servlet probe does only have one method that is called by AspectJ. The pointcut is set on execution(void Servlet.service(ServletRequest, ServletResponse)), the method that has been discussed in the previous chapter. This method is called for every request that is made from clients. The request and the associated response are passed to this method. The probe tries to read monitoring information from its header. If trace and order ids are found, a ReceivedRecord is created, otherwise an UnknownReceivedRecord is created. Afterwards the monitoring information are stored into the response header and the SentRecord is created. Finally the execution of the service method is resumed.

It is noteworthy to mention that the SentRecord is created at the same time as the ReceivedRecord. This is up to the implementation of the ServletResponse class. The response message is not sent when the service method returns, but can be sent from inside of the method. The message is sent through an OutputStream that is included in the

\footnote{https://build.se.informatik.uni-kiel.de/gitlab/explorviz/monitored-application}
4.1. Implementation

ServletResponse. The response may be sent at any time, even separated in multiple parts. The Point of time, when the message is sent depends on the underlying implementation of the servlet container. To be independent of the servlet container and the version of its implementation, the monitoring must exclusively depend on the servlet interface.

4.1.3 HttpURLConnection Probe

The corresponding probe for the HTTP Connection of the client has only one method that is called by AspectJ. This method should be called, when a request is sent. Therefore two pointcuts are required. The first pointcut is \texttt{call(void java.net.URLConnection+.connect())}. This is the method that is defined in the JavaDoc to open the actual connection to the remote object. The documentation also states that after the connect method has returned, the remote object has become available, and its headers and contents can be accessed. [Oracle 2013] The second pointcut is required, because the Lobo browser does not implement the connection as the documentation says. A subclass of URLConnection defines an own method that establishes the connection and makes the object available. Therefore the pointcut \texttt{call(* java.net.HttpURLConnection+.getResponseCode())} is required.

The method of the probe is called when one of the pointcuts can be applied. When the method is called, it tries to add the monitoring information to the header of the message. This can fail if the connection has already been established. If the connection has already been established, no message will be sent. This also means that no SentRecord has to be created. Otherwise a SentRecord will be sent to ExplorViz. After adding the monitoring information, the actual connect method will be called. The probe is still active and waits for the call to return. As said in the documentation, the response is available after the connect method returned. It only makes sense to create a Received Record, if a message has been sent, respectively a SentRecord has been sent in the previous part. If this condition is true, the method tries to extract trace and order id from the response header. Finally a ReceivedRecord respectively UnknownReceivedRecord is created and sent to ExplorViz.

There seems to be one problem depending on the implementation of the connection. Some implementations do not send a message for each URLConnection object, but receive the message from a cache. In this case a SentRecord is created where no matching ReceivedRecord will be created. Furthermore an second ReceivedRecord for the cached message will be created. Both of these records are ignored in the current implementation of ExplorViz, but it could be a problem in later versions.

4.1.4 Usage

The project uses the Apache Ant build tool. The build depends on the monitored-application project. Its location can be redefined in the \texttt{build.xml} file, if necessary.

When building with Ant, everything that is required for monitoring a program will be put in the \texttt{dist} directory of the project. Required are these four files:
4. Evaluation

![ExplorViz](image.png)

**Figure 4.2.** 2D view of ExplorViz showing the test system with multiple clients.

- explorviz-monitoring.jar: This is the Java program that executes the monitoring. It includes the AspectJ load time weaver that weaves the probes into the monitored program.

- META-INF/aop.xml: This is the configuration file of AspectJ. It makes it possible to delimit the monitoring on certain areas of the program and to deactivate a monitoring probe.

- META-INF/explorviz.live_trace_processing.default.properties: This is the configuration file from the monitored-application project. It defines a name for the monitored program that will be displayed in ExplorViz. You can also define the location of the ExplorViz server and disable all monitoring.

- servlet-api.jar: This has to be the version of the servlet api that is used by the servlet container. If a new version of the servlet API is released, this file has to be replaced by the new version.

All of these files are required for the monitoring, if one of them is missing the monitoring will probably fail. To monitor a program the explorviz-monitoring.jar file has to be defined as Java agent, for example like this:

```java
java -javaagent:/path/to/explorviz-monitoring.jar -jar program.jar
```

It is also possible to define a different ExplorViz configuration file with the additional option: `-Dexplorviz.live_trace_processing.configuration=/path/to/file`.
4.2 Quality Assurance

4.2.1 Tests

We did a lot of testing with multiple testing frameworks in the project. Creating the tests costed a lot of effort, but the results were sobering. There were only a very few findings and a lot of the testing frameworks were not compatible with each other. For example after mocking classes with PowerMock it was not possible to check the test coverage. That is because PowerMock creates new classes during run time, which is not visible to the test coverage checking framework.

The implementation of the servlet monitoring is a rather small project with only five classes that require testing. Therefore we decided to use only JUnit for testing this project. The created tests are similar to those in the monitored-application project. We use the concept of mocking to separate the tested classes from their environment.

