Monitoring Distributed Traces with Kieker

Master’s Thesis

Hannes Strubel

May 23, 2017

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf
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, 23. Mai 2017
Abstract

With the increasing complexity of software, understanding it becomes more important. To achieve a superior understanding of the structure and execution of software, there are monitoring tools which observe the software during the execution and process information about the software. Afterwards, the tools offer different analysis to examine different aspects of the execution.

Since the amount of software which distributed on multiple system grows, also the complexity of these systems increases. This leads to a new aspect for the monitoring tools. Instead of observing the execution on a single system, it is demanded to follow the execution across multiple systems.

This thesis discusses opportunities to monitoring distributed systems and to create execution traces of the distributed execution. Moreover, we discuss an approach to monitor distributed communication with the Kieker Monitoring framework. To test the approach, we provide an implementation for the Kieker Monitoring framework. A major challenge of the implementation is to realize a dynamical instrumentation of the communication. The instrumentation is based on aspect-oriented programming. In this connection, we discover general challenges related to the dynamical instrumentation.
Contents

1 Introduction ................................................................. 1
   1.1 Motivation .......................................................... 1
   1.2 Goals .................................................................. 2
      1.2.1 G1: Designing a Concept for Monitoring Distributed Traces .... 2
      1.2.2 G2: Implementing the dynamical Instrumentation and Monitoring .. 2
      1.2.3 G3: Implement the Reconstruction of Distributed Traces ............ 2
      1.2.4 G4: Evaluate the Concept ..................................... 2
   1.3 Document Structure .................................................. 3

2 Foundations and Technologies ............................................. 5
   2.1 Aspect-Oriented Programming ....................................... 5
   2.2 Communication Characteristics ..................................... 6
   2.3 Distributed Trace ..................................................... 6
   2.4 Approaches to Trace Distributed Systems .......................... 7
   2.5 Kieker Monitoring Framework ....................................... 8
      2.5.1 Kieker’s Monitoring Component .............................. 9
      2.5.2 Kieker’s Analysis Component ............................... 10
   2.6 Communication Techniques ......................................... 11
      2.6.1 Transmission Control Protocol (TCP) ......................... 12
      2.6.2 Representational State Transfer (REST) ..................... 13
      2.6.3 Simple Object Access Protocol (SOAP) ..................... 15
      2.6.4 Remote Procedure Call ...................................... 16
      2.6.5 Java Message Service ....................................... 16

3 Approach .................................................................. 17
   3.1 Monitoring .......................................................... 17
   3.2 Analysis .............................................................. 20
   3.3 Limitations ............................................................ 23
   3.4 Classification ........................................................ 23

4 Adaptations to the Kieker Monitoring Framework .................... 25
   4.1 Assumptions .......................................................... 25
   4.2 Record Structure ..................................................... 26
   4.3 Kieker Monitoring Component ...................................... 27
      4.3.1 Socket and ServerSocket ..................................... 28
   4.4 Kieker Analysis Component ........................................ 32
Chapter 1

Introduction

1.1 Motivation

Modern applications become more and more complex, while the understanding of their structure and behavior becomes more and more challenging. To gain knowledge about the structure and to understand the behavior of an application, monitoring tools are used. These tools observe an application during execution and usually provide different presentations of behavior and structure of the application during execution.

A common way to increase the performance of an application is to distribute the application more than one physical node. Further, with the increased use of clouds and the possibility of executing applications on a cloud infrastructure it becomes more important to distribute the application. At the same time, this increases the complexity of an application. This leads to a new aspect for monitoring frameworks. Instead of just presenting the execution of an application on a single node, we want to follow the execution through the whole distributed system. This requires techniques to follow the execution across multiple nodes.

In our work, we discuss an approach for the Kieker Monitoring framework to enable the monitoring of distributed systems. The Kieker Monitoring framework is an application-based monitoring tool which uses dynamic instrumentation. The framework monitors the application without modifying it. We could determine many so called tracing tools like Googles Dapper which allow tracing executions in distributed systems. In general, these tools are used with application which communicate with instrumented frameworks to gather the information. This information allows Dapper to follow the execution in the distributed system.

In contrast to Dapper, this work discusses the opportunity to instrument the application dynamically. Nevertheless, the general procedure to follow the execution in the distributed system are comparable. A common procedure is to pass an identifier in conjunction with the communication. In contrast to his, our work examines the opportunity to use existing identifiers and to abdicate passing additional information.
1. Introduction

1.2 Goals

1.2.1 G1: Designing a Concept for Monitoring Distributed Traces
The main intention of this work is to examine the possibility of a general concept to monitor distributed traces. This concept defines the amount and types of monitoring information. Further, it includes a procedure to instrument the application and gather the necessary information. Lastly, this concept includes the reconstruction of distributed traces. To reduce the overhead of the monitoring, the concept has to reduce the effort for the monitoring as much as possible.

1.2.2 G2: Implementing the dynamical Instrumentation and Monitoring
In order to test the monitoring, the implementation is the second goal. Since this approach is developed in context of the Kieker Monitoring framework, the implementation is separated into two parts. The first part affects Kieker’s monitoring component and is defined in this goal. This implementation includes the dynamical instrumentation of the application and the commitment to the required monitoring information. To reduce the overhead of the monitoring, the concept has to reduce the effort for the monitoring as far as possible.

1.2.3 G3: Implement the Reconstruction of Distributed Traces
This goal includes the second part of the implementation. The implementation enlarges the Kieker analysis component to enable a reconstruction of distributed traces. Further, the analysis component is able to visualize the distributed traces. In addition, a way to analyze monitoring information from multiple Kieker Monitoring instances has to be found.

1.2.4 G4: Evaluate the Concept
After the functionality has been implemented, the evaluation of the concept based on the implementation is the fourth goal. The evaluation is geared to the approach. Firstly, we evaluate the general probes. For this purpose, we use simple examples for several communication technology. Based on these examples we are able to make a statement about the feasibility and performance.

Afterwards, we use the gathered monitoring information to evaluate the trace reconstruction. Since our approach has to combine operations of multiple traces, we also need to evaluate the following procedure. Lastly, we evaluate the current presentation of distributed traces with Kieker. Given that we work with the reworked version of the Kieker analysis component which supports call trees, we restrict the evaluation to the presentability of distributed traces in call trees.
1.3 Document Structure

This thesis is structured as follows. In Chapter 2 we introduce the foundations and Technologies used in this work. Chapter 3 presents an approach to monitor distributed traces. Afterwards, design decisions related to our implementation are presented in Chapter 4. The evaluation of our approach and the related implementation are in Chapter 5. In Chapter 6 we present works which are related to the monitoring of distributed traces. Finally, Chapter 7 concludes our work by giving an overview of the results of our evaluation and by presenting potential future work.
In the following we present foundations and technologies which are related to this work.

2.1  Aspect-Oriented Programming

To understand the functionality of Aspect-oriented Programming, we first present the concept of crosscutting concerns and aspect-oriented programming [Laddad 2009]. The concept of crosscutting separates the functionality into pieces, to reduce the complexity of software. A single functionality is called concern. The concept differentiates between core concerns and crosscutting concerns. A core concern includes a core functionality of the system which can easily separated into a module.

The Crosscutting concerns are functionalities like logging, security or monitoring which are used in multiple modules. Thereof the crosscutting concerns receive their naming. The concerns cross across (crosscut) many modules.

To implement the functionality of the core concerns, the object oriented programming paradigm works well. Related to the crosscutting concerns, the object oriented programming paradigm forces the implementation to separate from the functionality in many modules. Hence, the aspect-oriented programming methodology is used to implement the crosscutting concerns. It modularizes the crosscutting concerns into so called advices. The advices are woven into the execution to compose the final system.

AspectJ [Miles 2004] is a Java implementation of aspect-oriented programming. It introduces the definitions of aspect, advice, joinpoint, and pointcut. An aspect is a class which includes the functionality of a crosscutting concern. It includes the implementation of pointcuts and advices. An advice is associated with the advice term from the crosscutting concerns and includes an implementation of the functionality. Each advice uses pointcuts to determine the places where they are woven in the final system. The position of an advice is defined by a joinpoint. A pointcut is a semantic expression of a joinpoint.

AspectJ offers the following types of advices which are introduced by the annotations given below:
2. Foundations and Technologies

- **@Around**: An advice with this annotation is woven around the joinpoint. Here the developer has to execute the joinpoint inside the advice by calling the proceed()-method of the joinpoint object.

- **@Before**: This kind of advice are executed before the joinpoint is called or executed.

- **@After**: After the joinpoint execution ends, this advices is executed.

- **@AfterReturning**: If an advice use the @AfterReturning, the advice is executed after the joinpoint processes successfully.

- **@AfterThrowing**: In the case that the execution of the joinpoint throws an exception, this type of advice is executed.

### 2.2 Communication Characteristics

This work differentiates between synchronous and asynchronous communications [Bengel 2013]. If we take a look at a simple communication in Java with TCP as shown in Section 2.6.1, we recognize that both nodes were able to perform operations between the sending and receiving of information. The information is sent to the receiver and stored on by the stream object until the receiver reads the message from the stream object. Moreover, it is possible that the node exchange information multiple times while the connection is open. The reading and writing of information on the streams does not affect the other node directly. This kind of communication is asynchronous.

In contrast, if we take at a simple REST communication as shown in Section 2.6.2, we observe a particular behavior. The client node performs a request to the server and waits for the response from the server. While the client node waits for the response, the node does not perform any operations. After the client receives a response, it continues its execution. There are certain mechanism like timeout to prevent a deadlock. The server waits for a request and performs the requested operations. Afterwards, it sends the response and waits for another connection until the service ends. This kind of communication is synchronous.

### 2.3 Distributed Trace

A trace represents the execution of a single thread. The trace contains a chronological order of the performed operations. Due to the restriction to one thread, a trace just includes the execution performed by one physical node. If the program execution is distributed on more than one physical node by using for example REST, SOAP or, RMI, every node generates a trace which shows the execution from its side.

A distributed trace represents the complete distributed execution. It shows the execution on
2.4. Approaches to Trace Distributed Systems

[Figure 2.1. General presentation of a distributed trace. The boxes mark the physical node which executes the operation.]

The client-side including the operations performed by the server-side. This works for every synchronous communication because we have no timeframe where both nodes execute simultaneously. If we just monitor a TCP communication, we have two threads, which are rather independent of each other and can execute at the same time. Figure 2.1 presents a distributed trace in form of a call-tree. If we just monitor the traces of an application, the client trace will only include all operations inside of the blue box.

2.4 Approaches to Trace Distributed Systems

[Sambasivan et al. 2014] introduces three approaches for tracing tools to trace an execution in distributed systems.

