



Master's Thesis

On Dependability Modeling in a Deployed Microservice Architecture

Author

Johan Uhle

June 25, 2014

Supervisor

Dr. Peter Tröger

Operating Systems and Middleware Group

I want to thank Peter Tröger for interesting me in the subject of dependability and sending me on the journey for this thesis. I want to thank all engineers at SoundCloud who answered my relentlessly naïve questions with admirable calmness and commitment. I especially want to thank Peter Bourgon for embedding me in his team and work, and Alexander Grosse for allowing me to execute my research with the company. I want to thank Nxtbgthng for hosting me in their office, when I finished writing this thesis. I want to thank Lea Voget for helping me with support and proof-reading, as well as Hannes Tydén for proof-reading.

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, June 25, 2014

(Johan Uhle)

Abstract

The microservice architectural style is a common way of constructing software systems. Such a software system then consists of many applications that communicate with each other over a network. Each application has an individual software development lifecycle. To work correctly, applications depend on each other. In this thesis, we investigate how dependability of such a microservice architecture may be assessed. We specifically investigate how dependencies between applications may be modeled and how these models may be used to improve the dependability of a deployed microservice architecture. We evaluate dependency graphs as well as qualitative and quantitative fault trees as modeling approaches. For this work the author was embedded in the engineering team of "SoundCloud", a "Software as a Service" company with 250 million monthly users. The modeling approaches were executed and qualitatively evaluated in the context of that real case study. As results we found that dependency graphs are a valuable tool for visualizing dependencies in microservice architectures. We also found quantitative fault trees to deliver promising results.

Zusammenfassung

Der “Microservice Architecture” Stil ist eine verbreitete Art um Softwaresysteme zu erstellen. Solch ein Softwaresystem besteht aus vielen Applikationen, welche miteinander über ein Netzwerk kommunizieren. Jeder Applikation hat einen individuellen Softwareentwicklungszyklus. Um korrekt zu arbeiten verlassen sich die Applikationen aufeinander. In dieser Arbeit untersuchen wir, wie Verlässlichkeit von solch einer “Microservice Architecture” bewertet werden kann. Im Speziellen untersuchen wir, wie Abhängigkeiten zwischen Applikationen modelliert werden und diese Modelle dann benutzt werden können, um die Verlässlichkeit einer produktiven “Microservice Architecture” zu verbessern. Wir evaluieren die Modellierungsansätze Abhängigkeitsgraph sowie qualitativer und quantitativer Fehlerbaum. Für diese Arbeit war der Autor in die Ingenieurgruppe von “SoundCloud” eingebettet, einem “Software as a Service”-Unternehmen mit 250 Millionen monatlichen Nutzern. Die Modellierungsansätze wurden im Kontext dieser echten Fallstudie ausgeführt sowie qualitativ ausgewertet. Als Resultate haben wir herausgefunden, dass Abhängigkeitsgraphen ein wertvolles Werkzeug zur Visualisierung von Abhängigkeiten in einer “Microservice Architecture” sind. Quantitative Fehlerbäume lieferten vielversprechende Resultate.

Contents

1	Introduction	1
2	Terminology	3
2.1	Dependability	3
2.1.1	Dependability threats	4
2.1.2	Failure semantics	7
2.1.3	Dependability means	7
2.1.4	Dependability attributes	8
2.1.5	Fault tree analysis	8
2.2	Software environment terminology	13
2.2.1	The microservice architectural style	13
2.2.2	Infrastructure systems	19
2.3	Summary	20
3	Case study	23
3.1	Architectural style	24
3.1.1	Technical details	26
3.2	Infrastructure systems	26
3.2.1	Deployment systems	27
3.2.2	Service Discovery	29
3.2.3	Telemetry	32
3.3	Summary	33
4	Constructing dependency graphs	35
4.1	Theory	35
4.2	Methods for construction	37
4.2.1	Manually creating a dependency graph	37

Contents

4.2.2	From service interface modules	43
4.2.3	From deployment configuration	47
4.2.4	From network connections	57
4.3	Discussion	63
4.4	Future work	65
4.5	Summary	66
5	Constructing qualitative fault trees	67
5.1	Theory	67
5.1.1	Dependency graph requirements	67
5.1.2	Failure semantics	68
5.1.3	Construction algorithm	69
5.1.4	Related work	70
5.2	Case study execution	71
5.3	Discussion	77
5.4	Future work	78
5.5	Summary	79
6	Constructing quantitative fault trees	81
6.1	Via source code metrics	82
6.1.1	Lines of code	83
6.1.2	Relative code churn	84
6.2	Via historical availability data	87
6.2.1	Theory	87
6.2.2	External heartbeat measurements	89
6.2.3	Production traffic measurements	95
6.2.4	Discussion	98
6.3	Discussion	100
6.4	Summary	103
7	Discussion	105
8	Summary	107
	References	109