Our tests for the probes are made of multiple test cases. In the first test case a message with monitoring information is received and a message is sent, where monitoring information have to be included. Another test case is that incomplete messages or null pointer are passed to the probe. In this cases it will be checked that no exceptions are thrown from the probe itself and that the order id is only incremented when a record is created. That no exceptions are thrown from the probe is important. Otherwise no messages could be sent if the monitoring had an error, and the monitored program would not be able to continue its execution. Another condition that the probes have to meet is, that the value of the monitoringEnabled variable from the ProbeController must be equal before and after the method has been called.

The serialization class has tests where the output buffer is checked for correct size and content. The ProbeController is checked for not returning null, and whether the enabling and disabling of the monitoring works. Finally the ExtendedAspect is checked for disabling the monitoring in the correct situations.

4.2.2 Problems

The big problem for the evaluation is, that the weaving cannot be tested with conventional tests. So we cannot verify if every sent and received message is intercepted by the monitoring.

The other problem is with the quality of our monitoring. The messages are intercepted on a very high level at general purpose interfaces and not inside of the implementation, where the message actually is sent. This has the big advantage that the implementation of the monitoring is theoretically compatible with any servlet container and implementation of HttpURLConnection, also in future versions as long as the interface is not changed. But therefore some information are lost. The servlet probe creates a received and a sent record directly after each other. Therefore there is no possibility to find out how many method calls are required to handle the request.
4. Evaluation

4.3 Performance

We use three tests for performance testing. The first test measures the response time of the monitored JPetStore. The second test measures the number of created records and traces per second that are produced in the servlet container and need to be processed by ExplorViz. In the third test, we measured the page load time of the monitored Lobo browser when sending requests to the monitored servlet container. The first two tests create CSV output files which are plotted afterwards with the program GNU Octave. The measurement in the third test was done manually. All tests were performed on a dual-core CPU with 4GiB RAM.

The request for the first two tests were made with the small test program that was introduced in Chapter 3. For the testing of JPetStore we defined two different settings. In the first setting 50 requests are sent to JPetStore. All requests in this setting are sent from one client. The next request is not sent until the response to the previous request was received. So the server has to execute one response at a time. This scenario will later be referenced as relaxed test. In the second setting, four clients create requests simultaneously. Each of the clients sends 50 request in a row, so the server has to create multiple responses at a time. This scenario will later be referenced as stress test.

The response time of JPetStore is measured directly in the test program that creates the requests. Thereby we collect this data for each request:

\[ \text{[requestId, request_time, response_time]} \]

The second kind of tests, measures the number of created records and traces per second. Therefore we created the `RecordCounterAspect`. This aspect is weaved into the trace registry and every method that creates records.

The data that is collected each second:

\[ \text{[timestamp, current_record_counter, started_traces_counter, completed_traces_counter]} \]

4.3.1 Response Time

The result of the relaxed response time test is shown in Figure 4.3. In this test we requested the same page for 50 times. The unmonitored JPetStore has an average response time of 49 ms. The monitored JPetStore has an average response time of 177 ms when the remote monitoring is disabled, and 187 ms with full monitoring enabled. The response time of the first request is significantly higher than the later response times. This is caused by the classloader, and when the monitoring is enabled also by the AspectJ weaver. When the first four requests are excluded from the calculation of the average, the unmonitored response time has an average of 15 ms, without remote monitoring the average response time is 57 ms, and with full monitoring the average response time is 55 ms. You can see that the average response time without remote monitoring is unexpectedly a bit higher than the average with complete monitoring. This is probably caused by inaccuracy of measurement on a computer, where a test cannot be performed twice under the exactly same conditions.
4.3. Performance

So the additional time required for remote monitoring is insignificant compared to the time used for general monitoring. The full monitoring has an overhead rate of 3.6. This is a bit more than the monitoring overhead in PubFlow, where we achieved an overhead rate of 3. The reason therefore is probably that PubFlow is a distributed system. So the unmonitored PubFlow already has a network overhead that does not multiply with the monitoring overhead.

We repeated this test, but did not request the same page for 50 times, but a random one at each request. The result is shown in Figure 4.4. The average response time without the first four requests is 12 ms without monitoring and 100 ms with full monitoring. In this test the classloader is also active in later requests, when a new function is used. Therefore the AspectJ weaver raises the average of the monitored requests and the monitoring overhead rate becomes 8.3.

The stress test with the same page requested by 4 clients simultaneously, is shown in Figure 4.5. The response time of the unmonitored test is similar to the relaxed test. But the
response times of the test with full monitoring has multiplied. The unmonitored requests have an average response time of 11 ms. This is 4 ms faster than in the relaxed test. This can be explained with the life cycle of a servlet. When there are a lot of requests in a short period of time, the instance of a servlet is reused for another request. But in the relaxed test with only one request per second the instances are only used once. In this case a new servlet instance has to be created for every request, which takes the additional time. The average response time of the monitored requests is 289 ms. The reason that the response time has multiplied is a CPU load of 100% on the test machine which is created by the monitoring of the four simultaneous requests. This means the monitoring and the handling of the monitoring records limits the number of handled requests to about 14 per second, on the test machine.
4.3. Performance

Figure 4.5. Response time test with multiple clients.