The first approach is the so-called metadata propagation. A common use of this approach is to implement the tracing component as part of the distributed system. The implementation modifies the implementation to propagate metadata. Hereby, it is possible to reenact the relationship between activities. Since they are commonly part of the distributed system, these implementations are often used on productive systems. Most of these implementations use sampling to reduce the runtime overhead. Popular implementations, which use this kind of information gathering, are Google’s Dapper and Twitter’s OpenZipkin. These implementations are used to work with large scale distributed systems.
2. Foundations and Technologies

The second approach is introduced as schema-based. This approach uses so-called temporal join-schemas which include the relationship among activities. These schemas are applied to the complete set of information after the execution of the application ends. To get the schema, the developer already needs knowledge of the structure of the application.

The third presented approach does not modify the traced application. Tools which use this approach infer relationships by correlating the existing information or by making simplifying assumptions.

2.5 Kieker Monitoring Framework

The Kieker Monitoring framework [Van Hoorn et al. 2012] is an application level monitoring and dynamic analysis framework. For more than ten years Kieker was developed by the Software Engineering Group at the Christian Albrecht University Kiel and the University of Stuttgart.

The Kieker framework is separated into Kieker’s monitoring component and Kieker’s analysis component (see Figure 2.2). The monitoring component is responsible for monitoring the application and collecting information about the application execution. Afterwards, the Kieker’s analysis component performs different analysis and creates visualizations of the results.
2.5. Kieker Monitoring Framework

Besides the main components of Kieker, there is a collection of associated tools to visualize results or to use Kieker with the Eclipse IDE. Although Kieker is developed in Java, it is possible to monitor and analyze code written in Visual Basic 6, COBOL and -NET based applications.

While using Kieker, the monitoring information is stored in so-called records. Every monitored event has its own record type. To simplify the definition of records, the Kieker project includes a Domain Specific Language (DSL) for the Records. The DSL is called Kieker Instrumentation Record Language (IRL) [Jung and Wulf 2016] and is available as Eclipse plug-in. To define a record for the Kieker Monitoring framework, it is necessary to define the classes and interfaces the record derives from as well as the additional attributes. Afterwards, the Kieker IRL creates the concrete implementation including all necessary methods and additional attributes. Since the Kieker Monitoring framework allows to trace the monitored application, most records include a trace identifier. This trace identifier is related to the thread which executes the operation.

2.5.1 Kieker’s Monitoring Component

The Kieker monitoring component has two major tasks. Firstly, the component uses so-called probes to collect monitoring information and to generate records for this information. The probes are responsible for the dynamical instrumentation of the monitored application. A common technology for probes is AspectJ. It is an implementation of the aspect-oriented programming paradigm. 2.1 describes the concept of aspect-oriented programming and AspectJ.

A probe contains different advices which are woven around the specified events. Inside of an advice, Kieker collects the monitoring information and generates the records with this information.

There are other technologies used to define a probe in Kieker, but all probes have in common that they generate records.

The second task of the monitoring component is the transferring of records to the Kieker’s analysis component. For this purpose, the monitoring component contains several writers. A writer is responsible for transferring the records to the Kieker analysis component. The Kieker project includes, for example, a writer to transfer the information with TCP or a writer to store the records on the file system. In general, the writer serializes the records before they are sent to reduce the amount of data. As a consequence, the analysis component has to deserialize the incoming information to receive the records. In the case that we use the file system writer, all record names are replaced by an identifier and an additional map-file contains the mapping between identifier and record name.

Kieker can be used in two ways. The first is to execute the analysis during the monitor-
2. Foundations and Technologies

In this case we use a writer like the TCP writer to send the information to the analysis component. Secondly, the analysis is executed afterwards. In this connection we use for example the file system writer.

2.5.2 Kieker’s Analysis Component

As we mentioned previously, the Kieker’s analysis component executes different analysis and creates visualizations of the results. One part of the analysis component which is related with this work is the trace reconstruction.

This work uses the reworked version of the analysis component\(^1\). Currently, the reworked version does not include all features of the current version, but it is possible to read records from the file system, reconstruct traces and create call-trees.

We decide to use the reworked version and refer the trace reconstruction to the new version. The analysis component architecture implement the pipe and filter pattern. The reworked version uses the pipe and filter framework Teetime\(^2\). Until the reworked version, Kieker uses an own pipe and filter implementation.

Figure 2.3 presents the structure of the reworked Kieker analysis component with the configuration we use.

The first stage is called ReadingComponent. This stage reads the Kieker records from an input directory and deserialized the records for the analysis. Afterwards, the records were sent to the next stage.

The second stage just ensures that every following stage receives Kieker records of a specified type. Currently this stage is configured with the IFlowRecord interface which is implemented by all compatible records.

The next three Assembler stages create models for the class or operation names. Then, the following stages can refer to the models instead of using full names.

Next to the Assembler stages, we use the TraceReconstructionStage. Until now, the stages pass record objects to the next stage.

The TraceReconstructionStage generates the traces and passes the complete traces as Trace objects to the next stage.

Afterwards, the TraceStatisticsDecoratorStage calculates different values for every monitored information. Currently, the stage determines the duration from each operation to its parent operation.

Then, the TraceToGraphTransformerStage generates a graph object of the trace and passes it to the DotTraceGraphFileWriterStage which finally converts the graph object to a dot file on the file system.

\(^1\)https://github.com/kieker-monitoring/kieker/tree/issue-1475-trace-analysis-tool-migration

\(^2\)http://teetime-framework.github.io/
2.6 Communication Techniques

There are different ways to implement the communication between nodes in systems. In the following, we present relevant techniques for communication, which are considered in this work. [Tanenbaum and Van Steen 2007]

\[\text{http://graphml.graphdrawing.org/}\]
2. Foundations and Technologies

2.6.1 Transmission Control Protocol (TCP)

The Transport Control Protocol (TCP) is dedicated to the Transport Layer of the Open Systems Interconnection model [Zimmermann 1980]. This layer is the fourth layer of the OSI model and also includes the User Datagram Protocol (UDP).

In contrast to the UDP, the TCP guarantees reliability. This means that TCP ensures that every information reach its destination.

There are two implementations in Java to realize a TCP communication. Firstly, we present the Socket and ServerSocket classes and then the SocketChannel and ServerSocketChannel class afterward.

Socket and ServerSocket Class

The java.net.Socket class allows opening a connection to a java.net.ServerSocket object, by calling the connect()-method. The Socket class identifies the server by its hostname and port number.

To allow a client to open a connection to a ServerSocket object, the server has to call the accept()-method. It is possible to open a connection to the ServerSocket object until the ServerSocket is closed. If a client opens a connection, the ServerSockets returns a Socket object on the server-side which is linked to the Socket object on the client-side.

After the connection is open, the Socket objects can exchange information. Therefore, the Socket objects contain InputStream and OutputStream objects. Theses stream objects offer methods for reading or writing information on the stream to the other node.

Finally, the connection gets closed by calling the close method of the Socket object. The connection closing on one side does not affect the objects on the other side.

Listing 2.1. A general implementation of a client with the Socket class.

```java
1  Socket clientSocket = new Socket();
2  clientSocket.connect(new InetSocketAddress(hostname, port));
3  DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
4  BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
5  outToServer.writeBytes(sentence + '\n');
6  modifiedSentence = inFromServer.readLine();
7  clientSocket.close();
```


2.6. Communication Techniques

A common procedure is the use of other classes to handle the reading and writing on the streams. Listing 2.1 shows a simple TCP client which uses the BufferedReader and text classes to handle the reading and writing. The written information remains on the streams until the connected application reads it from the corresponding stream. After the Socket is closed on the server-side, the ServerSocket executes the next incoming connection, until it is closed.

Listing 2.2. A simple implementation of a TCP server which uses the Socket and ServerSocket class.

```java
Socket clientSocket = new Socket();
clientSocket.connect(new InetSocketAddress(hostname, port));
DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());
BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
outToServer.writeBytes(sentence + '
');
modifiedSentence = inFromServer.readLine();
clientSocket.close();
```

?? represents a simple server which works with the client shown in Listing 2.1. If this server receives an incoming communication while it has an open connection, the incoming communication is stored and executed once the current connection is closed.

To allow multiple connections, we apportion the server implementation among different threads. The first thread executes the accept()-method and starts a new thread for each established communication.

SocketChannel and ServerSocketChannel Class

Besides the Socket and ServerSocket classes the Java library offers the SocketChannel and ServerSocketChannel classes which also allow a communication based on TCP. The opening and closing of a connection is similar to the Socket and ServerSocket classes. To exchange information, the SocketChannel class offers a method for reading and writing. It is not necessary to use secondary objects to send or receive information.

2.6.2 Representational State Transfer (REST)

Representational State Transfer (REST) is an architectural style for designing stateless communication between a client and a server [Mehta and Kalali 2013]. The clients send requests to get or modify a web resource.
2. Foundations and Technologies

A web resource is identified by an *Universal Resource Indicator* (URI). Every request is related to a resource. The types of request are derived from HTTP. It is possible to send GET, PUT, POST, DELETE or HEAD requests. The functionality of these requests is as followed:

- **GET**: returns a representation of the requested resource.
- **HEAD**: returns the HTTP Header of the requested resource. This request is often used to check whether a resource exists.
- **POST**: creates a resource on server-side. The resource is based on the representation that the client sends.
- **PUT**: updates a resource on server-side. This request is also used to create a reference to a resource on server-side.
- **DELETE**: deletes the requested resource on server-side.

On server-side, the request are proceeded and a response is returned. Since REST is stateless, every communication is stand-alone [Varanasi and Belida 2015]. The clients have to transfer all necessary information to the server so that it can understand the request. Moreover, the communication is independent from the implementation on client or server-side. Commonly, HTTP or HTTPS is used to transfer the messages in the application layer.

The REST support in Java is defined by the JAX-RS (The Java API for RESTful Web Services) specification\(^4\). In context of this work, we use the Jersey. This is a reference implementation of the JAX-RS specification [Burke 2013].

The Jersey implementation uses annotations to define the web service. In Listing 2.3 we use the @Path annotation to specify the relative path from this class. Moreover, we use the @GET annotation to specify the HTTP method for the method. The @Produces annotation defines the MIME media types this resource can produce.

**Listing 2.3.** This class defines a restful service with a method. The method can be requested by sending a HTTP GET request to this resource.

```java
@Path(HelloWorldService.webContextPath)
public class HelloWorldService {
    static final String webContextPath = "/helloworld";

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hallo() {
        // method implementation
    }
}
```

\(^4\)https://jax-rs-spec.java.net
2.6. Communication Techniques

2.6.3 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol (SOAP)\(^5\) is a protocol which based on XML mess-
ages [Balani and Hathi 2009] [Vohra 2012].
The protocol distinguish between a service provider and service consumer. A service
consumer sends a request-message to the service provider and receives a response-message
from the service provider. Each message is an XML document.
A service provider offers a description of the SOAP message style using the Web Service
Description Language (WSDL).
The XML messages are in accordance with a WSDL descriptions.