List of Figures

2.1	The dependability tree by Laprie [7]	4
2.2	The elementary fault classes by Laprie [5].	6
2.3	Chain of dependability threats	7
2.4	Example qualitative fault tree	11
2.5	Entity-relationship model of the terms defined in this chapter . .	15
3.1	Visualization of the architecture style evolution from monolithic architecture style to microservice architecture style. Boxes represent applications and directed edges represent service dependencies. Size of the “monolith” represents its significance in the architecture.	25
3.2	Visualization of the request flow from client to service through the Bazooka load balancer. Boxes represent machines with the domain name and port number as names. Solid edges represent the request flow. Dotted edges represent potential opportunities for different request flows, which would still yield the same results.	31
4.1	Dependency graph: example	36
4.2	Dependency graph: Manual annotation from case study with masked application names. Grey boxes are internal applications. Blue boxes with dotted outline are external applications.	41
4.3	Dependency graph generated from deployment configuration and executed with automatic reverse service discovery in the case study. The application identifiers are masked with incremented numbers.	52

List of Figures

4.4	Dependency graph generated from deployment configuration and executed with manual reverse service discovery in the case study. The application identifiers are masked with incremented numbers.	53
4.5	Dependency graph: Graph from network connections, resulting from execution in the case study. Application names are masked.	61
5.1	Example dependency graph for fault tree algorithm example. Root vertex is vertex “A”.	71
5.2	Visualization of the algorithm for transforming a dependency graph to a qualitative fault tree. This example is based on the dependency graph in Figure 5.1	72
5.3	Subgraph for vertex “73” from case study figure Figure 4.2. The outgoing edges for vertex “5”/application “github.com/soundcloud/soundcloud” have been removed. The vertex names have been unmasked to carry their original application identifiers.	74
5.4	Qualitative fault tree for vertex “73”/application “github.com/soundcloud/soundcloud” following the dependency graph from Figure 5.3.	75
5.5	Qualitative fault tree for vertex “26” from case study dependency graph figure Figure 4.2 (cropped)	76
6.1	Historical availability: Example of time series for a time period of 12 minutes with 1 representing “up” and 0 representing “down” for the respective time step	88
6.2	Heartbeat measurement: Duration of each measurement step for each application	90
6.3	Heartbeat measurement: Summarized HTTP request status codes for each measurement step for application <i>soundcloud</i>	91
6.4	Heartbeat measurement: Summarized HTTP request status codes for each measurement step for application <i>authenticator</i>	92
6.5	Heartbeat measurement: Summarized HTTP request status codes for each measurement step for application <i>threaded-comments</i>	93

List of Figures

6.6	Existing measurements: Summarized HTTP response status codes on <i>HAProxy</i> for application <i>soundcloud</i> . Please note that there are two y-axes	96
6.7	Existing measurements: Summarized HTTP response status codes for application <i>authenticator</i>	97
6.8	Existing measurements: Summarized HTTP response status codes for application <i>threaded-comments</i>	98

List of Tables

2.1	Fault tree elements	10
4.1	Approximate percentual statistics over languages of relevant applications in our case study.	44
4.2	Distribution of configuration cases for the dependency graph generated with manual reverse service discovery in Figure 4.4. . .	55
5.1	Shortened application identifiers of subgraph for vertex “73” shown in Figure 5.3.	73
5.2	Description of applications of subgraph for vertex “73” shown in Figure 5.3.	74
5.3	Sizes of dependency graphs and the generated fault trees	78
6.1	Lines of code calculation in the case study with accompanying failure probabilities.	84
6.2	Relative code churn in the case study calculated over several time periods.	85
6.3	Relative code churn in the case study: TOP event failure probabilities after the fault tree for several time periods.	86
6.4	Failure probabilities for application deploys measured via heart-beat measurements from April 18 2014 to April 24 2014 with <i>down</i> threshold 5%. The <i>mission time</i> for the failure probability calculation was 720 minutes (12 hours).	92

List of Tables

- 6.5 Failure probabilities for applications measured from production traffic from April 18 2014 to April 24 2014. The *mission time* for the failure probability calculation was 720 minutes (12 hours). Note that we scaled up the datapoints for *soundcloud* to make up for the low granularity provided from the measurement. 99