4.3.2 Created Records

Figure 4.6 shows the number of records and traces that are created in a test. The test consists of four clients which create as many requests as possible for 30 seconds. A maximum of 550000 records are created per second, separated in 55 traces. So every trace holds about 10000 records, which have to be processed by ExplorViz. There seems to be no problem with ExplorViz when processing this amount of records.

4.3.3 Page Load Time

We did not find any suitable measuring points in the Lobo browser, that indicated the loading has been completed. Therefore the exact measurement of the load time of the Lobo browser was not possible. We tried to measure the load time manually, but this was also a problem, because it was shorter than expected. The load time of one page with monitoring inclusive multiple requests at the monitored JPetStore to load all content and the rendering
4. Evaluation

![Graph of created records and traces per second.](image)

*Figure 4.6. Number of created records and traces per second.*

of the page, took much less then a second. The usage of the monitored browser does not feel much slower than the unmonitored one. Regarding the performance, there should be no problem to use the monitoring in a productive system.
Chapter 5

Related Work

It is difficult to find related work to the monitoring of remote procedure calls. There is one approach towards the monitoring of RMI to reverse engineer UML sequence diagrams from a running distributed system. In this work the message is not altered. The matching of the remote call at the caller and the callee is done with comparison of timestamps, as described in Chapter 3. Additionally the name of the called RMI method is compared, to gain a bit more safety in the matching process.[Briand et al. 2006]

This implementation could already cause trouble when two different clients call the remote method at about the same time. In large software systems, it is very probable that this situation appears.

A bit more can be found about the monitoring of servlets. One framework that provides performance monitoring for servlets is Kieker. The Kieker servlet probe can handle every incoming request and monitor its execution. It can also apply a handler chain to every request and it logs the CPU and memory usage during the execution of the request. [Project 2013]

This monitoring can only be used for performance analysis of the servlet. Neither the Servlet container is monitored, nor references to the sender of the request can be made.
6.1 Future Work

This thesis only deals with the development of a remote monitoring concept and a proof of concept like implementation. So there is still a lot of work to do.

There are at the moment implementations of this concept for the message technologies ActiveMQ, HSQLDB, JAX-WS, URLConnection, and Servlets. These are just a very few of all existing technologies. Especially RMI as a common used RPC technology is desirable to be implemented.

Additionally to the creation of the remote record, it seems to be interesting how to process and display their information. At the moment ExplorViz does only show ReceivedRecords with matching SentRecords on its 2D overview page. This could be enhanced by displaying addition information like the number of connections, and the destination string from the sent record which may differ from the application name.

At the moment ExplorViz does not show any information about UnknownReceived-Records, and SentRecords without matching ReceivedRecord. Therefore this also has to be implemented in the future. For example every sender or receiver that is not monitored, but receive messages from or send messages to the monitored system could be displayed as a black box. ExplorViz also lacks support for multicast messages. At the moment one SentRecord is matched to one ReceivedRecord. If a message is received from multiple listener, only the first receiver will be displayed.

The 3D representation in ExplorViz lacks support of remote records completely. Here are animations conceivable, where incoming and outgoing messages are represented by connections that leave the displayed area through a gate.

Another application where remote monitoring could be included is the standard Kieker version. Therefore two things has to be transferred to Kieker. At first the remote records must be implemented in the Kieker Analysis part. And second the monitoring probes have to be adapted to the Kieker API.

A lot of distributed systems probably does not consist of programs exclusively written in Java. Another possible future project is the translation of the whole monitored-application project including the remote monitoring, into other programming languages. The implementation of monitoring in other languages should not be a big problem, because nowadays there are aspect oriented extensions, in the style of AspectJ, for almost every
6. Conclusion and Future Work

language. There are also dynamic languages like Python where no extension is required for object orientation, but the language itself supports some kind of run time weaving.

6.2 Conclusion

The four goals that were defined in Chapter 1 have been met. We have analyzed the servlet interface and found a way how the sent and received messages could be intercepted. We also developed a few different monitoring concepts for remote calls. From these concepts one has been chosen as the best. This concept has been implemented for servlets and HttpURLConnections. Thereby we could set up a test system with full monitoring that could be evaluated.

The evaluation showed a monitoring overhead of factor 3.6, which seems to be relatively small for a full monitoring. It also showed that the work on a productive system with an overhead of factor 3.6 can be possible.
Bibliography


Appendix A

Data Disk

A.1 Content

Everything that was used in this thesis has been put on the disk. All programs, except from the Lobo Browser can be copied to a hard drive and run directly. For more information read the README file on the disk.

docs This directory contains the proposal, this thesis, and the presentation slides as pdf file.

jpetstore This directory contains a Tomcat 7 servlet container with JPetStore.

explorviz This directory contains a Tomcat 7 servlet container with ExplorViz.

monitored-servlet This directory contains the source code of the servlet monitoring implementation, including dependent projects.

lobo This directory contains the source code of the Lobo browser, including dependant projects.

httpconnector This directory contains the test program that was used to create the requests for the performance test.
**Eidesstattliche Erklärung**

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,