A benefit of a SOAP communication is that it is independent from the service consumer
or service provider implementation.
Moreover, SOAP is a stateless communication protocol.

There are several SOAP implementation for Java. In the context of this work we use the
JAX-WS implementation. This implementation is part of the Java standard libraries.

To implement a SOAP web service the implementation offers the @WebService annotation
(see Listing 2.4). This specifies a class as web service. By adding the @WebMethod annotation to
a method, we enable the access to this method by SOAP request. The SOAP implementation
handles the providing of the WSDL and the encoding of requests and responses.

Listing 2.4. This class defines a simple web service, which just returns "Hello world!"

```
@WebService(name = "ExampleWebServices")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public class ExampleWebService {

    @WebMethod
    public String hello() {
        return "Hello world!";
    }
}
```

---

\(^5\)https://www.w3.org/TR/soap
2. Foundations and Technologies

2.6.4 Remote Procedure Call

The concept of Remote Procedure Calls (RPC) allows calling functions on other systems. Here the involved systems are divided in client and server. The server offers a method which can be called by the client. A major design goal is to abstract from the communication and enable the developer to use RPC like a normal method. The communication with RPC based on XML messages.

Remote Method Invocation (RMI) is the object-orient equivalent of RPC in Java. It allows calling functions from other Java Virtual Machines. The communication with RMI uses components. On the client side we have the Remote Interface which includes a description of the remote procedure’s behavior. The server implements the behavior in a Remote Object and offers one or more instances of this object. To call an instance of this object, the client needs a Remote Reference which refers to the Remote Object.

The client receives the Remote Reference from a central RMI Registry. Therefore, the client has to transmit the Remote Interface to the registry. If the registry knows a matching Remote Object it passes the Remote Reference to the client. If a server offers a Remote Objects, it registers the Remote Objects at the RMI Registry.

2.6.5 Java Message Service

Apache ActiveMQ\(^6\) is a Message Oriented Middleware (MOM) from the Apache Software Foundation. They solicit ActiveMQ with high availability, performance, scalability, reliability and security for enterprise messaging.

It implements the Java Message Service (JMS) and is licensed using the Apache License. The goal is to provide standard-based, message-oriented application integration for a lot of languages and platforms [Snyder et al. 2011]. A main component in the communication with ActiveMQ is the message broker. It can be used in two scenarios. In the first, we have a producer and a consumer. The producer sends messages to the consumer by using the message broker. The message broker queues all messages for the consumer and removes a message once the consumer received the message. In the second scenario we have a publisher and a various number of subscribers. The publisher sends a message to the message provider which distributes the message to all subscribers. Since the messages are stored, this communication is asynchronous.

\(^6\)http://activemq.apache.org
Chapter 3

Approach

As defined in Section 1.2, we try to achieve a common approach that covers as many communication technologies as possible. The idea is to monitor the incoming and outgoing communication on the monitored nodes and to reconstruct the distributed traces in the analysis component.

We separate the approach into a monitoring task and an analysis task. In Section 3.1, the tasks of the Kieker’s monitoring component are presented. This section includes the significant events and the monitoring information. The following analysis of this monitoring information with the analysis component is presented in Section 3.2.

3.1 Monitoring

Taking a general approach, we are monitoring events which are independent of a specific communication technology. Hence, we choose simple TCP communication. This protocol is part of the transport layer in the Open Systems Interconnection model (OSI model)\(^1\) and the lowermost layer of the host layers. We expect to cover all TCP-based communication technologies by instrumenting the communication with TCP in Java.

Therefore, the first step is to define the relevant events of a TCP communication. In Section 2.6.1 we introduce the structure of a TCP communication in Java. We define four relevant events. Figure 3.1 shows a schematic view of a TCP communication and the relevant events.

Firstly, we monitor the beginning of the communication which includes the establishing of the connection. The monitoring data of this event contains the role of the node in this communication.

As soon as the connection is established, the nodes can send and receive information. The exchange of information is the second and third event we want to monitor.

After the information exchange ends, both nodes close the connection on their side. The connection closing is the last event we monitor. The monitored data of the closing events also includes the role of the node.

\(^1\)http://www.itu.int/ITU-T/Recommendations/rec.aspx?rec=2820
3. Approach

![Figure 3.1. Schematic view of TCP communication between a client and server. The annotations (red) describe the general communication events.](image)

Since we work with Kieker, we use one Kieker record type for each event. After we test the feasibility of this approach, we can reduce the number of different events. It should be enough to generate monitoring data only for the exchange of information. In case that we also want to monitor a general TCP communication, we should keep the current procedure and create additional records for distributed traces. We assume that a smaller amount of monitoring data reduces the overhead of the monitoring.

After monitoring an application, the following analysis has to reconstruct the communication of the nodes. The first step is to reconstruct the single traces including the monitored TCP events. Afterwards, we connect the monitored TCP events from a client trace with the TCP events from the server. A benefit of a TCP communication is that TCP guarantees chronological order for the exchanged information. This means that the first information sent by the client is also the first information which the server will receive. Hence, we can connect the sending of information on one node and the receiving of the information on the associated node without any additional effort. The major challenge is to connect the monitored information from a client node with the monitoring information from the associated server node.
We discussed several ideas to find a proper identifier. Similar considerations were done in the Kieker related work which we are presenting in Section 6.2.

A simple consideration is to use the timestamps of the TCP events to determine the associated events. This consideration has three major weaknesses. Firstly, this possibility presupposes that the physical nodes have the same system time. Unfortunately, we cannot assume this for every distributed system. Further, we need knowledge about the time a message needs to reach the connected node to guarantee a correct assignment while working with multiple messages at the same time. Moreover, the duration will probably deviate each time. Lastly, it seems to be impossible to guarantee a correct assignment if an application can process multiple connections at once.

Another consideration is to compare the actual message content to create an identifier on both sides. For this purpose, we need to generate a hash value of the message and use the hash value as an identifier for the message. The major disadvantage of this concept is the effort it takes. Further, we need to consider if the message could get modified before we create the hash value on the server-side.

Lastly, we thought about the use of a native message identifier. This has many advantages, but not every communication technology uses a message identifier and further not every technology allows to access them.

Since we want to support many communication technologies, we decide to use the IP address and the port number from the current node and the associated node. These addresses together allow us to identify a single communication. If we just take the IP addresses, it is not possible to reconstruct correct distributed traces. We cannot distinguish between multiple connections between the same nodes by using the port numbers.

One disadvantage of the port number is that the number is reused after a time. The specific time depends on the opened connections in this timeframe. We neglect this aspect for two reasons. Firstly, our identifier is not unique till we have the same port numbers on both sides. Secondly, our analysis bases on the pipe and filter architecture pattern which can work with repeating identifiers as long as they do not appear simultaneously. Since there is a large amount of possible port numbers, the interval between repeating port numbers is long enough.

The combination of IP address and port number has three properties which support our approach. At first, the IP address on its own is unique in the distributed system. Each IP address
3. Approach

identifies a single node in a distributed system. Secondly, the client and server nodes have both access to the same addresses and port number during the communication. Furthermore, the IP addresses and port number are independent of a particular communication technology. We do not need to find or generate an identifier with these characteristics. In case that we would generate an identifier, we need to pass the identifier between the nodes.

3.2 Analysis

Based on the previously defined events, we can reconstruct the trace in the analysis. We have to pay attention to the fact that TCP communication is asynchronous. The client and server can execute operations at the same time, and the transferred information is stored in the stream until the receiving node uses it.

If we have a synchronous communication like REST or SOAP, the communication fits to the following procedure. The client begins the synchronous communication by sending a request to the server. Then, the server completes the request and returns a response to the client. In this case, the TCP communication on client-side persists of the establishing of the connection, sending a request followed by reading the response. At last, the connection is closed. The client pauses after sending the request and before reading the response. The corresponding server-side persists of accepting the connection, reading the request, performing the requested operations, returning the response, and closing the connection. This results in two relevant events on each side. One event to mark the beginning of a synchronous communication and another event for the end of the communication.

Our approach stipulates that we receive the synchronous communication events by aggregating the TCP events based on the following schema. The TCP event for establishing a communication and sending information constitutes the communication beginning event for synchronous communication on the client-side. On server-side, the establishing and receiving of information create the synchronous communication beginning event. The events for the end of the communication are equivalent to the communication beginning events. Figure 3.2 illustrates the aggregation of the TCP events to receive the beginning and the ending of the synchronous communication.

Subsequently, we are able to create the distributed trace based on the beginning and
3.2. Analysis

![Figure 3.2](image)

**Figure 3.2.** Schematic representation of the aggregation and the merged distributed trace. On the left side, the communication events are combined. The red boxes mark the beginning and the blue boxes the end of the synchronous communication.

The distributed trace for the client follows the execution on the client-side, follows the request to the server-side, and returns to the client-side once the server sends the response. Sending the request resembles the beginning of the distributed part, and receiving the response marks the end of the distributed part on client-side. The client does not execute any operations between sending and receiving. On server-side, the distributed part begins by receiving a request and ends by returning of a response.
3. Approach

Figure 3.3. Schematic representation of the merging. On the left side, there are a client and server trace with the aggregated synchronous communication events. On the right side, we have the distributed client trace which summarizes the operations from the client and server-side.

To generate the distributed trace we have to insert a part of the server trace into the client trace. Figure 3.3 presents the merged client trace based on the aggregated communication beginning and ending events.

The aspired procedure for Kieker’s analysis component is as follows. Firstly, we connect the TCP events for the communication beginning on client-side with the communication beginning on the server-side. Afterwards, we connect the sending of information from one with the receiving of information from the connected node. After we reconstruct the TCP communication, we apply our schema and merge the server-side events into the client trace to generate a trace of the distributed execution.
3.3 Limitations

Although we tried to develop a general approach, it has some limitations. The approach works under the presumption that we can monitor the TCP communication. The worst case is that we cannot monitor the TCP communication while using a particular communication technology. At this moment, we have to monitor additional events of the particular communication technology. This results in two consequences. Firstly, we have to receive the IP address. This can be challenging because the communication technologies often abstract from the TCP communication. Secondly, the types for the monitored TCP events do not fit properly. As a consequence, we cannot use our approach correctly.

Moreover, we monitor the TCP layer in this approach, to cover all TCP based communication technologies. TCP protocol is part of the transport layer. The transport layer also includes the User Datagram Protocol (UDP). Unfortunately, our approach does not work with UDP. The reason for this is that UDP does not guarantee reliability. While using TCP, the protocol ensures that the connected node receives the information. If we send information using UDP, the protocol does not guarantee that we receive the information. In the worst case, we monitor multiple information sending events, but only one information receiving event from the connected node.

We also have to keep in mind that we combine events from different physical nodes. These nodes might use different system times. Therefore, the operations in a distributed traces may not be chronology ordered. To solve this, we could synchronize the timestamps of the server-side operations based on the beginning of the communication.