Listings

2.1	Example for a service discovery mapping from service discovery key to service locations with IP address and port number	20
3.1	Service discovery via Glimpse; Query example	32
4.1	Dependency graph: the manual annotation file in JSON format for the threaded-comments application	40
4.2	Shortened Gemfile from an application of the case study	45
4.3	Shortened list of imported packages of the thread-comments application	46
4.4	Example configuration of an application in Bazooka	50
4.5	Bazooka HAProxy configuration file excerpt for the threaded-comments application	50
4.6	Example configuration of an application in Bazooka	54
4.7	Sanitized example output from netstat. Schema is: local_address foreign_address process_id/process_name	58
4.8	Cgroup information example from the proc virtual filesystem retrieved via <code>cat /proc/20471/cgroup</code> in our case study . . .	59
4.9	Glimpse zonefile example excerpt	60

1 Introduction

In recent years the *microservice architectural style* has become popular for building software systems. Especially companies with “Software as a Service” products like Netflix [1], Twitter [2] and Amazon [3] utilize the style. A microservice architecture realizes a single application as many small individual applications which communicate over a network. Each application has its own software development lifecycle. This decoupling allows many small teams to work on individual applications. All applications then converge to deliver one software product to the users, who perceive the whole architecture as one single system.

One of the objectives of a software system is to provide a dependable and failure-free experience to the user. A system may never be “completely dependable”. Instead dependability is a trade off between a dependability goal and the cost needed to achieve it. Defining and measuring these dependability goals requires structured approaches.

Given that a microservice architecture consists of many individual applications, these may also fail individually. This leads to the questions of how these individual failures manifest themselves and how they influence other applications in the architecture. How do failures propagate through the systems? How do they influence the end user experience? We believe that understanding and modeling the dependencies between applications is a crucial part of understanding the dependability of a microservice architecture.

In this thesis, we propose the concept of *dependency graphs* for modeling the dependencies between applications in a microservice architecture. These graphically visualize an architecture with its applications and their interrelations. Based on the dependency graphs we introduce an algorithm for constructing *fault trees*. For decades fault trees have been used to model the dependability of

systems. This work is the first time they are used in the context of microservice architectures. We construct qualitative fault trees and investigate quantifying based on application failure probabilities.

While investigating this work the author was embedded in the engineering team of “SoundCloud”, a “Software as a Service” company in the space of social networking and user-generated content, and a product with 250 million monthly users. All proposed methods were executed and qualitatively evaluated in the context of that deployed microservice architecture.

The remainder of this thesis is structured as follows:

In chapter 2 *Terminology* we introduce the terminology of this work by describing related work in the fields of dependability research and fault trees. We then introduce our own terminology model for defining the microservice architectural style.

This view is then further refined in the following chapter 3 *Case study*, where we describe the architectural evolution of the case study system and present infrastructure systems.

In chapter 4 *Constructing dependency graphs* we introduce the concept of dependency graphs and discuss the four approaches we developed for generating dependency graphs from a deployed microservice architecture.

From a dependency graph we then construct a qualitative fault tree. We introduce the algorithm we developed for that in chapter 5 *Constructing qualitative fault trees*.

We use the qualitative fault trees as basis for calculations with quantitative fault trees. In chapter 6 *Constructing quantitative fault trees* we propose four approaches to quantify failure probabilities of individual applications and discuss them in the context of the case study.

In the closing chapters chapter 7 *Discussion* and chapter 8 *Summary* we conclusively discuss and summarize our findings.

2 Terminology

In this chapter we introduce the terms and concepts used in the remainder of this work. In section 2.1 *Dependability* we introduce the background for dependability theory from related work. In section 2.2 *Software environment terminology* we introduce our terminology model for the microservice architectural style. Since we did not find a comprehensive model in related work, we define this terminology model ourselves.

2.1 Dependability

In this section we introduce quantitative and qualitative fault trees as a way to model dependability of a system. As a basis to that, we introduce the concept of dependability in computer systems after Laprie. Building on that, we describe dependability threats, chain of dependability threats and fault tolerance.

Dependability is defined by Laprie [4] as a “property of a computer system such that reliance can justifiably be placed on the service it delivers”. Laprie also provides an alternative definition in [5]: “dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable”. Both definitions speak of a service, which Laprie [4] defines as: “The service delivered by a system is its behavior as it is perceptible by its user(s); a user is another system (human or physical) which interacts with the former”. Following this definition, we can see that dependability is defined in the context of how the user experiences the system from the outside.

In [6] Rausand et al define three main branches of dependability: *hardware*, *software* and *human*. In this work we focus on software dependability, but will rely on concepts from the world of hardware dependability. The concept of human dependability is not subject of this work.

To further describe dependability, we use the dependability tree in Figure 2.1 after Laprie [4]. Following that figure, we see dependability holding the aspects *threats*, *means* and *attributes*. We will explain these further next.

