Dynamic Analysis of SOA through Monitoring and Tracing

Master’s Thesis

Gert-Jan Deen
Dynamic Analysis of SOA through Monitoring and Tracing

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gert-Jan Deen
born in Rotterdam, the Netherlands
Dynamic Analysis of SOA through Monitoring and Tracing

Abstract

We have identified three fundamental monitoring dimensions for the maintenance and evolution of Maritime Safety and Security Systems, which are representing a class of systems that are required to be continuously online. The monitoring dimensions consist of: 'services', 'users' and 'time'. As traces crosscut these dimensions, we are interested in learning how to extract traces of dynamic and distributed systems. Through a literature study we identified three fundamental online tracing techniques, from which all tracing approaches can be derived: Tagged Tracing, Recognized Tracing and Timestamped Tracing. We devised an evaluation setup on which we evaluated tracing techniques for dynamic and distributed systems. With 28 percent overhead Timestamped Tracing has the lowest overhead, but it only functions on systems using synchronous communication. Tagged Tracing on the other hand, operated correctly on systems employing both asynchronous and synchronous communication with an overhead of 43 percent. We did not evaluate the accuracy of Recognized Tracing as the accuracy is dependent on the implementation of the dynamic system. Recognized Tracing presented an overhead of 36 percent. We developed a dashboard which allows storing and loading of requests and traces, such that the reconstructed traces could be evaluated for correctness. In addition, the dashboard aids maintainers and developers by revealing the usage of the services cluster and the dependencies of services.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: H.G. Gross, Faculty EEMCS, TU Delft
Committee Member: Dr. S.O. Dulman, Faculty EEMCS, TU Delft
Words cannot describe my gratitude for the support of my family and friends. In addition, I would like to thank my daily supervisor Gerd Gross, for his patience and invaluable advice throughout my graduation project. Without his advice and guidance this thesis would be nothing more than a chaotic collection of words. Furthermore, I am grateful for the opportunity Logica gave me to perform part of my graduation at their site. Finally, I would like to thank the TU Delft for the education the TU Delft provides, that enriched my view on the world and taught me essential skills.

Gert-Jan Deen
Delft, the Netherlands
June 11, 2012
## Contents

Preface .................................................. iii

Contents ................................................. v

List of Figures .......................................... vii

1 Introduction ........................................... 1
   1.1 Outline ........................................ 3

2 Background and Terminology ....................... 5
   2.1 Transaction traces ................................ 5
   2.2 Monitoring ........................................ 7
   2.3 Types of Monitoring ................................ 8
   2.4 SOA Concepts ..................................... 11
   2.5 Monitoring applied on SOA ....................... 13

3 Tracing techniques for services ................... 17
   3.1 Tagged Tracing ................................... 18
   3.2 Timestamped Tracing .............................. 21
   3.3 Recognized Tracing ................................ 21
   3.4 Summary ......................................... 22

4 Implementation ......................................... 23
   4.1 Case Study System ................................ 23
   4.2 Monitoring architecture ......................... 25
   4.3 Information Processing ........................... 30
   4.4 Displaying Information ............................ 35

5 Evaluation ............................................. 41
   5.1 Tracing techniques ............................... 41
   5.2 Reporting Approaches ............................. 51
## CONTENTS

5.3 Threats to validity .................................................. 53

### 6 Related Work
6.1 Comparable approaches .......................................... 55
6.2 Existing monitoring platforms for SOA ......................... 57

### 7 Conclusions and Future Work
7.1 Contributions & Conclusions ..................................... 61
7.2 Reflection .............................................................. 63
7.3 Future work ........................................................... 64

Bibliography ............................................................... 65

A Code Extracts .......................................................... 73

B Glossary ................................................................. 77
## List of Figures

1.1 The three fundamental Monitoring Dimensions: 'Users', 'Services' and 'Time' 2

2.1 An example trace ................................................. 6
2.2 Two traces (Red, Blue) ........................................... 7
2.3 Applications connected without ESB compared to applications connected through ESB [76] ............................................. 13

3.1 Two traces (Red, Blue) ............................................. 17
3.2 Two tagged traces (Red, Blue) ................................... 19
3.3 Pushing data to the TaggedQueue ............................... 20
3.4 Popping data of the TaggedQueue ............................... 20
3.5 Service monitoring through the proxy pattern ................... 21

4.1 AIS service cluster ............................................... 24
4.2 Communication using the registry ............................... 25
4.3 Monitoring UML diagram ....................................... 27
4.4 One-sided communication monitoring ........................... 28
4.5 Schematical overview of two-sided communication monitoring 28
4.6 The monitoring proxies and analyzer ............................ 30
4.7 Class diagram of TraceTreeClass, TraceTree and Request ........ 32
4.8 The Dashboard Status tab shows an overview of all services that are registered. The Requests to and from a particular service indicate how active the service has been. .................................................. 36
4.9 The Dashboard Requests tab shows all raw requests logged. It displays when the request occurred (time stamp) and who send it (from) to what service (to). If Tagged Tracing is applied, the trace ID reveals information about the trace to which the request belongs. Optionally, the monitors collect payload information of the requests as well. .................................................. 36
4.10 The Dashboard Traces tab shows the transaction traces. The Traces tab is especially useful for recognizing patterns in the services usage. The occurrences reflect the number of times a trace with a particular trace path were executed. Trace Length reflects the length of the trace path. The Involved Entities denotes the services part of the trace are display. ................................. 37

4.11 The windows for inspecting a trace are shown. The first window shows the trace path. The second window shows the individual occurrences of traces, the duration of individual traces and at what time the trace took place. In addition, the average trace duration is shown. Individual occurrences can be further inspected as the third window shows. Here, each single request of an individual trace is displayed with the time of occurrence, services involved and optionally the payload. ................................. 38

4.12 The Dashboard Overview Tab shows a graphical overview of the dependencies between services. ................................. 39

5.1 Requests correctly matched on synchronous communication .................. 46
5.2 Requests correctly matched on asynchronous communication ................. 46
5.3 Correct trace paths on synchronous communication ............................. 47
5.4 Correct trace paths on asynchronous communication ............................ 47
5.5 The monitoring overhead as percentage of the normal runtime ................ 48
5.6 The LOC for each tracing method ................................................. 49
5.7 The computation time of the analyzer on 1 minute of AIS data. ............... 49

6.1 eBay SOA Platform [51] ............................................................ 58
6.2 eBay Service Provider Framework [8] ........................................... 59
Chapter 1

Introduction

Service Oriented Architecture (SOA) is a computing paradigm that advocates to the quick and easy replacement of business logic in order to come up with new functionality in rapidly changing business computing environments [25, 64, 65]. Through their dynamic nature, SOA can also be used to build systems that must never be switched off, even if they are maintained or extended. An example of a class of systems that have to be continuously online, are systems in the Maritime Safety and Security Systems Domain in which system failure has direct consequences on the safety and security of Maritime Traffic [53, 63]. As requirements evolve over time, the system has to be maintained and updated accordingly. In a recent industrial survey which assesses the Application Performance Monitoring (APM) Market published in 2011 by Gartner [10] an increase in attention for runtime application architecture discovery, modeling and display is noted. Often it is impossible for anyone - even the code authors - to definitively describe the execution. This is a result of increased complexity of the software, but also the high levels of abstraction in which modern applications are developed. When adding a new functionality, the developer understands only the new functionality, while the calls to existing software are hidden behind interfaces. Furthermore, in Service Oriented Architectures, the actual binding to a particular service happens at runtime, thus it is impossible to predict the actual execution offline [10]. Inspired by the survey of Gartner [10] we have distinguished three dimensions of interest. These three dimensions consist of ‘services’, ‘users’ and ‘time’. The goal is to answer the question: what services are used by what users at what time? Transaction traces cross-cut the Monitoring Dimensions. In a recent survey on Application Performance Monitoring (APM) published in 2011 by Gartner [10] describes the term transaction as used by most APM vendors as: A single operation from the perspective of the user, even though it may consist of a number of operations from the perspective of the actual system implementation [10]. Transaction tracing attempts to trace the effects of this transaction across a sequence of components and network paths.

Maritime Safety and Security Systems represent the class of systems that we target with this thesis. We have implemented the tracing and monitoring concepts on a service cluster operating on Automatic Identification System (AIS) data. AIS is a system which allows ships to communicate their positions and status, this is essential for the safety of Maritime Traffic [53, 63]. In Chapter 4 we further describe the case study system. The domain of
1. Introduction

Figure 1.1: The three fundamental Monitoring Dimensions: 'Users', 'Services' and 'Time'

the case study system requires the system to run continuously and the system should never be down as this has immediate implications on the security and safety of Maritime Traffic. Information concerning the three fundamental monitoring dimensions we identified are required, in order to perform maintenance on the system. As traces crosscut these dimensions, we are interested in learning how to extract traces of dynamic and distributed systems. This lead us to the following research questions:

- What are techniques for extracting traces in dynamic systems?
- What are the trade offs between these tracing techniques?
- How can we present and use the data extracted by the traces?

Also in scientific literature the utility of tracing for dynamic analysis has been recognized. In a recent survey published in 2009 by Cornelissen et al [17], over 30 approaches were identified which use traces for dynamic analysis. However, the majority of those approaches are on monolithic systems. Even though monitoring and tracing are well explored domains in computer science, there are few approaches that use these methods in dynamic or distributed systems like SOA [23]. Tracing is especially useful in SOA due to its dynamic nature. The distributed aspect makes tracing radically different on SOA than on monolithic systems [17, 23, 30, 57]. Through a literature study we identified three fundamental online tracing techniques, to which all tracing approaches can be derived: Tagged Tracing, Recognized Tracing and Timestamped Tracing. There are a number of approaches that use online tracing techniques to extract traces [10, 14, 23, 22]. However, to the best of our knowledge, there is hardly any literature on the evaluation of tracing approaches on SOA nor articles that compare the tracing approaches to each other. Therefore we devised an evaluation setup on which we evaluated tracing techniques for SOA.
1.1 Outline

Chapter 2 presents the underlying theory and dynamics of monitoring, tracing and Service Oriented Architectures. In Chapter 3 we describe the monitoring and tracing concepts which we implemented. Chapter 4 describes our implementation. A comparison of the three fundamental online tracing techniques which we evaluated through a controlled experiment is in Chapter 5. In Chapter 6 we describe related work. Finally in Chapter 7 our Conclusions and Future Work can be found.
Chapter 2

Background and Terminology

Service Oriented Architectures (SOAs) are, due to their dynamic nature, an especially useful computing paradigm for systems that continuously need to be online. In this thesis we will assess tracing techniques for Service Oriented Architectures (SOAs). In this chapter we will explain transaction traces and the aspects of a transaction trace. While tracing is especially useful in SOA due to its dynamic nature, the distributed aspect makes tracing radically different on SOA than on monolithic systems [17, 23, 30, 57]. Due to the dynamic nature of SOA, dynamic online analysis is preferred over static offline analysis. An advantage of dynamic analysis is that it presents a current overview of the system [10, 18]. In order to obtain such a current overview, we need to apply monitoring on SOA. Therefore we will briefly introduce monitoring. We propose a monitoring classification for services based on terms used in the literature. We propose this monitoring classification to differentiate monitoring approaches and we use this classification in our evaluation to discuss trade offs. Finally, common concepts of SOA are described and how to apply monitoring using these concepts.

2.1 Transaction traces

We defined three fundamental dimensions of monitoring, consisting of ‘services’, ‘users’ and ‘time’. Transaction traces cross-cut these dimensions. Gartner describes the term transaction as used by most APM (Application Performance Monitoring) vendors as: A single operation from the perspective of the user, even though it may consist of a number of operations from the perspective of the actual system implementation [10]. Transaction tracing attempts to trace the effects of this transaction across a sequence of components and network paths. See Figure 2.1 for an example transaction trace. The external user sends a request (REQ1) containing AIS data to the Raw Data Publisher. Upon reception of this message, The Raw Data publisher sends a request containing the AIS data (REQ2) to its subscribers, which in this case is the Messagetype service. The Messagetype service requests (REQ3) the Type Identifier Service to identify the message type. The Type Identifier Service responds (REQ4) the message type to the Messagetype service. In this example, the transaction trace path consists of 4 requests.
A general description of a trace is given by Kraft [46]: "Trace recording, or tracing, is a commonly used technique useful in debugging and performance analysis. Concretely, trace recording implies detection and storage of relevant events during run-time, for later offline analysis. [46]". Thus transaction tracing poses some requirements on the monitoring process. It defines what information is essential, which has implications for the monitoring architecture.

The biggest problem with transaction tracing is that operations may often change data (generally stored in a database) [10]. These changes are not traceable with pure service transaction tracing and these database changes may cause a permanent change in the state of an application (and thus initiate new events). Tracing database changes requires the whole database model to be changed. Every data operation needs to be logged, not only the adding of data but also the removal of data. Removal of data may also initiate actions. In this thesis we do not investigate the tracing of database changes. The implications of tracing database changes on the overhead are unknown, we therefore denote it as future work.