Further, we assumed that all nodes of the distributed system is monitored, and the analysis receives all monitoring information while developing this approach. As a consequence, we cannot ensure that the analysis works only with the monitoring information of one node. The client trace is incomplete until as long as the server is unknown. If we do not monitor the server, the analysis will not return the client trace. To return the incomplete distributed traces, we need to implement additional mechanism. Since this work is related to Kieker, a simple solution is to flush the storage for the incomplete client traces before the analysis terminates.

3.4 Classification

In Section 2.4, we present three approaches for tracing distributed systems. Our approach is mostly related to the schema based approach. We use a general schema to determine the synchronous communication. The basis for this schema is the monitoring of TCP events.
3. Approach

To get correct results the user needs to configure the analysis. The analysis is not able to decide on its own in which situations it is necessary to apply the schema.
Adaptations to the Kieker Monitoring Framework

In order to test the feasibility of our approach, we extend the Kieker Monitoring Framework. In the following section, we focus on the major design decision we made to realize the approach.

At first, we keep hold of the assumptions we made.

Afterwards, we present the necessary Kieker records followed by the significant change on the Kieker monitoring and the Kieker analysis component.

4.1 Assumptions

The first assumption we made is that the trace identifiers are unique in the distributed system. The creation of the trace identifier leaves a low chance of equal trace identifiers that we neglect.

This work is not the first which neglects the low risk of equal trace identifiers. The implementation presented in 6.1 also uses the trace identifiers on multiple nodes without additional protection.

In case that this becomes a problem, it is a solution to modify the record import in Kieker’s analysis component. Before performing any analysis on the records, the trace identifier can be extended by a newly generated node identifier or by the hostname.

Another assumption we made affects the generating of hash values. While developing the implementation, we need to generate hash values to identify an object in different advices. For this purpose, we are the System.identityHashCode(Object) method. It is possible that this method generates equal hash values, but the likelihood is vanishingly small (Goetz and Peierls [2006]). That is why we assume the method can generate unique hash values. Nevertheless, the method is already part of the Kieker Monitoring component. It is used to create object and class identifiers, which are stored in records.
4. Adaptations to the Kieker Monitoring Framework

![Class diagram of the records for the start and end of a communication. We do not represent properties and methods.](image)

4.2 Record Structure

Since we use the new upcoming version of the Kieker’s analysis component, we have to take care on a specification related to the record structure. The new version determines that every monitored event persists of a before and after-record. Because of this, we need to define two records for each event. We define the records in the *Kieker Instrumentation Record Language (IRL)* [Jung and Wulf 2016]. Based on the definitions the *Kieker IRL* generates the record implementations. Section 2.5 contains a more detailed description of the *Kieker IRL*.

As mentioned previously, a TCP communication has four relevant events on client and server-side. The first records are the *BeforeClientCommunicationStartRecord* and the *AfterClientCommunicationStartRecord* for the communication beginning event on client-side. Moreover, we define the *BeforeClientCommunicationEndRecord* and *AfterClientCommunicationEndRecord* for the communication end. All records contain the local IP address and the remote IP address.

Similarly, we define the records for the server-side. We use different record types to allow a simple differentiation in the analysis.

Figure 4.1 shows the class diagram for the generated classes.

We reduce the number of different record types by using the same records for the information sending and information receiving events on client and server-side. These are named *BeforeIncomingCommunicationRecord*, *AfterIncomingCommunicationRecord*, *BeforeOutgoing-
4.3 Kieker Monitoring Component

To receive the necessary information we add probes to the Kieker monitoring component. A probe collects information from the monitored application and generates a record. The primary technology for probes in Kieker is AspectJ. As mentioned in Section 2.1, AspectJ works with pointcuts and advices. We use before, after-

---

**Figure 4.2.** Class diagram of the communication start and end records for a client and server

CommunicationRecord, and AfterOutgoingCommunicationRecord. The records include the IP address of the sender and receiver node. In combination with the communication beginning and communication end events, we can classify the role of the node. Figure 4.2 shows the class diagram of the created classes.

We add the IBeforeCommunicationRecord to all before-records and the IAfterCommunicationRecord to all after-records. Through this, it is simple to differentiate between common records and communication records. A discrimination is necessary while reconstruction the traces, because the communication records influence the trace structure.

Finally, we derive the AfterCommunicationFailureRecord from the general AfterOperationFailedEvent. This record signals that a communication operation had thrown an exception. It replaces the after-record from the failed operation. Currently, this record does not extend the information of the AfterOperationFailedEvent. Section 4.4 describes the handling of the AfterCommunicationFailureRecords.
4. Adaptations to the Kieker Monitoring Framework

Returning and afterThrowing-advises in our implementation. All pointcuts point to the method calls. This means that the pointcut finds all calls from the referred method. The advices will be executed before respectively after the method is called.

If a monitored operation throws an exception, the afterThrowing-advice applies. The afterThrowing-advice is similar for all probes. Essentially, the advice creates AfterCommunicationFailureRecord with the failure cause. As we described in Section 4.4, the handling of failure records is currently quite simple in the analysis. It is possible to add more information to the AfterCommunicationFailureRecord to increase its significance.

The afterThrowing-advice also gets triggered in case that the before or afterReturning-advises fail. That is why we have to ensure that the execution of our advices does not fail. In this context, there are two critical scenarios. The first one is that the before-advice throws an exception before the advice sends the record. The other scenario is that the afterReturning-advice fails after sending the record. In both cases, the analysis will receive an additional AfterCommunicationFailureRecord which prohibits an accurate reconstruction.

Unfortunately, we discovered a limitation of AspectJ, while working on the implementation and evaluation of this concept. AspectJ cannot access all Java standard libraries. As a consequence, we implement more probes than planned to make an evaluation of the analysis possible.

In the following, we first consider the probes for TCP Socket and ServerSocket classes and then briefly look at the SocketChannel and ServerSocketChannel classes. We do not introduce the additional probes because they do not fit in our approach for the monitoring and we just use them to evaluate the analysis part of our concept.

To allow a proper configuration of the monitoring, we separate the probes into multiple classes. Hence, we can select particular probes while monitoring to reduce the additional effort. Moreover, it is common practice to create an abstract class of the probes and derive a concrete class. The abstract class includes the advices and abstract pointcuts. In the concrete class, the pointcuts are defined. This allows defining custom pointcuts for the advices in the configuration file.

4.3.1 Socket and ServerSocket

The beginning of the communication is different on client and server-side. On the client-side, the Socket.connect()-method establishes the connection. We define the pointcut to
point to this method call. The advices can access the Socket object to receive the IP addresses and port numbers. On the server-side, the pointcut refers to the ServerSocket.accept() method. This method returns the Socket object which contains the IP addresses and port number.

We have to keep in mind that the connection is established after the connect and accept-method are executed successfully. Hence, the before-advice cannot receive the actual IP addresses and port numbers. While reconstructing this event in the analysis, we have to account for this fact.

The addresses are stored as an InetAddress-object\(^1\) in the Socket class. The InetAddress class contains the IP address, the port number, and also the hostname. We use the toString() method of the InetAddress to receive the local and remote address. At this moment, we avoid the concatenating of strings in the monitoring component. A downside of this procedure is that the returned string includes the hostname. In general, an address looks like "hostname/ipaddress:port" and if the hostname is unknown the address looks like "/ipaddress:port". As a result, we have to adjust the addresses before we start the analysis to match the common representation. The adjusting is done in the analysis component while reconstruction the trace. After the adjusting every address looks like "ipaddress:port".

Although we define the client and server address together as the identifier for the communication, we store the address in two variables. First, it makes the adjusting of the addresses easier. Secondly, it allows further differentiation in the analysis. Moreover, we do not have to concatenate the strings which would create an additional string instance.

After defining the pointcut for the communication beginning event, we focus on the communication end event. As previously mentioned, the communication on client and server side ends by calling the Socket.close() method. The method-close() is defined by the Closable interface\(^2\). As a consequence, we have to define our pointcut for Closable.close() to create clean code.

Inside the advices, we have to test if the matched operation is relevant for our monitoring. To select the relevant objects, we use the instanceof-method.

We cannot differentiate between client and server with different pointcuts. The differentiation has to be made inside of the advices. The Socket object does not note if it was created by a ServerSocket object. That is why we need to store additional information on

---

1. https://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html
2. https://docs.oracle.com/javase/8/docs/api/java/io/Closable.html
4. Adaptations to the Kieker Monitoring Framework

the server-side.

Firstly, we create a hash of the returned socket instance of the accept()-method and store this hash value. This is part of the after-advice.

Afterwards, we have to generate the hash value of the Socket object in every Socket.close() advice and to check if the hash value is stored.

There are a few opportunities where the hash value can be stored. Firstly, it is possible to store the hash value in a ThreadLocal object. As a result, the thread which executes the accept-method is the only one who can access the stored hash value. This solution works as long as the server handles only one open connection at once. In 2.6.1 we mentioned that the accept and close-method were executed in different threads if the server allows multiple connections.

In this case the Socket.close() advices will never find a stored hash value and the advices will always create a BeforeClientConnectionEndRecord and AfterClientConnectionEndRecord. Instead, we use a list inside the aspect class.

Another point to mention is that the connection is closed once the after-advice gets executed. That is why the after-records do not contain the correct addresses. The analysis needs to account for this fact.

Beside the advices for the communication beginning and communication end, we need advices for the actual communication. While working with TCP sockets, the sending and receiving is realized with java.io.InputStream and java.io.OutputStream objects. This leads to an issue which we discuss in the following by using the InputStream class. The InputStream and OutputStream classes are handled equally.

As described in Section 2.6.1, the method-write from the InputStream class sends information to the related OutputStream. If we just monitor every call of the write-method, we cannot guarantee that these calls are related to a TCP communication. That is why we are weaving our advices after the Socket.getInputStream()-method call. Here we create a hash value of the returned stream objects and store them in a ThreadLocal object.

Every time the write-method is called, we create the hash value of the object and check if this value is known.

In Section 2.6.1, we mentioned the common practice to use other classes to handle the InputStream or OutputStream objects. Since we are not able to weave into standard Java class libraries, we cannot recognize if any standard class calls the read or write method. This makes it necessary to implement another mechanism to monitor the communication. A possible implementation of a TCP server is shown in ??.

The InputStream object is given to the class InputStreamReader.

3https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html
4.3. Kieker Monitoring Component

Afterwards, the InputStreamReader object is also given to the BufferedReader class. In this case, we monitor the write-method of the BufferedReader object. Firstly, we define a pointcut for the constructor of the InputStreamReader class which has an InputStream object as a parameter. Additionally, we define an afterReturning-advice which checks if the hash value of the InputStream object is stored in our ThreadLocal object. If this is the case, we will replace the hash value of the InputStream with the hash value of the InputStreamReader.

Similar to this procedure, we define a pointcut and afterReturning-advice for the BufferedReader.