Relevant events. The relevant events for transaction traces are derived from the three Monitoring Dimensions. For each trace we need to store information about the user that initiated the trace, the services that are a part of the trace and the time of occurrence. This implies storing for each request the involved services and storing at what time the request happened. These requests need to be mapped to the corresponding transaction trace. Thus in order to log transaction traces, the following needs to be done:

- Log requests (From, To & Timestamp)
- Map the logged requests to their respective traces

Figure 2.2 contains two example transaction traces. In this example, all 8 requests (From, To & Timestamp) need to be logged. There are two resulting transaction trace paths: Blue (REQ1, REQ2, REQ5, REQ7) and Red (REQ3, REQ4, REQ6 and REQ8). In Chapter 3 we will introduce three fundamental online tracing techniques to extract these traces in Service Oriented Architectures.
We require transaction traces to crosscut the Monitoring Dimensions we identified. Due to the dynamic nature of the Maritime Safety and Security Systems domain, dynamic online analysis is preferred over static offline analysis. In order to obtain information dynamically, we need to monitor the system. Monitoring is the identification of the current state of a particular artifact. Bridges, dikes and buildings are all subject to monitoring. Since the 1990s all aircrafts are supplied with extensive monitoring systems [62]. Monitoring is especially useful in dynamic environments, to ascertain that the monitored artifact is in a correct state. This is why monitoring is a requirement for the class of systems such as the Maritime Safety and Security System. Monitoring is a well explored domain in computer science. It is applied in many different levels of abstraction, both in low-level hardware as well as high-level software. Monitoring is also used in many different domains, from Operating Systems to Web Services [20, 70]. Monitoring is especially vital in SOAs, which operate in a volatile setting. Services are generally intended to be up and running continuously for a longer period of time. Therefore runtime monitoring is valuable for services as they are deployed in an ever-changing environment. For a reasonable list of monitoring references see Chapter 6 which contains related work. The following two surveys provide a nice overview of the work on runtime monitoring in software engineering: A survey on self-healing systems: approaches and systems by Psaier and Dustdar published in 2011 [70] and A taxonomy and catalog of runtime software-fault monitoring tools by Delgado et al published in 2005 [20]. Monitors are used for a large number of different end-goals, such as:

**Performance analysis**

Application Performance Monitoring is the field especially interested in performance analysis [10]. Performance analysis may be used for several purposes such as bottleneck detection and software optimization [10, 20, 70, 76].

**Usage profiling**
2. BACKGROUND AND TERMINOLOGY

How are different software components related to each other? Which services are used by what users at what time? Questions cross-cutting these three dimensions (services, users, time) can be answered by applying monitoring techniques such as transaction tracing and entity usage [17, 70].

Software health
Software health monitoring may be used in combination with fault diagnosis and fault recovery to achieve a self-healing system [20, 70, 76].

Usage billing
This includes metering usage of bandwidth, computation time, etc for subsequent user billing [76, 51].

Service Level Agreements
When different organizations use each others services, they would like the services they depend on to exhibit reliable behavior. In these cases a set of usage and quality constraints are often defined in Service Level Agreements. Monitoring may be applied to verify that both parties are living up to the agreement [70, 76].

Security
In order to prevent unauthorized usage and block malicious intends, monitoring may be used to assess the nature of the interactions and react in case of misuse [55, 71].

Data mining
Monitoring may also be applied to log messages for audit or subsequent data mining [76].

Application events
Providers may provide a service, but often they would like to know how users use their service. For example, a bank offering a service to withdraw money, may want to know how many of these withdrawals are over 1000 euro [51, 76].

In this thesis we are monitoring to obtain the information derived from the Monitoring Dimensions we identified. Thus, for the end-goal Usage Profiling. However, in essence the information we monitor could also be used for the other monitoring goals as transaction traces are essential for many of these concerns.

2.3 Types of Monitoring

In order to evaluate our monitoring approaches in the case study, we will propose an extended classification for service monitoring types based on the definitions in literature. We will use this extended classification in our evaluation in Chapter 5 to assess the tracing techniques and discuss trade offs.

*Online Monitoring* is the term used for monitors that run in the same process (and thread) as the target application. The target application is the program subject to monitoring, in order to retrieve data from it. The monitor may also be performed *offline*, which
indicates that the monitoring code is run in a different process or thread (possibly on a separate machine). Delgado et al [20] coin the terms "offline (synchronous)" and "offline (asynchronous)" to further distinguish offline approaches. If there is any form of communication between the target code and the monitoring code, even though they are operating in a separate process or thread, the term "offline (synchronous)" is used. If the application has no interaction with the monitoring code whatsoever, it is "offline (asynchronous)". We will use this definition in the proposed extended definition of monitoring types for services.

Depending on the nature and purpose of the monitor, the monitor may require information (data) which is only available internally, this behavior is sometimes referred to as internal monitoring, or if no internal information (data) is required external monitoring [1, 5, 32, 84]. In this thesis, the terms internal monitoring and external monitoring will also be used to identify such behavior. We will use this definition in the proposed extended definition of monitoring types for services.

Intrusiveness refers to the degree of impact that a monitor has on the original behavior of the monitored system. It is an important property, because when a monitor consumes at least some resources of the target program (e.g. memory, CPU-time) to analyze its behavior, a monitor system may affect or alter that very behavior. This is referred to as the Heisenberg effect for software [77]. In the literature the terms intrusive and non-intrusive are commonly used to describe the impact a monitor has on the behavior of a system [3, 5, 12, 28, 31, 48, 49, 50, 60, 72, 77, 78, 83]. However, this definition is too broad, as non-intrusive allows no impact on the target system and intrusive allows any impact on the target system. There are definitely levels of degree to which an internal monitoring system is intrusive, therefore we propose an extended definition, which distinguishes approaches between non-intrusive, weak-intrusive and intrusive monitoring. Because we are especially interested in the monitoring of services, the proposed definition is specifically targeted at monitoring services. Since functionality is achieved through chaining services, changing messages sent between services is intrusive, because the functionality is also embedded in the communication layer. Therefore adapting the messages sent, even though they occur outside of any particular service, are also considered intrusive on the functionality.

Non-intrusive monitoring. Our definition of non-intrusive monitoring is that the monitoring units should not alter the functionality of the application at all. Thus it is not allowed to change the messages sent, nor is it allowed to access the internal data of a service. It is merely allowed to ‘listen in’ on communication, as this does not alter the functionality of the application. Thus for a monitoring technique to be considered as non-intrusive monitoring, it has to fulfill the following requirements:

- The monitoring is offline (asynchronous)
- The monitoring is external
- The monitoring process can be started and (unsafely) terminated without affecting the service monitored

The first requirement enforces independency of the monitoring application, if monitoring applications fulfill this requirement, they can be considered as decoupled from the applica-
tion itself. A monitoring application which is built in the normal application, can still be considered non-intrusive, as long as it does not require internal information (data) and runs in a separate thread. The third requirement is also important, for example: if the monitoring application crashes, it should not have an impact on the service. Normally, this is a direct consequence of the first and second requirement. Considering these limitations, we can conclude that non-intrusive monitoring systems have limited access to information regarding the traffic from and to a service. Basically, it can extract the destination and the receiver of a request, the payload and the time at which the request was sent/received. Therefore, not all monitoring applications are compatible with non-intrusive monitoring. In Chapter 3 we introduce tracing techniques, we will discuss whether these tracing techniques are compatible with non-intrusive monitoring. We have identified a number of non-intrusive approaches [43, 59, 72, 74, 75], in Chapter 6 related work is discussed.

**Weak-intrusive monitoring.** Weak-intrusive monitoring is the definition for monitoring systems that are intrusive, but in a very limited manner. We set up a number of requirements which allow monitoring applications to extend on the transport layer of the monitored application - to increase its capabilities - but still limit the risk and impact the monitoring system presents. This way, the functionality of a single service is not adapted, thus in that sense it is non-intrusive for a single service. However it is allowed to alter the functionality of a chain of services slightly, as it is allowed to extend the communication between services. We consider monitoring applications as weak-intrusive if the following requirements are met:

- The monitoring is offline (asynchronous)
- The monitoring process may alter the communication (semi-external), but does not change internal data of services

For non-intrusive monitoring the following requirement also holds: The monitoring process can be started and (unsafely) terminated without affecting the service monitored. Even though we do not add this to our requirements for weak-intrusive monitoring approaches, we do suggest implementations to take this into consideration. This is because it enforces that crashes of the monitoring application do not affect behavior of the target applications. For example, suppose messages are extended with additional information, if the monitoring application crashes at one of the services, it should not affect the functionality of the cluster. Thus, the service should still be able to understand extended messages (or messages should not be extended when send to a 'broken monitoring’ service). We have identified a number of weak-intrusive approaches [24, 31, 49, 56, 66, 85], in Chapter 6 related work is discussed.

**Intrusive monitoring.** Intrusive monitoring denotes any form of monitoring that does not apply to the restrictions set by non-intrusive monitoring or weak-intrusive monitoring. If implemented with care and using good programming standards, the actual infringement of the monitoring may be very marginal. For example, one may implement the monitoring process in such a way that it can be enabled and disabled without affecting the applica-
tion. We have identified a number of intrusive approaches \([1, 4, 42, 47, 80, 83, 84, 86]\), in Chapter 6 related work is discussed.

**Trade offs.** Non-intrusive monitoring implies no impact at runtime of the monitoring solution on the target application. In addition, non-intrusive monitoring does not require source code access of the service cluster. However, the data it can obtain is limited, this is why it is not compatible with all monitoring applications. Weak-intrusive monitoring implies limited impact and requires source code access of the communication layer, but not of the services. Intrusive monitoring incurs the most impact on the application. Internal monitors are intrusive. Internal monitors will always consume at least some resources of the target program (e.g. memory, CPU-time) when analyzing the system. By consuming these resources, the monitor may affect or alter the behavior of the target system. This is referred to as the Heisenberg effect for software \([77]\). That is a downside of internal monitoring. In addition, complete source code access is required. However, Internal monitors can gather and accumulate much more information about the internal state of a program which is invisible or unreachable from outside a program.

## 2.4 SOA Concepts

In this section we introduce concepts of SOA which are relevant for service monitoring and tracing. First we will describe the different types of communication in SOA on which we will evaluate the tracing techniques. We will introduce the following components: services, registry and the Enterprise Service Bus. In Section 2.5 we describe how these components can be used for monitoring.

**Messaging.** Since every service operates in a self-contained manner and is only accessible over a prescribed interface, the messaging system has a large impact on the overall performance of a SOA. A SOA may employ synchronous and/or asynchronous message systems. If high transaction volumes are expected, asynchronous messaging should be used. This greatly increases the runtime performance \([76]\). Another advantage of asynchronous messaging is that it further decouples the applications, as the service consumer does not have to wait for the reply of the provider, thus reducing the dependency of the consumer on the provider. In addition, asynchronous messaging systems are able to ensure delivery of a package, in case of a failure it can just resend the package. This is especially useful for mobile environments, where connectivity may be an issue. On the other hand there are also some disadvantages to asynchronous messaging. First, asynchronous messaging makes the communication more complex, as messages and replies have to be related and may arrive out of order. On top of that, asynchronous messaging implies a certain amount of overhead and bookkeeping to simulate synchronous interactions between two applications \([6, 76]\).

The messaging model has a large impact on tracing. On service clusters with synchronous communication it is a lot easier to perform tracing as there is only a single request handled by a service at a time, whereas multiple requests are handled simultaneously by a service in
the case of asynchronous communication. As described asynchronous messaging is more scalable and therefore it is important that our implementation of the tracing techniques also support service clusters with asynchronous communication.

**Services.** As Service Oriented Architectures are deployed in many contexts, large independent organizations, such as The Open Group, OASIS and The World Wide Web Consortium (W3C), have tried to define SOA in collaboration with business partners and/or user communities to standardize concepts and terminology [25, 64, 65].

OASIS SOA Model defines a service as: "A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [64].

The Open Group defines a service as: A service is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports). A service is self-contained and may be composed of other services. The service is a black box to its consumers.

A service may depend on other services to fulfill its purpose. This creates complex dependencies, as a complex service could depend on services that again depend on other services. Service Oriented Architectures are often operating in a highly dynamic environment. New services may emerge, old services may retire and the specifications of a particular service may change.

**Registry.** The SOA registry is one of the very important components of SOA [76]. As stated before, one of the main goals of SOA is to increase re-usability. In order to be able to find a piece of functionality, a registry which is able to locate the service, should be present. Also, this achieves further decoupling between the consumer and provider, as a provider may be replaced, without the consumer even being aware. Decoupling of the provider and consumer is also a goal of SOA [76].

**Enterprise Service Bus.** Large SOAs may be equipped with an Enterprise Service Bus (ESB) to connect the services. The ESB takes care of the routing between service providers and consumers. It further decouples the providers and consumers as they are unaware of each other and not directly linked. Furthermore, it reduces the connectivity problem of a cluster of services. For N services \((N - 1)/2\) connections are needed to connect them all to each other. With one ESB, only N connections are required [6, 76]. Figure 2.3 displays the difference between a connected cluster without esb and a connected cluster when using an ESB. Besides supplying routing support and providing scalable connectivity, an ESB is also able to perform protocol transformations and data/message transformation. This way, services are not required to use the same protocol or message format. Monitoring may be implemented on the ESB, as all traffic passes through the ESB. The advantage is that the monitoring only needs to be implemented in one place, the ESB. However, in the ESB the monitors do not have access to the internal state of the services.
2.5 Monitoring applied on SOA

The relevant events for transaction traces require each request to be logged. The proxy pattern [27] is a design pattern suited for this concern. A proxy redirects all incoming calls to the actual 'target' object, it may apply a transformation execute code for each call. By passing each request through a monitoring proxy, which redirects it to the actual service, all requests are logged. We think that there are three commonly used components in SOA [6, 76] which are especially suited for the responsibility of deploying the monitoring proxies:

- Registry
- ESB
- Services

**Registry.** By giving the responsibility to deploy monitoring proxies to the registry, the overhead of deploying monitoring proxies to existing services is minimal. The registry simply receives a new responsibility: Spawning a monitoring proxy whenever a service requests access to another service. Instead of returning the address of a service, the address of a newly spawned monitoring proxy which redirects to that service is returned. The advantage of this method - in contrast to extending the services to spawn a proxy - is that not every
service needs to be extended with the ability to spawn a monitoring proxy. In fact, no source code access of the services is required at all, thus it can easily be applied on large service oriented architectures with few adjustments. Failure of a monitoring proxy can also be handled by the service registry. By probing the monitoring proxy with heartbeats, the service registry can manage monitoring proxies and re-spawn a monitoring proxy, if needed. Also, the monitoring mechanism can easily be turned off in the service registry, which can return the service address again instead of a monitoring proxy based on a configuration variable. A disadvantage of spawning monitoring proxies at the registry is that there is no access to the transport layer or the internal state of a service, therefore only non-intrusive monitoring is supported for monitoring proxies spawned at the registry. In addition, not all Service Oriented Architectures work with a registry, thus in such a case, every service needs to be adapted to start using a registry.