Lastly, we extend the advices for the incoming communication with a pointcut for the

Figure 4.3. Sequence of TCP operations on client-side. The red arrows mark the position where advices are woven into the code.
4. Adaptations to the Kieker Monitoring Framework

reading and writing on the BufferedReader.

We limit the implementation to the classes we use in our evaluation examples. The implementation does not claim to be complete.

All in all, we conclude that we need more pointcuts and advices than records we create to monitor a TCP connection. The Figure 4.3 shows the sequence of Socket operations and highlights the position where the pointcuts apply. We need thirteen pointcuts and advices to generate the eight communication records.

SocketChannel and ServerSocketChannel

As we mentioned previously, we have to define additional probes to support other communication technologies. The SocketChannel and ServerSocketChannel classes offer another possible to communicate with TCP.

We describe the usage of these classes in Section 2.6.1.

The utilization of these channel classes is quite similar to the usage of the socket classes. That is why we extend the existing probes for the communication beginning and communication end.

This also includes the mechanism to identify the close event on the server-side.

A difference to the socket objects is that the channel objects have own read and write methods. We do not need to monitor additional objects. Thus, we need only eight advices to monitor the communication. Therefore, we implement our own probes for the reading and writing on the channels.

4.4 Kieker Analysis Component

This section presents the changes to Kieker’s analysis component. As previously mentioned, this work uses the new Kieker analysis architecture. Section 2.5 describes the architecture of Kieker’s analysis component which we use.

First of all, we present the change on the Trace and OperationCall classes.

Afterwards, we focus on changes to the trace reconstruction.

4.4.1 AsynchronousTrace Class

The current Trace class represents a trace inside the analysis component. Depending on the monitored application the analysis produces a various number of Trace objects. Every Trace object represents a single trace of the application. Based on the Trace object it is possible to define different analysis and graphical representations.
4.4. Kieker Analysis Component

The Trace class defines a tree structure of OperationCall objects to represent a trace. Every OperationCall has a reference to its parent OperationCall and a list of child OperationCalls. The child OperationCalls represent operations which are executed inside the parent operation. For example, all operations inside a method are child OperationCalls from the method OperationCall.

To allow a trace with asynchronous operation, we add another class called AsynchronousTrace. This class represents a client-side trace with the associated operations from the server-side. It derives all properties from the Trace class. Moreover, we add a list of OperationCalls which contains the root OperationCall of the related traces. Inside of this class, we connect each communication operation on one side with the associated communication event on the other side. The objects of this type are interstage-products which we aggregate into distributed traces in a following step.

4.4.2 Modification to the OperationCall Class

The OperationCall class represents a common operation. As mentioned previously, the OperationCall stores relations to its parent and children. To enable a reconstruction of asynchronous traces, we need relations for the connected communication operations. For this purpose, we use additional classes which derive from the OperationCall class. Figure 4.4 presents the class diagram of the OperationCall class. The created classes are in accordance with the monitored communication events. Moreover, the ClientSideComOpCall includes a relation to a ServerSideComOpCall. This represents the connection between the Socket.connect() method on client-side and the ServerSocket.accept() on server-side. Additionally, we store a reference to an IncomingComOpCall inside of an OutgoingComOpCall, to represent the connection between the writing and reading methods. In contrast to the OperationCall class, our defined classes have an influence on the structure. Thus, we have to modify the processing of OperationCall at some points to enable a correct reconstruction. We present the changes in 4.4 in a more detailed way.

4.4.3 Reading Composite

To realize our approach, we make a few modifications to the ReadingComposite. The ReadingComposite reads, reconstructs, and transmits records to the following stages. Currently, the Kieker Analysis component can only read from the file system.

The ReadingComposite expects an input directory which contains the map file and a various number of record files. The map file is related to the record file and necessary to deserialize the records.
4. Adaptations to the Kieker Monitoring Framework

Figure 4.4. This figure represents the OperationCall class and the additional classes we add during this work. This figure does not include any methods.

In case that we have records from more than one node, we have to ensure that the map files are assigned correctly. To achieve this, we determine that an input directory holds multiple directories. Each directory contains the map and record files from a single node.

To import the records correctly, we change the structure of ReadingComposite stage. The current ReadingComposite stage contains InitialElementProducer\(^4\) and the Dir2RecordFilter stage. The InitialElementProducer takes the elements from the given import directory and passes the records to the next stage. Then the Dir2RecordFilter deserializes the records and transmits them to the output port. The output port of the second stage is the output port of the ReadingComposite.

Our modified stage behaves differently. In the beginning, the modified ReadingComposite tests the import directory, whether it includes other directories. In this case, the ReadingComposite creates an InitialElementProducer and Dir2RecordFilter stage for each directory. Afterwards, the output ports of the Dir2RecordFilter stages are connected to a Merger\(^5\). The Merger’s output port is the output port of to the ReadingComposite stage.

This structure enables a simultaneous reading of records from different nodes. Figure 4.5 shows the schematic structure of the ReadingComposite before and after our changes.

\(^4\)https://teetime-framework.github.io/apidocs/teetime/stage/InitialElementProducer.html
\(^5\)https://teetime-framework.github.io/stabledocs/teetime/stage/basic/Merger.html
4.4. Kieker Analysis Component

Figure 4.5. a) The architecture of the ReaderComposite stage without changes. b) The modified architecture of the ReaderComposite stage, to import multiple monitoring information.

In case that we import a directory with monitoring information from a single node, the ReadingComposite works similar to the previous implementation.

If we read records from multiple nodes, the ReadingComposite publish the records without any preferences. While reconstruction the traces, we cannot assume that we are going to receive the client-side records before the server-side records or similar sequences. This behavior is similar to any network reader, which receives records from more than one node while the nodes are monitoring.
4. Adaptations to the Kieker Monitoring Framework

4.4.4 Trace Reconstruction

We separate the deployment of the trace reconstruction into two parts. The first part focuses on the reconstruction of traces with TCP events and the connecting of TCP events. For this purpose, we adopt the general concept of trace reconstruction from the TraceReconstructionStage\(^6\) and modify the procedure to match our requirements. Therefore, we create the DistributedTraceReconstructionStage. After finishing the first step, we are able to analyze TCP communications and create the previously mentioned AsynchronousTrace objects.

The second part deals with the schema based aspect of our approach. This includes the aggregation of the communication events to create a distributed trace.

In the following, we present the procedure of our trace reconstruction. The essential procedure of the DistributedTraceReconstructor and DistributedTraceReconstructionBuffer is equal to the procedure of the TraceReconstructor\(^7\) and TraceReconstructionBuffer\(^8\). Additionally, we create the TraceMerger class. This class handles the dependencies between traces and manages the reconstruction and merging of traces.

The DistributedTraceReconstructionStage differentiates between before, after and TraceMetadata-records and calls the corresponding methods of the DistributedTraceReconstructor. If the received record is a TraceMetadata-record, the DistributedTraceReconstructor creates a DistributedTraceReconstructionBuffer object configured with the information of the TraceMetadata-record. The TraceMetadata-record is the first record of a new trace. Any probe will send a TraceMetadata-record if it is the first probe in the current trace.

In case that DistributedTraceReconstructionStage receives a before or an after-record, the DistributedTraceReconstructor passes the record to the DistributedTraceReconstructionBuffer with the same trace identifier. The DistributedTraceReconstructionBuffer creates an OperationCall object for every incoming before-record. Afterwards, the buffer adds every new OperationCall object as a child from this OperationCall until he receives the associated after-record. The first incoming before-record is the root OperationCall of this trace and the root of the tree structure. The trace is complete once the DistributedTraceReconstructionBuffer receives the after-record to the root OperationCall.

If the DistributedTraceReconstructionBuffer receives an AfterOperationFailedEvent, he adds the failure cause to the OperationCall object and marks it as failed.

\(^6\)https://github.com/kieker-monitoring/kieker/blob/issue-1475-trace-analysis-tool-migration/kieker-analysis/src/kieker/analysisteetime/trace/reconstruction/TraceReconstructor.java
\(^7\)https://github.com/kieker-monitoring/kieker/blob/issue-1475-trace-analysis-tool-migration/kieker-analysis/src/kieker/analysisteetime/trace/reconstruction/TraceReconstructor.java
\(^8\)https://github.com/kieker-monitoring/kieker/blob/issue-1475-trace-analysis-tool-migration/kieker-analysis/src/kieker/analysisteetime/trace/reconstruction/TraceReconstructionBuffer.java
4.4. Kieker Analysis Component

Since the analysis component we are working with is still under progress, the DistributedTraceReconstructionBuffer just counts the failed operation. There is no proper retouching work.

At this point, our implementation deviates from the former implementation. The DistributedTraceReconstructionBuffer also differentiates between common records and communication record. If DistributedTraceReconstructionBuffer receives a communication before-record, the buffer creates particular communication OperationCall objects. We present the possible OperationCall types in Section 4.4.2.

As we mentioned before, the OperationCall types are in accordance with the TCP events and have additional properties for the addresses and to link the events afterwards. While adding the information to the communication OperationCall, we have to consider which records contain the correct addresses. We mentioned the restriction in Section 4.3. While reconstructing the traces, we cannot connect any communication events. Besides the reconstruction, the DistributedTraceReconstructionBuffer prepares the subsequent merging by storing additional information. If the DistributedTraceReconstructionBuffer adds a ClientSideComOpCall to the trace, he increases a dependencies-counter to note that this trace needs operations from another trace. Moreover, the buffer calls the TraceMerger, which itself stores the dependency to enable a reconstruction afterwards.

In case that the DistributedTraceReconstructionBuffer processes an AfterServerCommunicationEndRecord, we generate and store a TracePart object. A TracePart contains a copy of all OperationCall between the beginning of a communication and ending of a communication on a server-side. After the reconstruction, we merge these TraceParts into the client traces.

A downside of our approach is the sequence of records which the analysis receives if we monitor a synchronous communication. Our approach determines that the communication records were send before and after the actual method is executed. As a result, the DistributedTraceReconstructionBuffer will receive the before and after-records from the communication beginning event first and will use this event as root. Since the second record is the after-record for the root OperationCall, the trace is according to the definition complete. To avoid this behavior, we define the EmptyRootOpCall class which we use as a space-holder for the root OperationCall. We represent the use of the EmptyRootOpCall class in Figure 4.6. The EmptyRootOpCall gets use once we analyze a synchronous communication. If the DistributedTraceReconstructionBuffer processes a ServerSideComOpCall, it adds an EmptyRootOpCall object as root from
4. Adaptations to the Kieker Monitoring Framework

![Diagram of trace reconstruction](image)

Figure 4.6. This figure represents the use of the EmptyRootOpCall class.

As we defined previously, an DistributedTraceReconstructionBuffer is complete once he receives the after-record from the root OperationCall. After each after-record, the TraceMerger verifies if the DistributedTraceReconstructionBuffer is complete. If a DistributedTraceReconstructionBuffer is complete, the TraceMerger tries to expand the client traces. Therefore, we define an additional status for the DistributedTraceReconstructionBuffer class. A DistributedTraceReconstructionBuffer is finished as soon as the internal dependencies-counter is zero.

If the TraceMerger processes a complete and finished DistributedTraceReconstructionBuffer, the TraceMerger takes potential existing TracePart from the DistributedTraceReconstructionBuffer and stores them. Afterwards, we create an AsynchronousTrace object for this trace.

Additionally, we try to find the destination of the TraceParts. If the TraceMerger processes a complete DistributedTraceReconstructionBuffer, the TraceMerger tries to find the corresponding TracePart and merge them.

There are two possible scenarios. Firstly, the TraceMerger gets a TracePart before he knows the client DistributedTraceReconstructionBuffer. In this case, the TraceMerger stores the TracePart and as soon as the corresponding DistributedTraceReconstructionBuffer is complete, the TraceMerger merges the operations.
The second scenario is that the DistributedTraceReconstructionBuffer is complete before the TraceMerger receives the TracePart. In this case, the TraceMerger stores the DistributedTraceReconstructionBuffer and merges the traces as soon as he receives the TracePart.

As long as the DistributedTraceReconstructionBuffer does not contain all server-side operations, we cannot create a AsynchronousTrace object for the client trace.

The merging of a TracePart into a DistributedTraceReconstructionBuffer is done by the DistributedTraceReconstructionBuffer itself. This merging is separated into two steps. At first, we run through the trace and the associated TracePart to store all relevant communication operations. Afterwards, we connect the communication beginning events from the client with the communication beginning events on server-side in the order of their execution. Furthermore, we connect the OutgoingComOpCalls from one side with the IncomingComOpCalls from the other side. Currently, we do not connect communication ending events, because the nodes can close the Sockets without affecting each other.

At this point, the DistributedTraceReconstructor returns AsynchronousTrace objects for the client traces. To perform the second step of the reconstruction, we use the TraceAggregator class. This class runs through the asynchronous traces. If it reaches a ClientSideComOpCall, the communication operation calls are replaced by the OperationCalls from the connected server-side. Afterwards, the trace includes only OperationCall objects. Then, the AsynchronousTrace object gets reduce to a Trace object and transmit to the output Port the stage.

To make the analysis customizable, we use a boolean in the configuration of TeeTime. If this is set to false, all modifications related to the aggregation are disabled. Since the disabling just affects the aggregation, the DistributedTraceReconstructionStage can return AsynchronousTraces when the aggregation is disabled. This allows analyzing TCP connection.
Chapter 5

Evaluation

This chapter focuses on the feasibility of the previously presented approach and its implementation. While working on this project, an issue arose regarding AspectJ. Our approach assumes that we can monitor the TCP communication irrespective of the actual communication technology. Unfortunately, AspectJ is not able to weave code into standard Java libraries. Because of that, we could not achieve a general probe which covers all TCP-based communication. To evaluate the functionality of the analysis, we examine and create additional probes for some major communication technologies. The focus of the additional probes is to test the producibility of probes for the particular communication technology by using AspectJ.

In the first part of the following evaluation we present the results of monitoring simple TCP-based communication. Afterwards, we evaluate the implementation with a simple REST communication by using a particular probe. In addition, we are going to discuss the possible adoptions for SOAP and RMI. The last section examines ApacheMQ. In this connection we discuss the opportunity to use our approach with this kind of communication.

5.1 Monitoring and Analyzing TCP-based communication

Since our approach focuses on the communication with TCP, we test our implementation with simple TCP examples. Firstly, we examine the feasibility of our implementation. Afterwards, we test the performance.

5.1.1 Feasibility

First, we examine implementations which use the Socket and ServerSocket-classes. In this context, we notice a first consequence of the limitations from AspectJ. The Socket class offers two possible ways to open a connection. One possibility is to call the connect()-method. The second is to use the constructor which requires the host and port number.
5. Evaluation

This constructor initializes the object and establishes the connection. We assume that the constructor calls the `connect`-method itself, but we are not able to weave our advice around the call. That is why, we expand the advice and add a pointcut for the constructor. After this modification, our probes work with the tested examples.

Our basic example corresponds to the code in Section 2.6.1. After analyzing the monitoring information, we receive a call-tree for an asynchronous trace. The asynchronous trace contains the client trace with the server-side operations and connections between the communication events. Figure 5.1 displays a reduced call-tree of the asynchronous trace.
5.1. Monitoring and Analyzing TCP-based communication

As mentioned in Section 4.3, we also support TCP communication which uses the SocketChannel and ServerSocketChannel classes. These probes also work with the tested examples and we are able to reconstruct a call-tree of the asynchronous trace. Moreover, we use these probes while monitoring the Apache Grizzly framework. In this connection, we are also able to monitor the TCP communication events. Unfortunately, we discover that each TCP event was executed in a different trace. We discuss the problem more detailed in Section 5.2 while evaluating the REST communication.

5.1.2 Performance

While realizing the approach, we add a few mechanisms to receive the correct monitoring information. Since we are concerned, that these additional mechanisms cause a significant overhead. We test our probes with different Kieker configurations to be able to evaluate the performance.

All tests were executed on the same system. Our system is using an Intel Core i5-5200U with 2.20GHz, 8GB memory and a 256GB SSD. Furthermore, we are executing the client and server application in different Java Virtual Machines (JVM) on the same system. This avoids side effects caused by the network and makes the test executions contrastable.

The basic example is a simple TCP communication between a client and a server. The Client sends a string with 17 characters to the server. Then the server changes all characters to uppercase and returns the modified string to the client. This process is repeated 50,000 times.

We execute the example with two different Kieker configurations. The first one just includes a general probe to monitor every operation call. The second configuration includes the general probe and also our implemented probes for the Socket and ServerSocket classes.

We measure the time beginning from the after-record of the first connect-method on the client-side. On the server-side, we begin the measurement with the after-record of the first accept-method. This procedure ensures that the measured duration includes the communication and excludes the setup time. For our tests, we start the sever first and once the sever waits for a connection we are going to start the client application. Therefore, the total execution time of the server includes the interval where the server waits for the client. Since we are starting the client manually, these intervals can deviate from each other.
5. Evaluation

Firstly, notice that the execution times from the client and server relativity equal. The deviation is not measurably as long as we not use nanoseconds. Since we chose a large mound of requests, we use seconds as time unit. Hence, we the results from the client and server are equal. The executions return an average difference of 10.74 seconds for the client and server. This is equal to 18.59 percent.

Moreover, we determine a difference in the amount of records. The executions with our probes generates 400,000 records more. This is corresponding to eight records for each connection which is to be expected.

To demonstrate the disadvantage, we enlarged our example. We add a ByteArrayInputStream objects on the server-side and execute the read-method ten times. In the end we close the stream by calling the close-method. The ByteArrayInputStream class extends the InputStream class and implements the Closable interface. Thence, our pointcuts for Closable.close() and InputStream.read() method calls will weave our advices around these method calls, but we will not get additional records.

We execute the example with the same Kieker configurations as before. Unfortunately, the execution of this example does not demonstrate a larger overhead. We assume that, the JMS optimizes the handling of the ByteArrayInputStream.

Since we separate the probes for the events in several files, we are able to adjust the configuration on the requirements. While monitoring an application which does not use the Socket and ServerSocket classes we can disclaim the probes and reduce the additional effort. Further, we recommend limiting the aspects on the packages which execute the TCP communication.

5.2 Monitoring a Restful Communication

In a further step, we test the implementation with a REST communication, to evaluate our approach. We are using the Jersey (JAX-RS) implementation which is part of the standard Java class libraries. Here we add the @Path annotation to the server class and define the REST-method by adding the @GET, @SET, @DELETE or @CREATE annotation. Then the service runs with the Apache Grizzly framework2.

1https://docs.oracle.com/javase/7/docs/api/java/io/ByteArrayInputStream.html
2The Grizzly Framework is developed by Oracle and includes several components to take advantage of the Java NIO.
5.2. Monitoring a Restful Communication

While monitoring the execution of a request on the server, we receive the server-side records. If we limit the monitoring scope on our example classes and the REST implementation we do not get any communication record. We have to enlarge the scope on the Grizzly server which handles the TCP communication to receive the anticipated records. After taking a closer look on the records, we can determine that these records do not work with our analysis. The Grizzly server uses the SocketChannel and ServerSocketChannel class and modularizes the TCP communication in several tasks. As a result, the monitored events are part of different traces. We could not associate the TCP events to the executed request.

On client-side, we use the HttpURLConnection class so sent a Get request to the server. As we already expected, we are not able to apply our probes for the TCP communication on this implementation. To test the changes on the analysis component, we implement additional probes to instrument the Jersey implementation and the HttpURLConnection class.

On the server-side, the method which handles an incoming get request is marked with the @javax.ws.rs.GET annotation. We define the pointcut for the annotation and before, afterReturning, and afterThrowing advices. The pointcut focus on the execution of the method and not on the calling of the method. The calling of the method is done inside of the Jersey implementation which we cannot instrument with AspectJ.

To fit our approach, the before and afterReturning advices send four records respectively. Before the method execution, the before-advice sends the records for the communication beginning event and communication receiving event. After the method execution, the afterReturning-advice sends the records for the communication sending event and communication end event. This usage is not equated to the meaning of the records. It just ensures that the analysis component receives the records in the right order to perform the trace reconstruction correctly.

On client-side, we create an additional probe for the HttpURLConnection class. The probe is similar to the probe for the server-side. The before-advice generates the records for the communication beginning and the records for the sending event. After the execution, the afterReturning-advice generates the records for the receiving event and the connection closing. In contrast to the probe on the server-side, we can use pointcuts for the method call.
5. Evaluation

While creating these probes, a challenge is to get the IP address from the client and server-side. The client knows the server it wants to connect to so we extract the server address, but we cannot receive the client address. Moreover, we cannot determine the port numbers.

On the server-side, we cannot get any address. That is why we add the server address in this probe hard-coded, to allow testing. The client address is replaced by "<unknown client addr>" as simple space-holder on both sides.

Finally, these additional probes enables us to generate records for the analysis. Figure 5.2 presents the returned call-tree after connecting the TCP events and merging the server-side operations.

All in all, our approach does not work correctly with the Jersey implementation. There are some significant issues which prohibit correct monitoring.

Firstly, we are able to receive monitoring records with our TCP probes, but these are incompatible with our trace reconstruction.

Secondly, the appropriate probes do not fit well to the defined records and cannot access the IP addresses.
5.3 Monitoring SOAP based Communication

Hence, the IP address does not seem to be a proper identifier for a REST communication. While implementing the probe, we discovered another disadvantaged of AspecJ. As described before, we defined the pointcut to match on every method with the @GET annotation. This pointcut works as long as the restful service uses the annotation inside of the class. If the services defines an interface with the methods and the corresponding annotation, our pointcut does not match. AspectJ respects the Java regulation, that annotation are not inhered.\(^3\)

This problem could be solved by modify the instrumentation. It is necessary to check for each method whether the implemented interfaces contains a method with the same name and the requested annotation. We could limit this test to methods with an @Overwrite annotation, but Java does not prescribe the @Overwrite annotation for methods which implement an interface. As consequence, this procedure has to be done for every method, which will likely generate a significant overhead.