**Enterprise Service Bus.** The Enterprise Service Bus (ESB) is a suitable component to add the responsibility of deploying monitoring proxies to. Because the ESB is responsible for the routing of requests, each request passes through the ESB. Like the registry, when an existing service cluster needs to be extended with monitoring proxies, it requires not much effort. Only the ESB needs to be adapted. The ESB cannot provide internal access to the service, however it does allow monitoring of incoming and outgoing requests, as all communication goes through the ESB. Since the ESB represents the transport layer, both non-intrusive and weak-intrusive monitoring approaches are supported by monitoring proxies deployed at the ESB.

**Service.** An advantage of having the service component responsible for spawning monitoring proxies is that monitoring proxies have the ability to get internal information of services, thus it allows intrusive monitoring. A disadvantage of having the service component responsible for spawning monitoring proxies is that every running service needs to be adapted to spawn monitoring proxies. This can easily be achieved by extending all services with a super class that implements the monitoring proxy responsibility. However, if the monitoring needs to be added to an existing service cluster, all existing services have to be replaced.

**Monitoring applied on SOA.** In section 2.3 we described a classification for monitoring types. In this section we used that classification to discuss the type of monitoring supported in three commonly used components in SOA [6, 76]: (1) the registry; (2) the ESB; and finally (3) services. Table 2.1 summarizes the types of monitoring that are possible when deploying monitoring proxies at the corresponding component. As shown, if intrusive monitoring is required, monitoring proxies should be spawn at the services. If intrusive monitoring is not required and monitoring techniques have to be incorporated in an existing service cluster, it is better to deploy monitoring proxies from the ESB or registry. If monitoring proxies have to be deployed at the services in an existing service cluster, many existing components have to be adapted. Therefore the effort is considered higher than deploying at the ESB or registry, where only one component needs to be adapted.
Table 2.1: This table displays the implications of deploying monitoring proxies in a particular SOA component

<table>
<thead>
<tr>
<th>Component</th>
<th>Non-intrusive monitoring</th>
<th>Weak-intrusive monitoring</th>
<th>Intrusive monitoring</th>
<th>Effort (new cluster)</th>
<th>Effort (existing cluster)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registry</td>
<td>Possible</td>
<td>Hard</td>
<td>Impossible</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>ESB</td>
<td>Possible</td>
<td>Possible</td>
<td>Impossible</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Services</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>
Chapter 3

Tracing techniques for services

The relevant events for transaction traces are derived from the three Monitoring Dimensions we identified, consisting of ’services’, ’users’ and ’time’. For each trace we need to store information about the user that initiated the trace, the services that are a part of the trace and the time of occurrence. This implies storing for each request the involved services and storing at what time the request happened. These requests need to be mapped to the corresponding transaction trace. Figure 3.1 contains two example transaction traces. In this example, all 8 requests (From, To & Timestamp) need to be logged. There are two resulting transaction trace paths: Blue (REQ1, REQ2, REQ5, REQ7) and Red (REQ3, REQ4, REQ6 and REQ8). Through a literature study we identified that there are traces are obtained through online dynamic analysis and through offline static analysis. Because of the dynamic nature of the class of systems we are interested in, we are interested in online dynamic analysis [57]. We identified three fundamental online tracing techniques from which all tracing techniques can be derived:

- Tagged Tracing [10, 14, 23]
- Timestamped Tracing [10, 22]

Figure 3.1: Two traces (Red, Blue)
• Recognized Tracing [10]

The names for these techniques are introduced by us as there is no common terminology for these approaches. The names were inspired by the work of Gartner [10] which refers to Tagged Tracing as "Dope and Trace", Timestamped Tracing as "Timestamp analysis" and Recognized Tracing as "Recognize and Trace". We will evaluate the tracing techniques through a controlled experiment, the implementation is described in Chapter 4, the evaluation in Chapter 5.

An issue with transaction tracing is that operations may often change data (generally stored in a database) [10]. These changes are not traceable with pure service transaction tracing and these database changes may cause a permanent change in the state of an application (and thus initiate new events). Tracing database changes requires the whole database model to be changed. Every data operation needs to be logged, not only the adding of data but also updates and removal of data. As updates and removal of data may also initiate actions. In this thesis we do not investigate the tracing of database changes. The implications of tracing database changes on the overhead are unknown, we therefore denote it as future work.

3.1 Tagged Tracing

In order to correctly correlate events to a certain transaction, requests and computational steps (to identify consequent requests caused by a particular request) need to be related. The idea is to relate these requests and computational steps to their trace by using a correlation identifier (trace ID). This trace ID will be added to each request, we refer to this concept as 'tagging'. We refer to requests with a trace ID as tagged requests. This functionality may be achieved by adapting both the communication layer and the internal operation of services. The communication layer needs to be adjusted to allow messages with a trace ID. In addition, services need to relate subsequent requests to the initial request, thus a trace ID needs to be stored during computation steps. Figure 3.2 displays an example trace with the tagging concept in place. Every request gets tagged with an identifier, relating it to a particular trace. The advantage of Tagged Tracing is that the expected accuracy of the reconstructed traces is very high because the correlation of requests is done at runtime.

The Tagging approach also has some disadvantages. The adjusting and extending of the original operation implies that the original functionality could break. In addition, it is an extra development effort compared to simply using the already available information. Also, often user operations result in a database change [10], which may initiate new events. These effects are not traceable with the tagging approach, unless the entire database model is adapted. This is complex and generates a lot of overhead. In addition, if the service is multi-threaded, it becomes more difficult to correctly store trace IDs.

Multi-threaded Tagged Tracing. Recently, multi-threading has become increasingly important, because of the recent trend of parallel computing and the stagnating clock rates [68]. This emphasizes the need to facilitate multi-threaded tracing of requests. Tagged tracing presents some challenges in the case of a multi-threaded system, shared resources need to
be tagged such that events that cross thread boundaries through shared resources are traceable. Therefore we added support for multi-threaded tracing in our implementation. There is no other approach described in the literature which implements this behavior, to the best of our knowledge. There exist approaches that allow multi-threaded tracing, but these are focused on tracing the events per thread and correctly linking events to their respective thread. They do not cross-link events in multiple threads to each other [2, 26, 29, 69, 81].

We came up with a solution for threads that are created inside the thread in case of a certain event. By extending the Thread Class and adding a parameter to the constructor of the Thread, trace IDs may be passed into threads that are the result of an event (the request). This is more extensively described in Chapter 4. However, if there are background worker-threads which use shared resources, it becomes harder to track direct events of a certain request. For example, one thread may receive data from a request and store this data. A back-ground worker-thread may access this data and react accordingly - based on the data. If it is the intend to track this sort of behavior, every shared data resource should provide an interface that can be tagged. We implement a Queue supporting tags, such that background threads correctly set trace IDs upon popping data from the Queue. The TaggedQueue extends the behavior of the push and pop methods. Figure 3.3 shows the behavior of the TaggedQueue on a push. The push method automatically adds the trace ID of the current TaggedThread to the data and pushes the tagged data on the queue. When the pop method is called in a thread, the trace ID automatically gets removed from the tagged data and assigned to the thread that popped the data. This is displayed in Figure 3.4. This way, background threads correctly set trace IDs based on data shared through the TaggedQueue. We used the Ruby Benchmark suite to determine the overhead incurred by the added functionality on Queues. In our AIS case study, which is described in Chapter 4, the overall overhead of the TaggedQueue is only 6.5%.
3. TRACING TECHNIQUES FOR SERVICES

Figure 3.3: Pushing data to the TaggedQueue

Figure 3.4: Popping data of the TaggedQueue
Figure 3.5: Service monitoring through the proxy pattern

3.2 Timestamped Tracing

The requests sent and/or received by a service can be recorded without adjusting or extending the operation. We only need to listen in on the communication of each service. This will provide information on requests coming and going to a certain service. The proxy pattern [27] is a design pattern suited for this concern. A proxy redirects all incoming calls to the actual 'target' object, it may apply a transformation execute code for each call. By passing each request through a monitoring proxy, which redirects it to the actual service, all requests are logged. Figure 3.5 displays the proxy pattern.

If we assume synchronized clocks (for simplicity), it is possible to put the requests in a chronological order. The idea is to relate requests on their time of occurrence, based on the assumption that consequent requests are likely to happen shortly after the initial request. However, service oriented architectures are often set up to facilitate multiple requesters at the same time (asynchronous communication or load balancing using multiple services) [6, 21, 40, 76]. This implies that requests coming from service A, in between a receive and reply to service B, do not necessarily have to be initiated by service B. They may as well be initiated by service C, to whom A was also providing a service at that particular time. This complicates the correlating of requests through time stamp analysis.

3.3 Recognized Tracing

Just like for Timestamped tracing, the requests sent and/or received by a service can be recorded without adjusting or extending the operation. Therefore, the proxy pattern displayed in Figure 3.5 is also a valid option for Recognized Tracing. In addition, of each request the payload needs to be logged. Recognized Tracing is an approach described by Gartner in their survey of APM approaches [10]. The requests that are recorded are analyzed
3. **Tracing Techniques for Services**

and correlated based on the recognition of package headers and/or package payload features that remain constant or evolve in a predictable way. Because several sub-traces of an entire transaction path may arise, a 'stitching engine' is used which basically is an algorithm that couples multiple of such sub-traces to reestablish the original transaction trace. This method requires knowledge of the operation of the services and thus cannot be implemented independently of the service developers. In addition, the fact that the implementation requires knowledge of the operation of the specific cluster implies that it cannot (easily) be reused for another service cluster. However, this implementation can be very efficient if the transport layer of the service cluster already adds tags to the requests to maintain a context or simulate synchronous behavior on an asynchronous architecture.

3.4 **Summary**

Three fundamental online tracing techniques were introduced: Tagged Tracing [10, 14, 23], Timestamped Tracing [10, 22] and Recognized Tracing [10]. We expect Tagged Tracing to be able to reconstruct nearly all traces, however probably it will have the most overhead. Timestamped racing on the other hand, will probably have the least overhead. We do not expect that Timestamped Tracing performs well on asynchronous communication, as multiple traces happen at the same time. In Chapter 4 we describe how we implemented these tracing techniques. Chapter 5 the tracing techniques will be evaluated based on a controlled experiment.
Chapter 4

Implementation

4.1 Case Study System

Through a literature study we identified the following three fundamental online tracing techniques; Tagged Tracing, Timestamped Tracing and Recognized Tracing. There are a number of approaches that use online tracing techniques to extract traces [10, 14, 23, 22]. However, to the best of our knowledge, there is hardly any literature on the evaluation of tracing approaches on SOA nor articles that compare the tracing approaches to each other. Therefore, we devised an evaluation setup to validate and compare the three fundamental online tracing techniques through a controlled experiment on criteria such as overhead, accuracy and effort. The evaluation and an overview of the advantages and disadvantages of each tracing technique are described in Chapter 5. In this chapter we will describe our implementation of the tracing techniques and the service cluster, which are required for an evaluation.

We have built a monitoring implementation on a cluster of services which are operating on Automatic Identification System (AIS, [53, 63]) data. AIS is a system which allows ships to communicate their positions and status. There is a continuous stream of messages as each ship is required to send updates to help ships avoid collisions, as well as assisting port authorities to better control sea traffic [53, 63]. The Raw Data Publisher Service publishes the AIS messages to its subscribers. The Raw Data Publisher is implemented following the publish-subscribe pattern, which is a messaging model often used in SOA [13] [76]. In this model, whenever a service wants to send a message, it publishes the message it wants to send by placing it on a queue. Whenever a message is added to the queue, it is sent to all the subscribers. If there are no subscribers, the message is disregarded. The publishing of messages is completely decoupled of the reception by subscribers. There are (currently) two services subscribed to the Raw Data Publisher: The Messagetype service and the Navigational Status Service. The Messagetype Service allows services to subscribe to a single or range of message IDs. For example, if a service subscribes to message ID ’1’, it receives all AIS message with message ID ’1’ that the Messagetype service receives from the Raw Data Publisher. The Messagetype service uses the Type Identifier service to obtain the message ID from the message. The message ID is in a 6-bit field, of which 27 of the 64
4. IMPLEMENTATION

Figure 4.1: AIS service cluster

are currently used [38].

The Navigational Status Service uses the Type Identifier Service to obtain the message ID. Message ID one, two and three are Class A position reports. Class A position reports contain information on the position, speed, direction, type and navigational status of the ship. The Navigational Status Service uses the Payload Extractor to extract the payload of messages. This information is encoded in a 6-bit representation and should be decoded to a 8-bits representation in order to use it. The Navigational Status Service uses the Decoder Service to decode the messages from a 6 to 8 bit representation. The Navigational Status Service allows services to subscribe to a certain Navigational Status. These subscribers will receive all AIS messages with the corresponding Navigational Status. For example, if a service is subscribed to the ‘anchored’ navigational status, it will receive all AIS messages of ships that are anchored. The navigational status is defined in a 4 bit field, currently 10 of the 16 possible values are used [38].

Figure 4.1 displays the service cluster as described. Currently, there are services subscribed to the Messagetype Service for each type and to the Navigational Status Service for each status. These services perform no advanced functionality (yet). Expanding on them, for example to display the sea traffic in a specific region would be a potential extension of the experimental service cluster. However, we consider the case study system extensive enough to evaluate and compare the tracing techniques.
4.2 Monitoring architecture

The communication between services is implemented using Distributed Ruby (DRb). DRb is a Remote Method Invocation (RMI) implementation for Ruby [37]. We used Ruby, because it is a language especially useful for testing concepts. We did not use an existing SOA platform, because it is too verbose to test concepts and we would lose controllability over the execution. In order to allow services to communicate with each other, a registry is implemented which holds a Hash from service name to service address. Figure 4.2 illustrates how the Navigational Status Service may contact the Decoder Service using the registry. The communication works through three simple steps: (1) the Decoder Service registers itself; (2) the Navigational Status Service requests the registry for the address of the Decoder Service; and finally (3) the Navigational Service binds this address and uses it. This is much alike how the UDDI registry works in web services [76].

The registry source code to register a service is displayed in Listing A.1 in Appendix A. The registry generates a unique ID for each service. The maximum amount of services allowed is $2^{12} = 4096$ services. The identifier is shifted 52 bits to the left, such that every service may use that number as their start point for message sequence numbers. This way, unique sequence numbers in the service cluster are guaranteed. The unique sequence number is used for tagging traces with a unique trace ID as described later in this section. The sequence number overflows if it reaches a number bigger than $2^{52}$. The amount of bits used for services and sequence number may be amended to the amount of services expected and/or the type of traffic.

As described in Chapter 2 there are three components which are suitable for deploying monitoring proxies. Table 4.1 summarizes the capabilities of the three options, for more
4. IMPLEMENTATION

Table 4.1: This table displays the implications of deploying monitoring proxies in a particular SOA component

<table>
<thead>
<tr>
<th>Component</th>
<th>Non-intrusive monitoring</th>
<th>Weak-intrusive monitoring</th>
<th>Intrusive monitoring</th>
<th>Effort (new cluster)</th>
<th>Effort (existing cluster)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registry</td>
<td>Possible</td>
<td>Hard</td>
<td>Impossible</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>ESB</td>
<td>Possible</td>
<td>Possible</td>
<td>Impossible</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Services</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

information see Chapter 2. In synchronous communication data and thread tagging is not necessary, therefore Tagged Tracing is implementable on a synchronous service cluster with weak-intrusive monitoring techniques. However, in asynchronous service clusters threads need to be tagged as well as shared data resources, therefore Tagged Tracing is only implementable with intrusive internal monitoring techniques. Because we want our monitoring framework to deal with all scenarios, the services are responsible for deploying monitoring proxies in our implementation. Each service should extend the MonitorableService class. Figure 4.3 shows the class diagram of the classes relevant for monitoring. The MonitorableService returns a FrontProxy, whenever the get_service function is used, such that all outgoing communication is monitored by a FrontProxy. The BackProxy is only used if a service is in the tagged monitoring configuration. The ReportAgent is responsible for sending the monitored information to the centralized analyzer. The MonitorableService class allows the following three monitoring configurations:

**No monitoring** The incoming and outcoming communication goes through the DRbObject of the service.

**Timestamped monitoring** The incoming communication goes through the DRbObject of the service. The outgoing communication is monitored through a FrontProxy. For each Service a different FrontProxy is created. For example, if the Navigational Status Service sends a request to the Decoder Service and to the Payload Extractor, the traffic goes through two different proxies. This is illustrated in Figure 4.4. This monitoring configuration is also viable for Recognized Tracing, with a configuration variable, payload logging may be toggled on and off.

**Tagged monitoring** The incoming communication passes through a BackProxy. The outgoing communication is monitored through a FrontProxy. Figure 4.5 illustrates the
employed two-sided communication monitoring.

The FrontProxy is responsible for monitoring outgoing communication. Listing A.2 in Appendix A shows how FrontProxy.rb implements this behavior. It makes use of Ruby language concepts such as 'the method_missing method. The method_missing method automatically gets called by the Ruby interpreter, if the method called on an object is not present. This way, the FrontProxy is able to act as a proxy for any service without requiring knowledge of the methods of that service, as all method calls will be redirected to method_missing.

**Timestamped monitoring.** Timestamped monitoring logs the requests from services without adapting the functionality of the services themselves. This monitoring approach is sufficient for Timestamped Tracing and Recognized Tracing. Payload logging, which is required for Recognized Tracing, may be toggled on and off through a configuration variable. The FrontProxy is responsible for monitoring outgoing communication. Listing A.2 in Appendix A shows how FrontProxy.rb implements this behavior. It makes use of Ruby language concepts such as 'the method_missing method. The method_missing method automatically gets called by the Ruby interpreter, if the method called on an object is not present. This way, the FrontProxy is able to act as a proxy for any service without requiring knowledge of the methods of that service, as all method calls will be redirected to method_missing. The FrontProxy does not tag messages in the timestamped monitoring configuration. Therefore it performs regular calls to other services, without extending the message. However as our implementation is ready for hybrid configurations of both tagged monitored services
and timestamped monitored Services, a special routine call is performed in case the other service is running a BackProxy. This way, the BackProxy does not incorrectly classify the call as a call coming from an external user.

**Tagged monitoring.** The FrontProxy is responsible for monitoring outgoing communication. Listing A.2 in Appendix A shows how FrontProxy.rb implements this behavior. It makes use of Ruby language concepts such as 'the method_missing method. The method_missing method automatically gets called by the Ruby interpreter, if the method called on an object is not present. This way, the FrontProxy is able to act as a proxy for any service without requiring knowledge of the methods of that service, as all method calls will be redirected to method_missing. If the service is running in the tagged monitoring
configuration, the trace ID of the current thread is sent along with the request and logged in the report. If there is no trace ID set, that means that this call initiates from this service. In that case, the local counter will be used to generate a new unique trace ID. The local counter interval is obtained from the service registry when the service registers, this ensures that the trace ID is unique throughout the entire service cluster. After logging the call, it is forwarded to the actual service. If this FrontProxy is tagging messages, and the service responds to the \_taggedcall\_ method, the trace ID is sent along. This way, if the proxy broke down on the other side, communication does not break down but continuous without tagging. If the FrontProxy is not tagging messages, a regular call is performed. However, if the service responds to the \_timedcall\_ method, that method is called with the regular parameters, such that the proxy on the other side can differ it from an external call.

The BackProxy is responsible for monitoring incoming communication. Listing A.3 in Appendix A shows how BackProxy.rb implements this behavior. The BackProxy is only used if the service is running in the tagged monitoring configuration. In this configuration, whenever a request arrives, consequent actions should be bound to a trace ID. There are two possibilities, (1) a trace ID was sent along with the request, this trace ID is used to bind consequent actions; or (2) no trace ID is sent along, therefore a trace ID is generated by using the local counter of the service, which guarantees a unique trace ID. As visible in Listing A.3, the trace ID is set to Thread.current[\:trace]. This is a thread-variable, which helps identifying consequent actions of a particular trace in a multi-threaded environment. Since new threads may be started as a result of a request, we also implemented a new class which extends the normal Ruby Thread class: TaggedThread. The constructor of TaggedThread requires a trace ID, which is immediately set as thread-variable. A helper factory method which automatically uses the current thread-variable helps ensuring that whenever a thread is created that the trace ID is also set in a thread variable for the new thread.

In our experiment many services are Subscribable, for example Raw Data Publisher publishes the raw AIS stream to whoever is interested. The subscribe functionality is implemented through a module called ‘Subscribable’. This allows services to implement the subscribe functionality through a simple include of the Subscribable module. Listing 4.1 shows the each\_subscriber method of the Subscribable module, which other services may use to transmit their message to their subscribers. Each subscriber is dealt with in a separate thread. The each\_subscriber method makes use of the startThread functionality of TaggedThreading, to automatically set the thread-variable with the trace ID in the newly generated threads.

A common messaging model used in SOA is the publish-subscribe pattern [13] [76]. In this model, whenever a service wants to send a message, it publishes the message it wants to send by placing it on a queue. Whenever a message is added to the queue, it is sent to all the subscribers. If there are no subscribers, the message is disregarded. The publishing of messages is completely decoupled of the reception by subscribers. Because of this decoupling, a common implementation involves two threads. One to receive publications and one which distributes new messages to its subscribers. Since this implementation uses two threads a thread-variable is not able to link resulting calls to a publication. Therefore, the data has to be tagged. In Listing 4.2 a Publish-Subscribe implementation is shown. We used our own
4. IMPLEMENTATION

Listing 4.1: Method each_subscriber of Subscribable (Simplified)

```ruby
# usage with block
# e.g. each_subscriber { |sub| sub.receive(data)}

def each_subscriber
  @subscribers.each { |address|
    TaggedThreading::startThread { yield get_service(address) }
  }
end
```

Figure 4.6: The monitoring proxies and analyzer

Publish-Subscribe implementation in order to have full controllability over the execution. Data is pushed on the queue and tagged with the trace ID. The pop method of the Ruby Queue is a blocking method, therefore the background thread is always running. Whenever new data is pushed, the background thread will first retrieve the tag from the data. The background thread gets the new trace ID and sets it as thread-variable. The tag is cut from the data and send to the sub method, which in turn uses the each_subscribers function (see Listing 4.1) of the Subscribable module to transmit the message to all subscribers.

4.3 Information Processing

The implementation allows the users to adjust the monitoring information to their needs. Trace IDs and payload are both optional, this way, the user can determine how much information he wants at what cost. (Better information, both quality and quantity wise, gen-
Listing 4.2: Publish-Subscribe implementation

class PubSub < MonitorableService::Service
  include Subscribable

  def initialize
    @queue = Queue.new
    @background_thread = Thread.new { 
      while (true) do
        data = @queue.pop
        trace_id = data.split(" ", -1)[-1]
        Thread.current[:trace] = trace_id
        sub(data.slice!(0...-(1 + trace_id.length)))
      end
    end
  end

  def receive(data)
    @queue << (data << "#{Thread.current[:trace]}")
    return true;
  end

  def sub(data)
    each_subscriber { |sub| sub.receive(data) }
  end
end

erally implies more overhead). Each monitor collects and records information. In order to trace transactions, the information logged by multiple monitors need to be combined. Therefore, all the monitored information is gathered in a central place. All the gathered information is send to an analyzer which combines and analyzes all the data. The analyzer is basically a monitoring service. Figure 4.6 shows the architecture of the monitoring and analyzer aspects. The analyzer obtains its information from the monitoring proxies through reports. The analyzer is implemented in Java, because Java has better GUI support than Ruby. The monitoring aspect is implemented in Ruby, because it allows fast development of distributed systems. The communication between monitoring proxies (implemented in Ruby) and the analyzer (implemented in Java) is achieved through Apache’s Thrift protocol [79], which is a high performance protocol with a code generation engine to build services that work efficiently and seamlessly between a large number of programming languages [79]. The ReporterAgent (See Listing A.4 in Appendix A) is responsible for sending the data accumulated by the proxies to the analyzer. It has two configurations, instant reporting and time-interval reporting. Time-interval reporting optionally allows a maximum number of requests parameter to be set (@max_req). In case the maximum number of requests is
reached, an instantaneous report is triggered. It should be noted that, while instantaneous reporting allows the fastest reaction time to a certain event, it is also a very expensive method in terms of overhead. The overhead in terms of messages sent would be at least 100% on the number of requests, as each request gets reported. In the evaluation chapter we will evaluate the different reporting techniques with respect to the latency and overhead of the results.

Tracing. Traces are constructed using the trace IDs or reconstructed by the time stamp algorithm if trace IDs are not available. Traces may be represented as a tree. They have a request which marks the start of the transaction (start-request), a number of requests that follow as a result of the start-request (children), optionally followed by an answer to the original start-request (end-request). Since the same paradigm applies to the children of the start-request, these may themselves be considered a TraceTree as well. The order of the requests is the trace path of the TraceTree. Where a TraceTree refers to a particular transaction in time, the TraceTreeClass refers to a collection of transactions with the same trace path. Figure 4.7 displays the relations between TraceTreeClass, TraceTree and Request in an UML diagram.

![Figure 4.7: Class diagram of TraceTreeClass, TraceTree and Request](image)

A TraceTree always consists of a start-request. A request may be accepted as the end-request of a start-request if it fulfills the following requirements:

- End occurs after start (end.ts > start.ts)
- The sender of end is the receiver of start (end.from == start.to)
- The receiver of end is the sender of start (end.to == start.from)

A request may be accepted as a child if it if it fulfills the following requirements:

- Child occurs after start (child.ts > start.ts)
- The sender of end is the receiver of start (child.from == start.to)
- The receiver of end is not the sender of start (child.to != start.from)
The following example shows the construction of a TraceTree:

Consider the following events in our AIS experimental setup: The Raw Data Publisher receives a new AIS message from an external source (REQ0). It then sends a request with the AIS message (REQ1) to the Navigational Status Service, which has subscribed to the Raw Data Publisher. Following, the Navigational Status Service sends a request (REQ2) to the Type Identifier Service to identify the type AIS message that it is dealing with. This request occurs after REQ1 and starts from an allowed start point (REQ2.from == REQ1.to). However, it does not match the 3th requirement (REQ2.to == REQ1.from) thus it cannot be matched as an endrequest of REQ1. However, it does fulfill all requirements to be accepted as a child of REQ1 and is thus added to the children of REQ1. The Type ID Service replies (REQ3) that the AIS message is of type 3. This request does not fulfill the requirements to be accepted as a child or endpoint of REQ1 (REQ3.from ≠ REQ1.to). REQ3 does fulfill the requirements to become the end of REQ2 and thus becomes the end of REQ2.