5.3 Monitoring SOAP based Communication

In addition, we test our probes with a common SOAP based communication. The evaluated example uses Java 1.8 and the JAX-WS implementation which is part of the JDK since version 1.6.

We define a web service by adding the @WebService annotation to the service class. Inside of this class we add @WebMethod annotation to a method which makes this method accessible via SOAP. Additionally, we could define an interface with the annotations and implement a class which uses the interface. If we use an interface with the annotation, the implemented methods do not require the @WebService annotation.

The service starts by calling javax.xml.ws.Endpoint.publish-method with addresses and an instance of the server class. As soon as the server is published, the WSDL\(^4\) file is available by calling the servers address with the query '?wsdl'.

Beside the server implantation, we add a simple client class which calls the service methods. Therefore, we use the WSDL file and the wsimport-tool\(^5\) to generate the necessary classes. The wsimport-tool reads the WSDL file from a published web service and create the classes


\(^{4}\)http://www.w3.org/TR/2001/NOTE-wsdl-20010315

\(^{5}\)https://docs.oracle.com/javase/6/docs/tecnote/tools/share/wsimport.html

47
5. Evaluation

for the client.

While monitoring the example with Kieker, we determine that the TCP probes cannot be applied to the implementation. Subsequently, we take a closer look at the usage of the SOAP implementation to realize a proper probe.

Firstly, we examine the possibility to get access to the IP addresses. This is possible by using the `javax.xml.ws.WebServiceContext` object. This object allows us to access the `javax.xml.ws.handler.MessageContext` which contains the SOAP Header with the IP addresses of the remote node. We have to note, that the Header just contains the IP address and not the port number.

The second challenge is the definition of the pointcuts. We want to weave the advices around the execution of method with the `@WebMethod` annotation. We use the execution keyword for the pointcuts, because the actual method call takes place inside the SOAP implementation.

As mentioned in Section 5.2, the pointcut will match as long as the service does not use an interface with the annotation. Probably, the implementation with interfaces is the common practice while working with JAX-WS.

In summary, it is possible to define a probe which could work with some applications, but the probe does not work with any application.

Further, it is not reasonable to use the IP address and port number as long as we focus on this application level, since we are not able to access the IP address and especially the port number.

These results are similar to the results we receive from the prior evaluation of restful services.

5.4 Remote Procedure Invocation (RMI)

Since the concept of Remote Procedure Calls is quite popular, we also test our implementation with a RMI example.

In our example, we use the standard RMI implementation which is part of the Java libraries. As discovered before, AspectJ could not weave our probes inside the RMI implementation. Therefore, we examine the possibility to define own probes.

To implement a RMI server, it is necessary to create an interface which includes remote methods. This interface has to deviate from the `java.rmi.Remote` interface. Then a class can use the interface and implement the methods.

As mentioned in Section 5.3, we cannot define proper pointcuts on interfaces, but we
5.5. Java Message Service

All methods, which are available with RMI, have in common, that they can throw a RemoteException. Hence, it is possible to define pointcuts for the methods.

The second challenge is to gain access to the IP addresses and port numbers. Here we do not find a proper solution. In our opinion, the combination of IP address and port number to identify a communication is not reasonable if we are working on this application level.

5.5 Java Message Service

Beside the previously mentioned communication technologies, we thought about monitoring a Java Message Service (JMS) with our probes.

As described in Section 2.6.5, JMS defines an asynchronous communication. The communication is quite similar to a TCP communication, but on a higher layer in the OSI model. That’s why we cannot generate distributed traces out of the monitored information. Nevertheless, we monitor a simple example to evaluate if it is possible to create an asynchronous trace similar to the results of the TCP communication.

For the evaluation we use an example based on the HelloWorld example which is part of the Apaches ActiveMQ documentation. The example contains a producer which sends a string to the consumer. Unfortunately our pointcuts do not match. Based on the output from AspectJ we assume that ActiveMQ uses RMI. Since, we are not able to monitor RMI communication, we cannot evaluate our approach with ApaceMQ. In case that, we could monitor RMI, a major challenge is to follow the message though the message broker. Maybe this demands that the message broker is also instrumented.

5.6 General Awarenesses Related to Monitoring Distributed Traces

While evaluating our approach, the major problem was to instrument the communication. Firstly, this is caused by AspectJ and its limitation to weave into standard Java class libraries. In this connection we examined an approach to weave into standard Java libraries with AspectJ. [Villazón et al. 2008] presents an approach which enables us to apply AspectJ aspects on standard Java class libraries. Unfortunately, this approach offers a significant

\[\text{http://activemq.apache.org/hello-world.html}\]
5. Evaluation

overhead for the evaluated examples. Thus, this work does not fit on the requirements of application monitoring.

Secondly, we revealed some issues while defining the pointcuts for the communication. If we use pointcuts which refer to annotations we cannot ensure that these will work for all applications.

Another challenge related to the monitoring of distributed system is that the monitored systems can have different system times. In Chapter 3 we mentioned the consequences of different system times for the trace reconstruction.

Furthermore, different system times also have influences on the meaningfulness of the resulting analysis. The timestamps of the operations we added from the server-side in a distributed trace refer to another system time and can differ from the other timestamps. A possible solution to adjust the timestamps is to assume that the communication start events on both sides were executed sequential. Then we can synchronize the timestamps from server-side operations with the communication start event.
Chapter 6

Related Work

This work is not the first to discuss the monitoring and reconstruction of distributed traces. There are two works related to Kieker which we want to present in the following. Afterwards, we take a look at other tracing tools.

6.1 Kiekers SOAP Probes

Since Kiekers version 0.9, there are probes to monitor SOAP. Unfortunately, there is no documentation for the SOAP implementation. That is why we have to make statements based on the code. The implementation supports SOAP communication based on the CXF implementation.

In contrast to this work, the implementation use Interceptor\(^1\) to instrument the SOAP communication. An interceptor enables to modify the request and response objects before they were sent respectively before the application processed them.

The general idea is to exchange different information between the client and sever application to enable the server to continue the client trace on the server. At first the client application adds the necessary information to the Header of the SOAP request. As soon as the server receives the request, the server extracts the trace identifier and use this identifier thenceforth. Once the server processed the request, the server returns the trace identifier in the Header of the SOAP response.

This approach does not affect Kieker’s analysis component. The server receives enough information about the current client trace to assign the monitored operations to the client trace.

The approach works with SOAP communication, so we can use the SOAP Header. The Header allows storing additional information without any additional effort. A downside of this approach is that it depends on the communication technology. Since we want general solution, we do not reused this approach.

As far as we know, this approach will not work with the reworked analysis component.

\(^1\)http://docs.oracle.com/javaee/6/tutorial/doc/gkeed.html
6. Related Work

The analysis demands that the records pass the architecture in a chronological order.

In case that we store the records of the client and server on the file system, it is possible that the analysis completes the client trace first and receives the operations from the server-side afterwards. Then the trace reconstruction returns a trace for the client without the server-side operations and a trace with the same identifier which contains the server-side operations. In case that we execute the monitoring and analysis at the same time, the server-side operations are added at the current position of the trace reconstruction. If we have a constant network overhead and the monitoring records for the client and server need the same amount of time, this could work. If anything changes, the reconstructed trace will be incorrect.

Nevertheless, we notice that this approach uses trace identifier on several nodes. While monitoring one independent node, there is only one instance of Kieker monitoring which creates and uses trace identifiers. In this case, we can guarantee unique trace identifiers. This implementation and also our implementation uses the trace identifiers on several nodes. This is possible because the probability of generating the same trace identifier is rather low.

6.2 Monitoring of Remote Procedure Calls

This approach was developed by Niels Matthisen in context of his bachelor thesis. The approach focuses on Remote Procedure Invocation (RMI). It was developed in the context of a high-throughput version from Kieker. Unfortunately, we were not able to examine the code.

The general idea of his work is to sent the trace identifier from the client to the server and the server-side monitoring creates a monitoring record as soon as he receives the information. Therefore he defines three monitoring records. The first is the so called SentRecord. If a client sends a request to a server, this record is created by a client. It contains the clients trace identifier, order identifier and the destination. The second type is the ReceivedRecord. This record represents that a node received a request with information from the client. The record contains the clients and servers trace identifier and order identifier. Lastly, the so-called UnknownReceivedRecord marks the receiving of a record without additional monitoring information. This record contains the servers trace identifier and order identifier and also information about the sender and destination of the request. It is used when the client is not monitored with Kieker. An interesting aspect of his work is, that his evaluation offers an overhead rate of 3.6. Due to the missing implementation, we cannot examine the cause of this overhead.
6.3 SPASS-meter

SPASS\(^2\)-meter is an application-level monitoring tool [Eichelberger and Schmid 2014]. A key feature of SPASS-meter is the monitoring of resource consumption. As far as we know, SPASS-meter supports monitoring of the network utilization, but does not support a tracing of a distributed application [Eichelberger et al. 2015].

6.4 Google Dapper

[Sigelman et al. 2010] introduces the Dapper infrastructure, which is developed by Google. They describe Dapper as production distributed systems tracing infrastructure. The goals are to design a tracing system with a low-overhead, an application-level transparency and scalability on a very large scaling distributed system.

At that time, the tools supports RPC in Java and C++. Beyond that, they also trace SMTP sessions, HTTP requests and outbound queries to SQL server.

To trace an execution, Dapper adds an identifier to the request. Based on this identifier each instrumented point can identify the request. The tool uses an annotation-based monitoring schemes to gather the information. This means that the application is monitored based on annotations in the implementation. The tool is restricted to specific implementations for the communication which all applications are using. These implementations include the instrumentation to receive the information for the tracing. This instrumentation scheme fits the use with Google distributed systems, but makes the tool less portable.

Since this tool is designed for large scaling distributed systems and is used with productive systems, a core requirement is a low-overhead. To reduce the overhead, Dapper processes just one samples trace for every 1024. Hereby, they determine an average additional overhead under one percent for latency and throughput.

There are several tracing tools which are based on Dappers architecture. There is for example the appdash\(^3\) which is an application tracing tool for Go\(^4\) and Twitters OpenZipkin which we present in the following Section.

\(^2\)SPASS - Simplifying the Development of Adaptive Software Systems
\(^3\)https://github.com/sourcegraph/appdash
\(^4\)https://golang.org/
6. Related Work

6.5 OpenZipkin

OpenZipkin (Zipkin)\(^5\) is a distributed tracing system created by Twitter. The project is open source under the APLv2 license and available on GitHub\(^6\). The design of OpenZipkin is based on the Google Dapper paper [Sigelman et al. 2010]. We present the design of Dapper in Section 6.4.