Messages of type 3 contain Navigational Status Information [63]. The Navigational Status Service now sends a request (REQ4) to the Payload Extractor Service to extract the payload from the AIS message. The Payload Extractor Service extracts the payload from the AIS message and returns the result (REQ5) to the Navigational Status Service. The Navigational Status Service requires the 4 bits starting at the 38th bit [63]. Since the data is encoded in a 6 bits format [53], the Navigational Status Service requests (REQ6) the Decoder Service to convert part of the payload (the 38th to 42th bit) into an 8-bit format. The Decoder Service replies (REQ7) that the decoded number is '5', to the Navigational Status. The Navigational Service sends (REQ8) the AIS message to the Moored Service (as '5' indicates moored).

The TraceTree would now look like:

```
* (External) → Raw Data Publisher (REQ0)
  * Raw Data Publisher → Navigational Status Service (REQ1)
    * Navigational Status Service → Type Identifier Service (REQ2)
      Navigational Status Service ← Type Identifier Service (REQ3)
    * Navigational Status Service → Payload Extractor Service (REQ4)
      Navigational Status Service ← Payload Extractor Service (REQ5)
    * Navigational Status Service → Decoder Service (REQ6)
      Navigational Status Service ← Decoder Service (REQ7)
    * Navigational Status Service → Moored Service (REQ8)
```

Listing 4.3 shows the matching implementation (which is implemented in Java). It implements the rules we defined earlier. First it places the events in order, as only events that occur later can be matched as child or endpoint. The to method checks whether the condition the first request.to equals the second request.from. E.g. to(REQ2, REQ3) evaluates to true, because REQ2.to = Type Identifier Service and REQ3.from = Type Identifier Service. In case to evaluates to false, the request that occurs later performs the matching algorithm on the children of the request that occurs earlier. When requests fulfill the initial requirements, the actual success of the matching depends on whether the request is already matched and if
so, if the new matching is considered as a better matching. The challenge is in determining which match is better. In general, this will be less problematic in Tagged Tracing, as only other requests from the trace might challenge another trace for a match. In Timestamped Tracing, many requests (which in practice belong to different traces) may challenge another trace for a match. We came up with the following possibilities: (a) first come first serve; (b) minimizing on time between timestamps; (c) traffic-pattern matching. Neither of these options is perfect and all have their drawbacks. First come first serve is the easiest approach, as once a trace has a parent or an end it simply does not accept another. This approach works well for Tagged Tracing, as the matching is chronological and only requests from the trace itself may challenge a match. Minimizing on time between timestamps means that whichever request is closer in time is preferred as match. However, this may not necessarily be the best match as there is also computation time. Traffic-pattern matching means that requests prefer matches that are more likely and/or reject matches that are not allowed. For example, if there are no loops in the service cluster, the Timestamped Tracing method should disallow loops and/or prefer matches that do not involve loops.

Listing 4.3: Matching code

```java
public MatchResult match(SingleTraceTree t1, SingleTraceTree t2) {
    if (startsBefore(t1, t2)) {
        // t1 startsBefore t2: possibly an end or offspring of t1.
        if (to(t1, t2)) {
            if (to(t2, t1)) {
                return startMatchEnd(t1, t2);
            } else {
                return parentMatchKid(t1, t2);
            }
        }
    } else if (startsBefore(t2, t1)) {
        // t2 startsBefore t1: possibly an end or offspring of t2.
        if (to(t2, t1)) {
            if (to(t1, t2)) {
                return startMatchEnd(t2, t1);
            } else {
                return parentMatchKid(t2, t1);
            }
        }
    }
    return matchOnChildren(t2, t1);
}
```

// unmatched falls through
return MATCHFAILED;
```
4.4 Displaying Information

An important advantage of dynamic analysis is that it presents a current overview of the system [10, 18], as it logs the current use of the system. However, this implies a high volume of data [10] which is continuously increasing. In a recent work [18] a lot of approaches were identified that investigate the reduction of this information and visualization techniques to reduce the cognitive overload for the users. We designed a prototype dashboard to provide insight in the monitored information. As described in the introduction, Maritime Safety and Security Systems represent the class of systems that we target with this thesis. This class of systems are required to be continuously online, as downtime of the system has immediate consequences on the safety and security of Maritime Traffic. As requirements evolve over time, the system has to be maintained and updated accordingly. The dashboard aids the maintainer and developer by providing a dynamic overview of the system. Furthermore, the dashboard allows data to be stored to disk and loaded from disk, such that dynamic information can easily be reviewed and compared. This was an essential part for the controlled experiment we conducted, as the reconstructed traces had to be checked for correctness.

The main window of the dashboard has four tabs, the status tab, requests tab, traces tab and finally the overview tab.

**Status tab.** The first tab shows the status of the service cluster, which services are currently up and running and how much traffic is directed to - and originates from a certain service. In figure 4.8 the dashboard status tab is shown.

**Requests tab.** In figure 4.9 the requests tab is shown. The requests tab shows all raw requests logged. It displays when the request occurred (time stamp) and who send it (from) to what service (to). If Tagged Tracing is applied, the trace ID reveals information about the trace to which the request belongs. Optionally, the monitors collect payload information of the requests as well.

**Traces tab.** Figure 4.10 displays the trace tab. The traces tab shows the transaction traces. The traces tab is especially useful for recognizing patterns in the services usage. Each trace can be inspected, which shows additional information for a trace. In figure 4.11 the windows for inspecting a trace are shown. The first tab shows the trace path. The tab occurrences shows the individual occurrences of traces, the duration of individual traces and at what time the trace took place. In addition, the average trace duration is shown. Individual occurrences can be further inspected as the third window shows. Here, each single request of an individual trace is displayed with the time of occurrence, services involved and optionally the payload.
Figure 4.8: The Dashboard Status tab shows an overview of all services that are registered. The Requests to and from a particular service indicate how active the service has been.

Figure 4.9: The Dashboard Requests tab shows all raw requests logged. It displays when the request occurred (time stamp) and who send it (from) to what service (to). If Tagged Tracing is applied, the trace ID reveals information about the trace to which the request belongs. Optionally, the monitors collect payload information of the requests as well.
Figure 4.10: The Dashboard Traces tab shows the transaction traces. The Traces tab is especially useful for recognizing patterns in the services usage. The occurrences reflect the number of times a trace with a particular trace path were executed. Trace Length reflects the length of the trace path. The Involved Entities denotes the services part of the trace are display.
Figure 4.11: The windows for inspecting a trace are shown. The first window shows the trace path. The second window shows the individual occurrences of traces, the duration of individual traces and at what time the trace took place. In addition, the average trace duration is shown. Individual occurrences can be further inspected as the third window shows. Here, each single request of an individual trace is displayed with the time of occurrence, services involved and optionally the payload.
Figure 4.12: The Dashboard Overview Tab shows a graphical overview of the dependencies between services.
Chapter 5

Evaluation

Transaction tracing is a method to provide insight in a service cluster which cross cuts the three monitoring dimensions we defined. For a given service, all transaction traces that pass through that particular service contain information on the users of the service and at what time that specific service was accessed. For a given user, all transaction traces coming from that specific user contain information on the services accessed and at what time by that specific user. For a given time, all transaction traces that occurred at that specific time contain information on the services that were used and by which users at that specific time. In Chapter 3, we described the three fundamental online tracing techniques: (1) Timestamped Tracing; (2) Recognized Tracing; and (3) Tagged Tracing. With a controlled experiment we compare these three techniques. See Section 5.1 for this comparison.

As described in Chapter 3 the monitoring of the service cluster is achieved through monitoring proxies. These monitoring proxies report their data to the centralized analyzer. In Chapter 4 we distinguished two reporting approaches to report data to the centralized analyzer: (1) instantaneous reporting; and (2) interval reporting. With a controlled experiment we compare these two approaches. See Section 5.2 for this comparison.

5.1 Tracing techniques

The goal of this experiment is to determine the performance of each of the three fundamental online tracing techniques, by weighing up the pros and cons against each other. In Subsection 5.1.1 we describe the experiment setup and define the criteria which we use to compare the three tracing techniques. The results of the experiment are in Subsection 5.1.2. A discussion of the results and our conclusions can be found in Subsection 5.1.3.

5.1.1 Experiment Planning

In order to compare the three tracing techniques, we define the following criteria:

Monitor type Monitor type refers to the type of monitor as described in Chapter 2: non-intrusive, weak-intrusive or intrusive monitoring.
5. Evaluation

**Service cluster independent** This property describes whether or not the functionality can be used for other service clusters, or if it is a service cluster specific implementation.

**Accuracy** Accuracy describes the correctness of the generated traces.

**Overhead** Overhead refers to the increase in computation time due to the monitoring. Possibly, an application has to conform certain performance requirements, therefore limiting the amount of overhead a monitoring solution may have.

**Complexity** The complexity of the monitoring aspect and analyzer aspect.

**Effort** The required effort to adopt the technique in a new or existing service cluster.

**Monitor type.** As described in Chapter 2 we define three monitoring types for SOA: (1) non-intrusive monitoring; (2) weak-intrusive monitoring; and (3) intrusive monitoring. The monitor type required for a tracing technique depend largely on the requirements of the tracing technique. If the tracing technique does not need to change the functionality of the service cluster at all, the tracing technique is implementable with non-intrusive (external) monitoring techniques. If the tracing technique requires modification of the communication aspect, but no modification of the services themselves, the tracing technique is implementable with weak-intrusive monitoring techniques. Finally, if modification of the services is required, the tracing technique needs to be implemented with intrusive monitoring techniques. This criterion reveals information about the monitoring solution regarding the impact the solution may have on the behavior of the application. Non-intrusive monitoring implies no impact at runtime of the monitoring solution on the target application. In addition, non-intrusive monitoring does not require source code access of the service cluster. Weak-intrusive monitoring implies limited impact and requires source code access of the communication layer (for example the ESB). Intrusive monitoring infringes with the service code, thus bugs in the monitoring solution could directly affect the functionality of the target application. Furthermore, intrusive monitoring indicates that source code access of services is required. See Section 2.3 for a more extensive description of monitoring types.

**Service cluster independent.** The service cluster independent criterion describes whether the tracing technique implementation is implementable independent of the service cluster. Service cluster independent solutions have advantages over service cluster dependent solutions. A service cluster independent solution indicates that the solution may be reused for different service clusters. Reusability is one of the underlying principles of SOA [64, 65, 76]. Reusability reduces the development, test and maintenance costs [65, 27]. Developing a service cluster independent solution also indicates that the solution is loosely coupled with the service cluster. Loose coupling is considered a good programming practice to achieve higher software quality [27]. Another advantage of service cluster independent solutions is that no knowledge of the internal operation of the services is required. Thus the tracing solution is still implementable if knowledge of the internal operation of services is unavailable, for example if the service cluster was designed by a different development team than the development team of the tracing solution.
**Accuracy.** Accuracy describes the correctness of the generated traces. Traces are a collection of requests which are considered to be directly related to each other. However, in practice requests may incorrectly be considered as related to each other. As extensively described in Chapter 1 and Chapter 2, we distinguished three fundamental monitoring dimensions consisting of 'users', 'services' and time. The traces are intended to map users, time and services to each other. Therefore, traces should correctly map users to time and services. If all the requests that were part of a particular transaction trace are related to each other by the tracing technique, then the tracing technique correctly deduced information on all of the three monitoring dimensions. In order to be able to measure the correctness of the reconstructed transaction traces we ran the service cluster on AIS data of June 1st, 2007. We used the data of one specific date for all measurements such that we have controllability over the tests. This way, it is clear what to expect every test and each test is performed under equal circumstances. We analyzed this data to identify the actual transaction traces, after which we ran it 20 times for each communication model (asynchronous vs synchronous and bidirectional vs unidirectional). The analyzing of the data was a precise and important task, as wrongfully established traces would yield incorrect results regarding the accuracy of the tracing techniques. The analysis was done in two stages. First, the payload of all the AIS data which would be sent through the service cluster was analyzed to identify the trace paths that should be present. Then, the services operation was run with intentional pauses between the different services and with a large pause between different traces. The pause between different traces was chosen to be long enough that it should never exceed the duration of an entire trace. This way, the established traces could easily be verified to be correct, as a trace with a duration longer than the pause between different traces would indicate problems. We extended our Java implementation with the ability to store traces to disk and to compare different files containing traces on the measurements we devised.

We did not evaluate the accuracy of Recognized Tracing, as the accuracy of Recognized Tracing differs for each service cluster as it is a service cluster specific implementation. The accuracy depends on the ability to extract traces by recognizing the patterns that the payload and message headers exhibit [10]. Therefore, the results would not claim anything about the accuracy of Recognized Tracing, it would tell whether our service cluster contains predictable patterns or not.

Reconstructing the correct mapping of users, time and services may fail for various reasons. For instance, the trace path may not be completely reconstructed or requests that are not related are linked together in one trace. We constructed the following two measurements to measure different types of reconstruction failures:

**Request Correctness** The percentage of reconstructed transaction traces which only contain requests that are correctly linked to each other.

**Path Correctness** The percentage of reconstructed transaction traces which correctly reflect a complete trace path.

If a tracing technique scores 100% on both tests, the traces correctly map users, time and services to each other. Request Correctness measures the percentage of reconstructed transaction traces which only contain requests that are correctly linked to each other. This is
5. Evaluation

especially relevant when extracting details about trace paths such as the average duration of a trace and the average response time per hop. Furthermore, insight in particular traces, for example for the investigation and tracking of specific cases or bugs requires that all requests of a trace are correctly linked to each other. This is not the focus of this work however, and is therefore classified as potential future work. Path Correctness measures the percentage of reconstructed transaction traces which correctly reflect a complete transaction path. With Path Correctness we measure whether the communication patterns of the service cluster are correctly identified. This is a strict measure, since the percentage of complete trace paths is measured. Thus for example a trace path contains 15 correct requests but the 16th request also belonging to this particular trace path is missed, it is counted as incorrect. This method was constructed with the following purpose in mind, being able to correctly identify complete traces such that communication patterns may be identified. Complete traces assure a correct mapping between users and services, satisfying two of the three dimensions.

Overhead. To measure the computation overhead of the tracing approaches, we focused on the monitoring aspect. The computation time of the analyzer aspect is less relevant, because it (may) run on a different machine and does not (need to) infringe with the service cluster. We performed 100 runs for each monitoring configuration. Thus 100 runs with no monitoring, 100 runs with only timestamps and sender/receiver information (as required for Timestamped Tracing), 100 runs with payload, timestamps and sender/receiver information (as required for Recognized Tracing) and finally 100 runs with trace IDs, timestamps and sender/receiver information (as required for Tagged Tracing). We measured the runtime of each of those runs. We estimate the computation overhead of the tracing techniques by comparing the average runtime of each monitoring configuration to the average runtime of a blank system with no monitoring.

Complexity. The complexity criterion describes the complexity of the monitoring aspect and analyzer aspect. As described in Chapter 4, the implementation consists of a monitoring aspect, which is logging all the relevant information and an analyzer aspect, which is concerned with using the logged information to satisfy our monitoring goals. The analyzer is responsible for reconstructing the traces from the information it receives from the monitors. If the monitors log more information, it is easier to reconstruct the traces. The tracing techniques differ in the amount of information that is logged and therefore the Analyzer aspect may become more complex. This criterion displays the difference how the tracing techniques differ with respect to the Monitoring and Analyzer Aspect. We compare the complexity of the monitoring aspect based on the lines of code required for each tracing technique. For the analyzer aspect only Timestamped Tracing and Tagged Tracing are part of the comparison, as we did not implement the Recognized Tracing method in the analyzer. Recognized Tracing differs for each service cluster as it is a service cluster specific implementation. The complexity depends on the complexity of the patterns that the service cluster payload and message headers exhibit [10]. In addition to LOC, we used the computation time of the analyzer to compare the complexity of the tracing approaches. A higher computation time indicates that it is harder to reconstruct the transaction traces for the analyzer. The computation time of the analyzer is not relevant for the overhead, as it
may run on a different machine and therefore does not (need to) affect the performance of 
the service cluster.

**Effort.** The effort criterion describes the required effort to adopt the technique in a new 
or existing service cluster. The intrusiveness of the monitoring solution is an indication 
for the graduation of coupling between the monitoring code and the code of the services. 
A higher coupling between the monitoring code and the code of the services indicates a 
higher effort to adopt the technique in an existing service cluster, as it requires adapting all 
services. Another pointer for the required effort to adopt the technique in a new or existing 
service cluster is the complexity of the monitoring and analyzer aspect. A higher complexity 
indicates that the required effort to adopt the technique increases.

### 5.1.2 Experiment results

**Monitor type.** Tagged Tracing requires adjusting the communication protocol, as re-
quests need to be tagged with trace IDs. As described in Chapter 2 we consider an approach 
weak-intrusive if it only changes the communication layer and not the services itself. In syn-
chronous service clusters, data-tagging nor thread variables are required, therefore Tagged 
Tracing is implementable on a synchronous service cluster with weak-intrusive monitoring 
techniques. However, in asynchronous service clusters, threads and shared data resources 
need to be tagged. Therefore Tagged tracing is only implementable with intrusive internal 
monitoring techniques in asynchronous service clusters. Recognized Tracing and Timestamped Tracing are both implementable with non-intrusive 
external monitoring techniques.

**Service cluster independent.** Timestamped Tracing and Tagged Tracing are service clus-
ter independent, meaning that the functionality does not have to be adapted if it is used on 
a different service cluster. Recognized Tracing is the only one of the three tracing methods 
that is not service cluster independent. Knowledge about the operation of the service cluster 
is required to correctly recognize patterns in the message headers or payload for messages 
that belong to the same trace. The fact that the implementation requires knowledge of the 
operation of the specific cluster implies that it cannot (without adaptations) be reused for 
other service clusters.

**Accuracy.** Figure 5.1 and Figure 5.2 show the Request Correctness on synchronous and 
asynchronous communication respectively. Request Correctness reflects percentage of re-
quests that are correctly matched together to form a trace path. Thus Request Correctness 
measures the correct combination of requests and not the correct identification of commu-
nication patterns. This is especially relevant when extracting details about trace paths such 
as the average duration of a trace and the average response time per hop. Furthermore, it 
offers insight in particular traces, thus allowing the investigation and tracking of specific 
cases or bugs. In Figure 5.1 the performance of the Tagged Tracing and Timestamped Trac-
ing on a cluster using synchronous communication is shown. Clearly, all three methods 
are extremely valid for matching requests to traces in synchronous communication service
clusters; All three score a perfect 100%. On service clusters using asynchronous communication the accuracy results are different. As Figure 5.2 shows, Tagged Tracing still performs extremely well with 100% correctness. However, the Timestamped Tracing technique performs horribly with only 2.2% correctness for bidirectional communication and 1.8% in the case of unidirectional communication. In an effort to improve the Timestamped Tracing quality, the time stamp matching algorithm was extended with information about the communication patterns of the cluster. Concretely, the timestamped algorithm was extended with information that there is no circular communication (e.g. $A \rightarrow B \rightarrow C \rightarrow A \rightarrow B$) and that there are no repeating requests for a particular trace (e.g. $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow C$, $B \rightarrow C$). However, the performance did not improve and remained a mere 2.2%.

The second measurement, Path Correctness, measures the percentage of reconstructed transaction traces which correctly reflect a complete transaction path. Path Correctness measures whether the communication patterns of the service cluster are correctly identified. This is a strict measure, since the percentage of complete trace paths is measured. Thus
for example a trace path contains 15 correct requests but the 16th request also belonging to this particular trace path is missed, it is counted as incorrect. This method was constructed with the following purpose in mind, being able to correctly identify complete traces such that communication patterns may be identified. Complete traces assure a correct mapping between users and services, satisfying two of the three dimensions. In Figure 5.3 and Figure 5.4 the amount of correct complete trace paths is shown. As Figure 5.3 shows, Tagged Tracing achieves 100% correctness for both bidirectional as unidirectional communication on synchronous communication. Timestamped Tracing on the other hand, achieves 100% for bidirectional communication, but drops slightly to 97% for unidirectional communication. Figure 5.4 displays the performance on asynchronous communication. Tagged Tracing maintains its perfect score as it shows 100% correctness for both bidirectional and unidirectional communication. Timestamped Tracing on the other hand scores a discomforting 0%. Again, adding communication information to the timestamped algorithm did not improve the results as it remained 0% correctness.
5. Evaluation

Overhead. Figure 5.5 displays the average overhead of Tagged Tracing, Recognized Tracing and Timestamped Tracing in the experiment. As shown, the overhead of Timestamped Tracing is 28.07%, Recognized Tracing 35.70% and Tagged Tracing 42.55%. Our AIS case study system requires the approaches to handle 1 minute of AIS data in less than 60 seconds. All of the approaches performance conform this performance requirement. For 1 minute of data, Timestamped Tracing requires on average 13.81 seconds, Recognized Tracing requires on average 14.63 seconds and Tagged Tracing requires on average 15.37 seconds. A blank system with no monitoring at all requires 10.79 seconds.

Complexity. The analyzer code for Recognized Tracing was not implemented, because of its service cluster specific nature, therefore it is not present in the Analyzer complexity comparison. Figure 5.5 and Figure 5.6 are indicators that the complexity of the monitoring code of Tagged Tracing is higher than the complexity of the monitoring code for Recognized Tracing and Timestamped Tracing. As Figure 5.5 shows, Tagged Tracing requires nearly twice as much computation time as Timestamped Tracing, with Recognized Tracing in between Timestamped Tracing and Tagged Tracing. As Figure 5.6 shows, the Lines Of Code (LOC) of Tagged Tracing is twice the LOC of Recognized Tracing and Timestamped Tracing.

Figure 5.6 and Figure 5.7 are indicators that the complexity of the analyzer code of Timestamped Tracing is higher than the complexity of the analyzer code of Tagged Tracing. The Analyzer computation time of Timestamped Tracing is about 8 times as much: Tagged Tracing requires 1.46s or 1.74s respectively for 1 minute synchronous or asynchronous communication of AIS data, Timestamped Tracing on the other hand requires 12.21s or 14.30s respectively for 1 minute synchronous or asynchronous communication of AIS data. As Figure 5.6 shows, the LOC of Timestamped Tracing is also more than the LOC of Tagged Tracing. The computation time of the analyzer is not relevant for the overhead of the tracing techniques, as the analyzer may run on a different machine than the services. It is merely
Tracing techniques

Figure 5.6: The LOC for each tracing method

![Lines of Code (LOC)](image)

Figure 5.7: The computation time of the analyzer on 1 minute of AIS data.

![Analyzer Computation Time](image)

an indication of the complexity of the analyzer.

The LOC cannot be compared between the monitoring and the analyzer aspect, as the monitoring part was implemented in Ruby, whereas the analyzer part in (the more verbose) Java. As stated in Chapter 4, the analyzer is implemented in Java as Java has better GUI support than Ruby. The monitoring aspect is implemented in Ruby, because it allows fast development of distributed systems. However the computation time is a good indicator and shows that the complexity of the Tagged Tracing is in the Monitoring part and the complexity of the Timestamped Tracing is in the analyzer part.

**Effort.** The additional effort to add a new service to the service cluster when using Timestamped Tracing is low. This is because Timestamped Tracing does not require internal access to services. In our implementation new services are automatically able to monitor through inheritance of the MonitorableService class. Recognized Tracing and Tagged Tracing are automatically supported through inheritance of the MonitorableService class. In the case of Tagged Tracing and an asynchronous service cluster, all threads and shared resources in the new service need to be identified and adjusted for tagging. Therefore we consider the effort high for adding a new service to an existing cluster when using Tagged Tracing. In the case of Recognized Tracing, the service operation needs to be (manually) analyzed to adjust the Central Analyzer accordingly, such that requests that go through this new service also are recognizable and may get mapped to their respective trace. However,
5. Evaluation

if a framework is used which has a very predictable messaging pattern, the actual additional
effort for adding a new service may be limited. The required effort to adopt the tracing
approach in an existing service cluster is the lowest for Timestamped Tracing. This is be-
cause it is an non-intrusive monitoring approach, therefore the code of the services does not
need to be adapted. Recognized Tracing is also an non-intrusive monitoring approach, but
Recognized Tracing requires investigation and analysis of the message headers and payload
patterns in order to be able to correctly store traces of a given service cluster. Therefore it
is the most unpredictable of the three methods. For synchronous service clusters, Tagged
Tracing may be implemented as a weak-intrusive approach, thus only requiring the com-
unication layer to be adapted. In asynchronous service clusters, threads and shared data
resources need to be identified and adjusted for tagging.

5.1.3 Discussion and Conclusions

Recognized Tracing and Tagged Tracing have the advantage that they may be implemented
with non-intrusive monitoring techniques. Tagged Tracing on the other hand is a weak-
intrusive (in synchronous service clusters) or internal monitoring approach. Therefore,
Recognized Tracing and Tagged Tracing have the following two advantages over Tagged
Tracing: (1) Recognized Tracing and Tagged Tracing are usable even if there is no source
code access; and (2) Recognized Tracing and Tagged Tracing do not adjust the original
implementation, thus it will not add complexity and/or bugs to the original functionality.
Both Tagged Tracing and Timestamped Tracing are service cluster independent implemen-
tations. This is an advantage, because the software may be developed independent of the
service cluster, no knowledge about the operation of the service cluster is required. Also the
implementation may be reused for many service clusters. Recognized Tracing on the other
hand requires a deep understanding of the message header and payload patterns, to correctly
map them to traces. With regard to accuracy and overhead the following observations can
be made: Timestamped Tracing has the least overhead and in synchronous service clusters
it reaches an accuracy of 98%-100%. Therefore, it is probably the best choice in service
clusters with synchronous communication. In asynchronous service clusters Timestamped
Tracing is unable to match requests correctly to traces. Therefore it is not an option for asyn-
chronous service clusters. Recognized Tracing has less overhead than Tagged Tracing, but
more overhead than Timed Tracing. The accuracy of Recognized Tracing is unpredictable,
because it is different from service cluster to service cluster. However, the accuracy is not
as strongly affected by the type of communication as with Timestamped Tracing. This is
because it uses the payload and message headers for matching, which are not as sensitive
to the communication type as timestamps. Tagged Tracing is the most safe bet for high
accuracy. In the controlled experiment, it reached 100% correctness on synchronous and
asynchronous communication. In the experiment the overhead of Tagged Tracing was not
near the performance requirements of the AIS case, as it took on average 15.37 seconds
 computation time on 1 minute data.

In conclusion, we think that Tagged Tracing is the best approach as it has high accuracy
and is service cluster independent. The additional costs, meaning the additional overhead
and the modification of the communication layer and data structures are well worth the
benefits. Table ?? displays the results of the experiment.

Table 5.1: A comparison of tracing techniques

<table>
<thead>
<tr>
<th>Tracing Method</th>
<th>Accuracy (synchronous)</th>
<th>Accuracy (asynchronous)</th>
<th>Monitoring overhead</th>
<th>Effort (existing)</th>