OpenZipkin instruments several libraries for Go, Java, JavaScript, Ruby, and Scala. In context of Java they currently support Jersey, RestEASY, JAXRS2, Apache HttpClient and Mysql.

Furthermore, there are community implementations to instrument other libraries. The general procedure is to add an identifier to a request and to pass them along to all services. It is designed to run on productive systems. Therefore, OpenZipkin uses sampling to reduce the overhead caused by the monitoring. Then the results of their monitoring are accessible in a Web UI.

The OpenZipkin project consistently adds further implementations to cover enlarge their supported libraries.

6.6 Magpie

[Barham et al. 2003] introduces the Magpie online modelling infrastructure. The goal is to collate traces from multiple systems to generate models of the request behavior. This includes a blackbox-instrumentation, which means an instrumentation of an application without knowledge about the application.

To track a request, Magpie uses an identifier which is added to the head of each request. Magpie uses existing tracing tools on each node and synthesizes the monitored events into models.

Mentioned tools in this work are, Event Tracing for Windows\(^7\) and the .NET Profiling API\(^8\). Afterwards the created models are combined by a query engine, to receive the distributed traces.

6.7 X-Trace

X-Trace is a monitoring tool for distributed systems [Fonseca et al. 2007]. It is designed to support different networks. The goal is to trace the execution of requests in a system and visualize the resulting trace. Therefore, X-Trace annotates the network

\(^5\)http://zipkin.io/
\(^6\)https://github.com/openzipkin/
\(^7\)https://msdn.microsoft.com/de-de/library/windows/desktop/bb968803(v=vs.85).aspx
\(^8\)https://msdn.microsoft.com/en-us/library/bb384493(v=vs.110).aspx
requests with an identifier. This allows to reconstruct the execution of the request across multiple servers.
Conclusions and Future Work

7.1 Conclusions

This work discussed an approach to monitor distributed executions and generate distributed traces. The approach was designed with the aim to realize a general implementation which enables a monitoring of different TCP based communications. The idea of the discussed approach is to monitor the TCP communication and use the knowledge about communication to create distributed traces. To identify the communicating nodes in a distributed system, we used the IP addresses and port numbers.

We separated the trace reconstruction into two steps. This firstly created an asynchronous trace which included the TCP communication. Afterwards, we aggregated the TCP communication based on our knowledge about the communication. In case that we did not have a synchronous communication we could disable the second step. Hence, the analysis did not return distributed traces.

To evaluate the feasibility, we presented an implementation for the Kieker Monitoring framework. The evaluation of our implementation determined a few problems which restricted the implementation of our approach.

Since we worked with Kieker, we used AspectJ in our implementation to instrument the monitored application. AspectJ is used by Kieker to enable a dynamical instrumentation. While evaluating our implementation, we detected that AspectJ cannot weave our probes into standard Java class libraries. Since the majority of the communication implementations we tested are part of these standard Java class libraries, the dynamical instrumentation did not work.

Furthermore, we evaluated the trace reconstruction. Therefore, we surveyed the possibility to create proper probes for the tested communication technologies. While creating the probes, the main challenge was to find a way to access the IP address or the port number. In general, we were not able to access them. This demonstrates that the IP address and the port number are theoretical an adequate identifier but do not work in practice for communications in the application layer.
7. Conclusions and Future Work

Another observation we made is that it is difficult to define pointcuts for the tested implementations. We examined different pointcuts which refer to annotations. Unfortunately, Java does not inherit the annotations. If the annotations, which our pointcuts refer to, were used in interfaces, our pointcuts cannot match correctly. A major issue is that annotations are not inherited.

7.2 Future Work

In this work we are able to determine different issues related to the monitoring of distributed traces. Based on this knowledge, we are recommending another approach, which we would further examine. We can summarize that a schema based approach creates additional effort associated with the implementation as well as to the use of the analysis. Therefore, we recommend to use a metadata propagation approach. Many tracing tools like Googles Dapper and Twitters OpenZipkin and also the work in Section 6.2 demonstrate that this approach works with SOAP and RPC. We expect this approach to also work with REST, if we can add metadata to the HTTP Header.

Unfortunately, an approach based on metadata propagation requires an instrumentation for each communication. In our opinion, it is enough to transfer the current trace identifier from the client to the server. This information suffices to connect the traces. Further this guarantees a fixed size of data.

The most challenging task is to implement the dynamical implementation. We discussed the possibility to define probes with AspectJ. In this connection, we identify implementations which our probes cannot instrument. It should be worth considering further alternatives to AspectJ to instrument an application. In Section 6.1, we already present an alternative procedure to instrument a SOAP communication.

Moreover, we recommend to use other records. It should be enough to create a before and after record for the method which connects or receives a communication. For example, if we use RMI, we create a before and an after-records for the remote method call on client side. Additionally, we have a before-record before the remote method is executed and an after-record after the execution ends. This allows to create the traces correctly. The client trace will have a single OperationCall for the communication. In the server trace, the communication will be represented by an OperationCall and all operations, which are executed during the communication, are children of this OperationCall.

In case that one side of the communication is not monitored, these records will also fit.
we just monitor the server, we would generate the records anyway to mark the existing of
a communication.

7.1 represents our recommended approach.
In general, we need monitor the before and after-event on the client and server-side. Addi-
tionally, the client passes its trace identifier to the server.
Since the before event is executed before the server processes the request, the before-record
can access the transferred information. Hence, the after-record summarizes the trace identi-
ifier from the client and server.
Based on these records, a trace reconstruction should be possible.
It needs to evaluated if the trace identifier is enough, to allow a correct reconstruction.

Once Kieker is able to create distributed traces, it will be useful to make improvements
to the visualization of distributed traces. Currently, the hostname allows comprehending
which node execute an operation. To improve the clearness, we purpose to use different
colors or add shapes for the nodes.
7. Conclusions and Future Work

As soon as Kieker is able to monitor distributed systems, it can be tested with different systems. We assume that this approach will work properly with Load Balancer or the Message-Broker in *ActiveMQ*, as long as these do not modify the Headers of the messages.

Since our work provides an implementation to monitor TCP-based communication, it is possible to adopt this part of our work. In this connection, there are three aspects which need to be considered.

As mentioned in Section 5.1, our implementation could create a potential overhead in some cases.

Further, maybe the current instrumentation does not cover all possible scenarios.

Lastly, the analysis does not return the client trace as long as the server trace is not processed. If the information from the server is not available, we could think about returning the incomplete client traces.

A solution will be to flush the storage of the `DistributedTraceReconstructionStage` before the stage terminates and return all remaining client traces.

All in all we conclude, that our approach does not comply our requirements, but we were able to evaluate different challenges and to recommend further proceedings in this topic.
Appendix A

TCP Communication Test Results

Table A.1. Results of the execution from the basic example without the TCP probes

<table>
<thead>
<tr>
<th>Without TCP probes</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95.202.914.798 ns</td>
<td>95,20 s</td>
<td>95.202.536.441 ns</td>
<td>95,20 s</td>
<td></td>
</tr>
<tr>
<td>93.684.549.302 ns</td>
<td>93,68 s</td>
<td>93.685.468.837 ns</td>
<td>93,69 s</td>
<td></td>
</tr>
<tr>
<td>94.108.492.934 ns</td>
<td>94,11 s</td>
<td>94.108.395.429 ns</td>
<td>94,11 s</td>
<td></td>
</tr>
<tr>
<td>93.939.710.079 ns</td>
<td>93,94 s</td>
<td>93.939.447.421 ns</td>
<td>93,94 s</td>
<td></td>
</tr>
<tr>
<td>94.847.764.099 ns</td>
<td>94,85 s</td>
<td>94.847.764.099 ns</td>
<td>94,85 s</td>
<td></td>
</tr>
<tr>
<td>95.343.937.604 ns</td>
<td>95,34 s</td>
<td>95.343.616.631 ns</td>
<td>95,34 s</td>
<td></td>
</tr>
<tr>
<td>95.555.798.618 ns</td>
<td>95,56 s</td>
<td>95.555.581.215 ns</td>
<td>95,56 s</td>
<td></td>
</tr>
<tr>
<td>95.017.951.216 ns</td>
<td>95,02 s</td>
<td>98.102.089.783 ns</td>
<td>98,10 s</td>
<td></td>
</tr>
<tr>
<td>94.418.649.427 ns</td>
<td>94,42 s</td>
<td>99.463.015.215 ns</td>
<td>99,46 s</td>
<td></td>
</tr>
<tr>
<td>93.585.761.125 ns</td>
<td>93,59 s</td>
<td>98.831.565.644 ns</td>
<td>98,83 s</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2. Results of the execution from the basic example with the TCP probes

<table>
<thead>
<tr>
<th>With TCP probes</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115.761.427.699 ns</td>
<td>115,76 s</td>
<td>115.761.593.319 ns</td>
<td>115,76 s</td>
<td></td>
</tr>
<tr>
<td>108.237.238.952 ns</td>
<td>108,24 s</td>
<td>108.251.604.403 ns</td>
<td>108,25 s</td>
<td></td>
</tr>
<tr>
<td>109.850.107.956 ns</td>
<td>109,85 s</td>
<td>109.850.518.504 ns</td>
<td>109,85 s</td>
<td></td>
</tr>
<tr>
<td>108.795.509.257 ns</td>
<td>108,80 s</td>
<td>108.796.097.553 ns</td>
<td>108,80 s</td>
<td></td>
</tr>
<tr>
<td>107.730.990.690 ns</td>
<td>107,73 s</td>
<td>107.731.448.357 ns</td>
<td>107,73 s</td>
<td></td>
</tr>
<tr>
<td>102.839.529.344 ns</td>
<td>102,84 s</td>
<td>102.839.973.482 ns</td>
<td>102,84 s</td>
<td></td>
</tr>
<tr>
<td>100.157.106.028 ns</td>
<td>100,16 s</td>
<td>100.156.399.232 ns</td>
<td>100,16 s</td>
<td></td>
</tr>
<tr>
<td>101.800.110.195 ns</td>
<td>101,80 s</td>
<td>101.800.477.822 ns</td>
<td>101,80 s</td>
<td></td>
</tr>
<tr>
<td>102.825.813.771 ns</td>
<td>102,83 s</td>
<td>102.826.049.370 ns</td>
<td>102,83 s</td>
<td></td>
</tr>
<tr>
<td>102.560.564.709 ns</td>
<td>102,56 s</td>
<td>102.560.982.255 ns</td>
<td>102,56 s</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Project Overview

Our implementation can be found at https://github.com/HaStr/kieker/tree/hstr-thesis-distributed-traces. The project is a Fork of the Kieker project and the implementation is separated into a branch.

The example for our evaluation can be found at https://github.com/HaStr/kieker-evaluation.


Bibliography