<th>Effort (new)</th>
<th>Requires source code access</th>
<th>Service cluster independent</th>
<th>Monitoring complexity</th>
<th>Analyzer complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tagged</td>
<td>100%</td>
<td>100%</td>
<td>43%</td>
<td>High</td>
<td>High</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Recognized</td>
<td>-</td>
<td>-</td>
<td>36%</td>
<td>High</td>
<td>High</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>-</td>
</tr>
<tr>
<td>Timestamped</td>
<td>98-100%</td>
<td>0%</td>
<td>28%</td>
<td>Low</td>
<td>Low</td>
<td>No</td>
<td>Yes</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

5.2 Reporting Approaches

As described in Chapter 3 the monitoring of the service cluster is achieved through monitoring proxies. These monitoring proxies report their data to a centralized analyzer. In chapter 3 we distinguished two reporting approaches to report data to the analyzer: (1) instantaneous reporting; and (2) interval reporting. With a controlled experiment we compare these two approaches. In Section 5.2.1 we describe the experiment setup. The results of the experiment are in Section 5.2.2. A discussion of the results and our conclusions can be found in Section 5.2.3.

5.2.1 Experiment Planning

For the monitor reporting we explored two types of push-techniques. (1) instantaneous reporting; and (2) interval reporting. Instantaneous reporting instantaneously reports events as they occur. The advantage is optimal reaction speed, as an event gets reported as fast as possible. The downside is the enormous overhead. For every single request a report has to be sent resulting in a 100% report to message overhead. Interval reporting means that all events during a given time frame are reported by one single report. In our experiment we implemented and tested instantaneous reporting and interval reporting with time frames of 0.05, 0.1, 0.25, 0.5 and 1 second. During a run, 71288 messages are sent.
5. Evaluation

Table 5.2: Runtime measured for Reporting approaches

<table>
<thead>
<tr>
<th>Reporting Method</th>
<th>Runtime</th>
<th>Overhead (%)</th>
<th>Number of Reports</th>
<th>Messages per Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>No reporting</td>
<td>10.784s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Instantaneous</td>
<td>25.949s</td>
<td>140.15%</td>
<td>71288</td>
<td>1</td>
</tr>
<tr>
<td>Interval (0.05s)</td>
<td>13.835s</td>
<td>28.31%</td>
<td>1505</td>
<td>47</td>
</tr>
<tr>
<td>Interval (0.1s)</td>
<td>13.812s</td>
<td>28.05%</td>
<td>1503</td>
<td>47</td>
</tr>
<tr>
<td>Interval (0.25s)</td>
<td>14.102s</td>
<td>30.77%</td>
<td>739</td>
<td>96</td>
</tr>
<tr>
<td>Interval (0.5s)</td>
<td>14.280s</td>
<td>32.41%</td>
<td>417</td>
<td>171</td>
</tr>
<tr>
<td>Interval (1s)</td>
<td>14.357s</td>
<td>33.11%</td>
<td>221</td>
<td>323</td>
</tr>
</tbody>
</table>

5.2.2 Experiment Results

Instantaneous reporting instantaneously reports events as they occur. Interval reporting means that all events during a given time frame are reported by one single report. In our experiment we implemented and tested instantaneous reporting and interval reporting with time frames of 0.05, 0.1, 0.25, 0.5 and 1 second. During a run, 71288 messages are sent. Table 5.2 shows the runtime measured for different reporting approaches on the Timestamped Tracing implementation.

5.2.3 Discussion and Conclusions

Table 5.2 shows the runtime measured for different reporting approaches on the Timestamped Tracing implementation. The Runtime column reflects the computation time required to report all the messages. The overhead of the instantaneous reporting method is nearly five times larger than the overhead of interval reporting. Waiting just 0.05 seconds to collect messages for a batched report lowers the runtime to 13.835 down from 25.949 seconds. The interval reporting with a time frame of 0.1 second performs the fastest. We speculate that the small increase for larger time-frames is caused by the increasing amount of messages that have to be handled in a single report. For example reports contain on average $71288/221 = 323$ messages when the time frame is set to 1 second, while only 47 messages when the time frame is set to 0.1 second. We speculate that the time frame of 0.05s sends about the same amount of messages as the time frame of 0.1s, because of synchronization locks. A lock has to be obtained on the recorded messages, because the reporting and monitoring are handled in different threads. For implementation details, please have a look at Listing A.4 in Appendix A.
5.3 Threats to validity

There are some threats to the conclusions that we drew from the experiments. First and foremost, the test was only applied on a particular case. The communication may not represent the communication pattern of an arbitrary SOA. For instance, the volume of messages in the AIS case may not be representative for high volume communication that might, for example, be necessary in a stock market application. Furthermore the approach is not tested for multiple streams of data input. Therefore the approach should be tested on more SOA implementations to confirm the validity of the results. This is noted as future work in Chapter 7.

The overhead and accuracy measurements reflect the overhead and accuracy as measured on our implementation of the three fundamental online tracing techniques. Therefore, the measured overhead and accuracy may not reflect the overhead and accuracy in any arbitrary system. In order to confirm the validity of this experiment, it should be repeated on a significant number of other implementations.

The accuracy measurements Request Correctness and Path Correctness were chosen to reflect the Accuracy of Tracing Techniques. Since we used archived data, we were able to calculate the correctness accurately for these measurements. If a method scores 100% on both of these measurements, it means that all traces have been correctly reconstructed. However, in case the tracing technique is not flawless, more and/or other measurements may give a better impression of the accuracy of the technique. For example, the Path Correctness measurement may quickly yield low percentages as a trace missing only 1 request of the trace path is counted as an incorrect trace in our measurement. A possible improvement could be to adjust it to a fractional number e.g. a trace is counted as (15/16) correct in case of a trace path of length 16 with 15 requests correct. In addition, we choose the respective measurements because the measurements reflect the tracing goals we identified. Other and/or more measurements may reflect the tracing goals even better or may be used to reflect different tracing goals.

The Effort criterion was based on a number of metrics and facts of our implementations. However, the calculated Effort is only an estimation, we have not proven a direct correlation between the metrics and facts and the Effort criterion. In addition, the metrics reflect the metrics such as measured on our implementation and may not reflect any arbitrary implementation.

The Timestamped matching algorithm could be improved which might yield better results. For example, the algorithm could be extended with additional information on the type of communication it could expect, however this makes the approach less generic and requires in depth knowledge of the structure of the cluster. If such information is available, it is likely that source code access is also available, which means Tagged Tracing could be applied as well which performs much better. Nevertheless, the algorithm employed in Timestamped Tracing may be enhanced to achieve better results. But since the results are a few orders of magnitude worse than Tagged Tracing, we suspect that Tagged Tracing is a much easier and more accurate approach and therefore advice Tagged Tracing for any asynchronous service cluster.

The experiments were performed on a single computer, thus each service used the same
clock. The accuracy of Timestamped Tracing may be less in service clusters with no synchronized clocks.

The estimation made on the effort of adding the approach into a given project is very language-dependent, and since the approach was implemented on Ruby it is a valid concern whether the calculated effort is relevant or comparable to other programming languages. However, while it is true that Ruby does provide good support for these kind of constructs (especially with regards to separation of concerns), the approach does not use any techniques which are not commonly available in modern object oriented languages such as Java and C++. A special note on this is that Object Oriented languages may be extended with Aspect Oriented Programming (AOP). Aspect Oriented Programming enriches traditional Object Oriented programming languages with the definition of aspects, which allow for easier separation of concerns, which is particularly useful for monitoring [35][44][45].
Chapter 6

Related Work

Even though monitoring and tracing are well explored domains in computer science, there are few approaches that use these methods in dynamic or distributed systems like SOA [23]. While tracing is especially useful in SOA due to its dynamic nature, the distributed aspect makes tracing radically different on SOA than on monolithic systems [17, 23, 30, 57]. Moreover, to the best of our knowledge, there is hardly any literature on the evaluation of tracing approaches on SOA nor articles that compare the tracing approaches to each other. Therefore we devised an evaluation setup on which we evaluated tracing techniques for SOA. In this chapter, we will discuss comparable approaches we identified in the literature. We have identified a number of tracing approaches and a number of service monitoring approaches.

6.1 Comparable approaches

Tracing. A number of approaches for Change Impact Analysis rely on tracing [16, 22, 36, 39, 52, 82]. Change Impact Analysis is the process of identifying the potential consequences of a change in one software unit [9]. An approach described in [16] monitors applications to evaluate the current and historical state of the application. If the application is not in compliance with its requirements, traces are used to identify affected entities. These traces are built using information retrieval techniques. Egyed et al present an approach [22] that relates features to source code, it should be adopted in unit-tests to identify the features. It relates these features to source-code by relating runtime events based on the time of occurrence. Marcus et al present an approach [52] which maps source code to documentation and uses trace links to identify related source code. Trace links are established through information retrieval. A tracing method IntelliTrace is presented in [36], which is used to support impact analysis of SOA systems. Trace links are assumed to be present, the authors refer to [16, 22, 52] for establishing trace links.

Dynamic Analysis is a popular method for program comprehension, in a survey published in 2009 by Cornelissen et al [17] 176 approaches which use dynamic analysis for program comprehension were identified in the period 1972-2008 [17]. More than 30 of the approaches that the survey covers use execution traces for their dynamic analysis. How-
ever, only eight of the approaches that the survey covers are aimed at distributed systems, even though Distributed Systems and especially SOA are more difficult to understand than monolithic applications due to their dynamic nature [17, 30, 57]. An approach is presented in [23] which traces incoming web methods, but also provides the ability to monitor internal method calls. A Thread-Local integer is set to trace the consequent calls of a particular incoming call. Multi-threaded applications are not covered. Chen et al [14] discuss an approach for a monitoring framework which aims at aiding software engineers by presenting a runtime topology of a service cluster. The intend is to trace requests throughout the service cluster by using an identifier which relates all consequent requests.

**Trace visualization.** An important advantage of dynamic analysis is that it presents a current overview of the system [10, 18], as it logs the current use of the system. However, this implies a high volume of data [10] which is continuously increasing. Therefore, a lot of approaches investigate the reduction of this information [33, 73] and visualization techniques [19, 41] to reduce the cognitive overload for the users. In a work by Pacione et al [67] five dynamic visualization tools are compared. in a survey published in 2004 [34], eight trace visualization are described. Cornelissen et al present a work [18] in which they focus on the visualization aspect of traces, the related work section contains an extensive overview of 21 other trace visualization approaches.

**Service monitoring.** Service Oriented Architectures are a very live topic and because of the volatile nature of these architectures, many researchers employ some sort of monitoring on these Service Oriented Architectures to continuously validate the behaviour. These monitoring approaches are interesting to get an understanding of how services are often monitored. The following two surveys provide a nice overview of the work on runtime monitoring in software engineering: A survey on self-healing systems: approaches and systems by Psaier and Dustdar published in 2011 [70] and A taxonomy and catalog of runtime software-fault monitoring tools by Delgado et al published in 2005 [20].

Most internal monitoring approaches are Aspect Oriented approaches [1, 4, 42, 47, 80, 83, 84, 86]. Aspect Oriented Programming is a programming technique introduced in the late 1990s, because developers ran into problems that were not clearly expressible in the Object Oriented (or Procedural) manner [45]. The intend of Aspect Oriented Programming is to avoid 'tangled’ or 'scattered’ code. When different responsibilities and functionalities are intermixed in the code, this is considered tangled code. Scattered code describes the situation when a certain responsibility or functionality is spread out over different classes or files. Generally, some design decisions are difficult to cleanly capture in code. The issues these design decisions address are called aspects. The reason these aspects are often hard to modularize, is because they cross-cut the basic functionality of the system. An example of such a cross-cutting concern is monitoring, as the events that have to be monitored are often scattered over multiple classes. For more information see [35, 45, 44].

Another common monitoring approach is the probe technique [10, 15, 43, 83]. Monitoring through probes means that probes at strategically placed positions (e.g. near the location of consumers) send requests to a service in order to extract information of the monitored ser-
Existing monitoring platforms for SOA

Both monitoring and SOA are big markets in industry. Therefore there are a lot of business with approaches and tools. Gartner has released a document which assesses the APM (Application Performance Monitoring) market [10]. In this report Gartner estimates that in 2011, approximately $ 2 billion will be spent on APM licenses and first-year maintenance contracts. There are several development platforms in industry to deploy Service Oriented Architectures, many of these provide inbuilt monitoring approaches. Example development platforms are the IBM SOA platform, JBoss Enterprise SOA Platform, Apache Axis2, and Ebay Turmeric. However, these businesses generally do not provide insight in their methods, nor release measurements regarding the overhead [46]. We will summarize and describe the characteristics of the Ebay Turmeric platform. We will especially look into how they facilitate and how they approach the monitoring aspect.

**Ebay Turmeric.** Most of the internal software by eBay is service oriented. They have been using SOA technology for over 10 years and have developed their own platform. eBay summarizes their own platform as following: "Turmeric is a comprehensive, policy-driven
6. RELATED WORK

Figure 6.1: eBay SOA Platform [51]

SOA platform that you can use to develop, deploy, secure, run and monitor SOA services and consumers."[21]. From this description it may also become clear, that they consider monitoring as one of the important aspects of SOAs.

eBay uses the platform for their own services. The internal system of eBay has to deal with 20 billion messages at any given time, thus proving the scalability of their approach. Since speed is important for a website like eBay, there are also strong constraints on the overhead. Sastry Malladi from eBay mentions that the latency of the overhead may not exceed 2ms [51].

The platform provides developer tools in the form of eclipse plugins (thus, full IDE support for developing SOA apps with extra functionalities generated on top). Also there are operational tools for management of services (through the registry/repository), monitoring (through a dashboard) and alerting.

In Figure 6.2 the Service Provider Framework is shown. The numbers represent the steps of an incoming message. The monitoring does not interfere/get generated inside the service source code, however it extracts that information from the message before it arrives at the service. The monitoring is performed by listeners on the pipeline (message bus). Interested (internal) parties may define metrics themselves to define what kind of information is relevant. This information ultimately pops up in a dashboard, where it can be viewed by the interested party. In the eBay context, one may for example define a monitor for the event "a search on the term 'ipad'". The monitor needs knowledge of the message structure, to extract that kind of information. It gets this information from the service wsdl file. The functionalities of the dashboard are described as:

- What services are currently out there
- What consumers are using that service (daily, hourly, ...)

58
What is the traffic like

What are the errors

What are the dependencies

Ebays Turmeric project is still under development and the dashboard is still work in progress. For more information see the Turmeric Wiki page [21]. For an example of the dashboard application see [61].
Chapter 7

Conclusions and Future Work

7.1 Contributions & Conclusions

Maritime Safety and Security Systems represent the class of systems that we target with this thesis. We have implemented tracing and monitoring concepts on a service cluster operating on Automatic Identification System (AIS) data. AIS is a communication system which allows ships to communicate their positions and status, this is essential for the safety of Maritime Traffic \[53, 63\]. The domain of the case study system requires the system to run continuously and the system should never be down as this has implications on the safety of Maritime Traffic. In order to perform maintenance on the system, information concerning the three fundamental monitoring dimensions of 'services', 'users' and 'time' are required. As traces crosscut these dimensions, we are interested in learning how to extract traces of dynamic and distributed systems. Through a literature study we identified three fundamental online tracing techniques, from which all tracing approaches can be derived: Tagged Tracing, Recognized Tracing and Timestamped Tracing. Even though monitoring and tracing are well explored domains in computer science, there are few approaches that use these techniques in dynamic or distributed systems like SOA \[23\]. Tracing is especially useful in SOA due to its dynamic nature. The distributed aspect makes tracing radically different on SOA than on monolithic systems \[17, 23, 30, 57\]. Moreover, to the best of our knowledge, there is hardly any literature on the evaluation of tracing approaches on SOA nor articles that compare the tracing approaches to each other. Therefore we devised an evaluation setup on which we evaluated tracing techniques for SOA.

We performed a controlled experiment with both synchronous and asynchronous communication. In our experiment, the accuracy of Tagged Tracing was 100 percent on both synchronous and asynchronous communication. Thus Tagged Tracing operated correctly on both synchronous and asynchronous communication. The accuracy of Timestamped Tracing was 100 percent on bidirectional synchronous communication and 98 percent on unidirectional synchronous communication. However, Timestamped Tracing performed very bad on asynchronous communication in our experiment. The accuracy was 0 percent. We did not measure the accuracy of Recognized tracing, as the accuracy is service cluster specific. No conclusions can be drawn from an accuracy measurement, it would merely reveal
whether our implementation contained predictable patterns in the payload or message data.

In our experiment, the overhead of Tagged Tracing on the monitored system was the highest. The overhead of Tagged Tracing was 43 percent, the overhead of Recognized Tracing 36 percent and the overhead of Timestamped Tracing 28 percent.

An indication of the required effort of adopting the tracing technique in an existing cluster is obtained by looking at the monitoring type and whether or not the technique is service cluster independent. The level of intrusiveness reflects how much the monitors infringe with the application code, a lower level indicates a lower effort as the implications of the monitoring on the application are limited. If the solution is independent of the service cluster, in-depth knowledge of the operation of the service cluster is not required in order to correctly implement the tracing technique. Therefore, we consider whether or not a technique is service cluster independent an indicator for the effort. As described in our reflection, we did not prove a correlation between the indicators that we used and therefore the conclusions are merely an indication of the required effort. Timestamped Tracing may be implemented with non-intrusive monitoring techniques and is a service cluster independent solution. These advantages indicate a low effort to adopt Timestamped Tracing. Tagged Tracing is also service cluster independent solution, but requires more intrusive monitoring techniques than Timestamped Tracing. Therefore, the required effort is higher than the required effort for Timestamped Tracing. Recognized Tracing, like Timestamped Tracing, may be implemented with non-intrusive monitoring techniques. However, Recognized Tracing is a service cluster dependent solution, thus requires extensive analysis of the systems behavior to identify recognizable patterns. Therefore, this indicates that Recognized Tracing requires more effort than the other two techniques.

The results of the evaluation are summarized in Table 7.1.

<table>
<thead>
<tr>
<th>Tracing Method</th>
<th>Accuracy (synchronous)</th>
<th>Accuracy (asynchronous)</th>
<th>Monitoring overhead</th>
<th>Effort (existing)</th>
<th>Effort (new)</th>
<th>Requires source code access</th>
<th>Service cluster independent</th>
<th>Monitoring complexity</th>
<th>Analyzer complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tagged</td>
<td>100%</td>
<td>100%</td>
<td>43%</td>
<td>Moderate</td>
<td>Moderate</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Recognized</td>
<td>-</td>
<td>-</td>
<td>36%</td>
<td>High</td>
<td>High</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>-</td>
</tr>
<tr>
<td>Timestamped</td>
<td>98-100%</td>
<td>0%</td>
<td>28%</td>
<td>Low</td>
<td>Low</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Low</td>
</tr>
</tbody>
</table>

For the evaluation we had to evaluate the accuracy of the approaches. We developed a
dashboard which allows storing and loading of requests, such that the reconstructed traces could be evaluated for correctness. Furthermore, the dashboard aids maintenance and development on service clusters. Communication patterns and information regarding the three fundamental dimensions can easily be extracted through the traces tab. In addition, the dashboard provides an overview of the dependencies of services with a graph.

In our implementation of Tagged Tracing we added support for multi-threaded tracing. Multi-threading is becoming increasingly important, because of the recent trend of parallel computing and stagnating clock rates [68]. Tagged tracing presents some challenges in the case of a multi-threaded system, shared resources need to be tagged such that events that cross thread boundaries through shared resources are traceable. This is a relevant contribution, as to the best of our knowledge, there is no other approach described in the literature which implements this behavior. Existing approaches described in [2, 26, 29, 69, 81] allow multi-threaded tracing, but these are focused on tracing the events per thread and correctly linking events to their respective thread. They do not trace data which is shared through shared data resources.

7.2 Reflection

There are some threats to the conclusions that we drew from the experiments. The test was only applied on a particular case. In order to confirm the validity of this experiment, it should be repeated on a significant number of other implementations. This is noted as future work.

The communication may not represent the communication pattern of an arbitrary SOA. For instance, the volume of messages in the AIS case may not be representative for high volume communication that might, for example, be necessary in a stock market application.

The overhead and accuracy measurements reflect the overhead and accuracy as measured on our implementation of the three fundamental online tracing techniques. Therefore, the measured overhead and accuracy may not reflect the overhead and accuracy in any arbitrary system.

The effort criterion was based on a number of metrics and facts of our implementations. However, the calculated effort is only an estimation, we have not proven a direct correlation between the metrics and facts and the effort criterion. In addition, the metrics reflect the metrics such as measured on our implementation and may not reflect any arbitrary implementation.

The accuracy measurements 'Request Correctness' and 'Path Correctness' were chosen to reflect the accuracy of Tracing Techniques. Since we used archived data, we were able to calculate the correctness accurately for these measurements. If a method scores 100% on both of these measurements, it means that all traces have been correctly reconstructed. However, in case the tracing technique is not flawless, more and/or other measurements may give a better impression of the accuracy of the technique. For example, the 'Path Correctness' measurement may quickly yield low percentages as a trace missing only 1 request of the trace path is counted as an incorrect trace in our measurement. A possible improvement could be to adjust it to a fractional number e.g. a trace is counted as (15/16) correct
in case of a trace path of length 16 with 15 requests correct. In addition, we choose the respective measurements because the measurements reflect the tracing goals we identified. Other and/or more measurements may reflect the tracing goals even better or may be used to reflect different tracing goals.

7.3 Future work

As described in our reflection, the validity of the results could be increased by running the experiment on different implementations. Therefore we suggest the experiment to be repeated on different implementations. In addition, the experiment should be repeated with different implementations of the tracing techniques, to increase the validity of the measurements.

Furthermore, we suggest to extend the experiment with services that act on database changes. As described in chapter 3, database changes are not traceable with pure service transaction tracing, while these database changes may cause a permanent change in the state of an application. Tracing database changes requires the whole database model to be changed. Every data operation and its consequences need to be logged. The implications on the overhead of logging every data operation and its consequences are unknown, we therefore denote it as future work.

Hybrid tracing approaches could be explored, where multiple tracing techniques are used in combination with each other. This way, the strengths of each technique can be combined. For example, if some services are not under heavy load and therefore not suited for Tagged Tracing it could be implemented with the more lightweight Timestamped Tracing while the rest is implemented with Tagged Tracing. Furthermore, multiple techniques on the same component is also an option. For example combining Timestamped Tracing with Recognized Tracing on a single component. This way, the monitoring can still be achieved in a weak-intrusive manner, while the accuracy might improve. This could be especially useful in clusters with asynchronous communication, where the accuracy of Timestamped Tracing was particularly low in our experiment.

Control features such as toggling payload information on/off of specific services and toggling trace IDs off for specific services could be added to the dashboard. By extending the dashboard with such control features, the user has more flexibility over the overhead of the service cluster. The graph of the service cluster could also be improved by adjusting lines and sizes according to traffic. This way, it is immediately clear which services generate a lot of traffic and which services are used a lot. In addition, requests belonging to a certain trace might be drawn in a specific color, such that traces can also be easily identified on the graph.

In addition, the monitored information may get analyzed and displayed in more diverse ways by adding or removing tabs of the dashboard.
Bibliography


[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.


Appendix A

Code Extracts

Listing A.1: Extract of Registry.rb

REQUEST_COUNTER_BITS = 52;
MAX_SERVICES = 2**(64–REQUEST_COUNTER_BITS)–1

def register_service(name, address)
id = incr_count(name)
  if (id == 0)
    #maximum number of services reached
    return 0;
  else
    shifted_id = (@name_to_id[name] << REQUEST_COUNTER_BITS);
    @name_address[name] = address
    print "Registered #{name}, #{shifted_id}\n"
    return shifted_id;
  end
end

def incr_count(name)
  @mutex.synchronize do
    #critical section
    if (@count == MAX_SERVICES)
      print "Error: Maximum amount of services reached, #{@count}\n"
      return 0;
    end
    @count += 1
    @name_to_id[name] = @count
  end
end
Listing A.2: Extract of FrontProxy.rb (Simplified)

```ruby
def method_missing(name, *args, &block)
    trace_id = get_trace_id
    report_request(trace_id, name.to_s, args.to_s)
    reply = send_call(trace_id, name, *args, &block)
    report_reply(trace_id, name.to_s, reply.to_s)
    return reply;
end

private

def get_trace_id
    return 0 unless @tag_messages
    trace_id = Thread.current[:trace]
    if ((trace_id.nil?) || trace_id == 0)
        return @service.get_local_count;
    end
    return trace_id;
end

def send_call(trace_id, name, *args, &block)
    if (@tag_messages)
        if (@object.respond_to?(:__taggedcall__))
            return @object.__taggedcall__(trace_id, name, *args, &block)
        end
    else
        if (@object.respond_to?(:__timedcall__))
            return @object.__timedcall__(trace_id, name, *args, &block)
        end
    end
    #fall–through if its not a BackProxy
    return @object.__send__(name, *args, &block)
end
```
Listing A.3: Extract of BackProxy.rb (Simplified)

def __tracecall__(trace_id, method_name, *args, &block)
    Thread.current[:trace] = trace_id
    @object.__send__.(method_name, *args)
end

def __timedcall__(method_name, *args, &block)
    Thread.current[:trace] = @object.get_local_count
    @object.__send__.(method_name, *args)
end

def method_missing(method_name, *args, &block)
    trace_id = @object.get_local_count
    Thread.current[:trace] = trace_id

    # external call, thus call hasn't been logged at front: LOG here.
    report_request(trace_id, method_name.to_s, args.to_s)
    reply = @object.__send__.(method_name, *args)
    report_reply(trace_id, method_name.to_s, reply.to_s)
    return reply;
end

def respond_to?(symbol, include_private=false)
    if (symbol == __taggedcall__ || symbol == __timedcall__)
        return true
    else
        return @object.respond_to?(symbol);
    end
end

def report_request(request)
    if (@delay_reports)
        @mutex.synchronize {
            @requests << request
        }
        if (@max_req != 0 && @requests.size > @max_req)
            @background_thread.wakeup
        else
            @requests << request
        end
    end
end

def instant_reporting
    while (true)
        @central_gatherer.add_request(@requests.pop)
    end
end

def delayed_reporting
    while (true) do
        sleep(@sleep_time)
        if (@requests.size > 0)
            sending = []
            @mutex.synchronize {
                sending = @requests
                @requests = []
            }
            @central_gatherer.add_requests(sending)
        end
    end
end
Appendix B

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AIS: Automatic Identification System
APM: Application Performance Monitoring
AOP: Aspect Oriented Programming
ESB: Enterprise Service Bus
SOA: Service Oriented Architecture