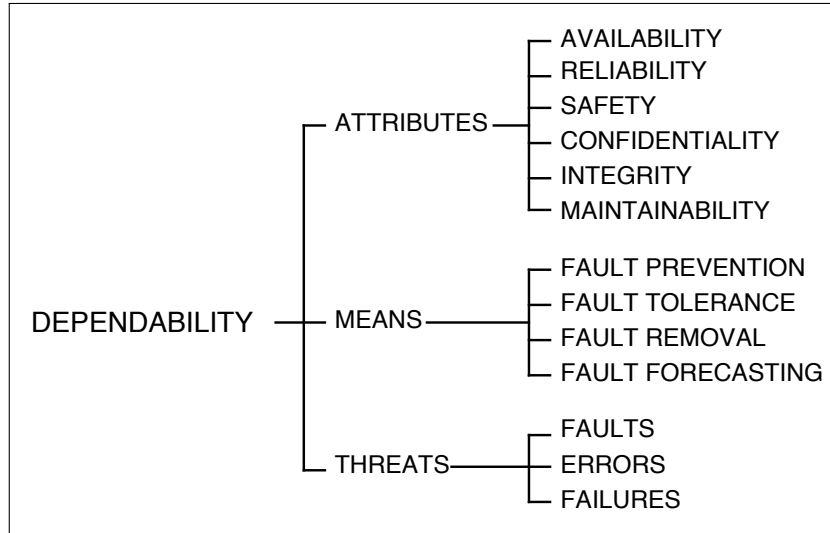


Figure 2.1: The dependability tree by Laprie [7]

2.1.1 Dependability threats

The user has expectations over the service to be delivered by a system. These expectations may be formalized in a *specification*. The delivered service can be split in two main categories: **correct** service and **incorrect** service (after Laprie [5]). The specification may include both, even though it is more common for the specification to only include the expectations of correct service.

An example for a specification is a description of a service using the HTTP protocol [8]. The HTTP protocol specifies a request/response style message flow between a client and a server. The specification may include the location of the HTTP resources, how requests may be formed and which responses are to be expected. The specification may also include expectations for incorrect service. Each HTTP response message includes a numerical status code. If the server can not fulfill a request due to an error in itself or in its dependencies, it may use a status code equal to or higher than 500 to signify incorrect service to the client.

In the event of incorrect service being delivered, we call that a service failure, or short **failure**. The failure occurs at the moment when the user experiences incorrect service. There are different ways of how failures manifest themselves. We will reflect on some cases in subsection 2.1.2 *Failure semantics*.

The reason for a failure to occur is the system being in an erroneous state, or short **error**.

This means that the internal state of the system allowed a failure to occur. It is to be noted that an error is the state of the system and not an event. Only in the event of the user requesting the system's service does an error turn into a failure.

An example for an error is a server that has crashed. It is then in a state where it can not answer any client requests. The system can be in this state for a period of time, but a failure event only occurs in the moment when the client executes a request against the server.

A **fault** is the "adjudged or hypothesized cause of an error" [4]. Given that a fault is active, it is the cause of an error, thus there is an implication from fault to error. It is often hard to do that deduction the other way around, thus to identify the cause of an error as an exact fault. As engineers we may not be able to detect the faults themselves, but only the errors they have caused¹. Therefore we use the qualifiers *adjudged* and *hypothesized* in the definition.

An example for a fault is a software bug which manifests itself as a single line of code and causes the server to crash when it is executed. When the software bug is in the software's codebase we call it a *dormant* fault. As long as the line of code is not executed, the fault remains *dormant*. It is activated the moment where the line of code is executed and then becomes an *active* fault which subsequently puts the executed software into an erroneous state.

Faults can be classified more precisely. Figure 2.2 shows the elementary fault classes after Laprie. For this work, we are interested in classifying faults by *system boundary*. A system boundary is defined by Laprie [5] as "the common frontier between the system and its environment". The environment are all other systems that are not the system of interest. A system itself may contain other systems (usually called sub-systems or components). When classifying faults, we separate between **internal** and **external** faults.

An *internal* fault originates within the system boundary. To illustrate this with an example we take a computer as a system. Internally, the computer consists of many components as sub-systems. The failure of one of the components (e.g. the memory) is a fault within the system, thus an *internal* fault.

An *external* fault originates from outside the system boundary. To illustrate, we again take the computer as an example. This time, we see the computer components CPU and memory as two separate systems. The CPU's correct service relies on the memory to work correctly. The failure of the memory may therefore be a fault to the CPU. Since

¹For reference see section 3.3. in [9]

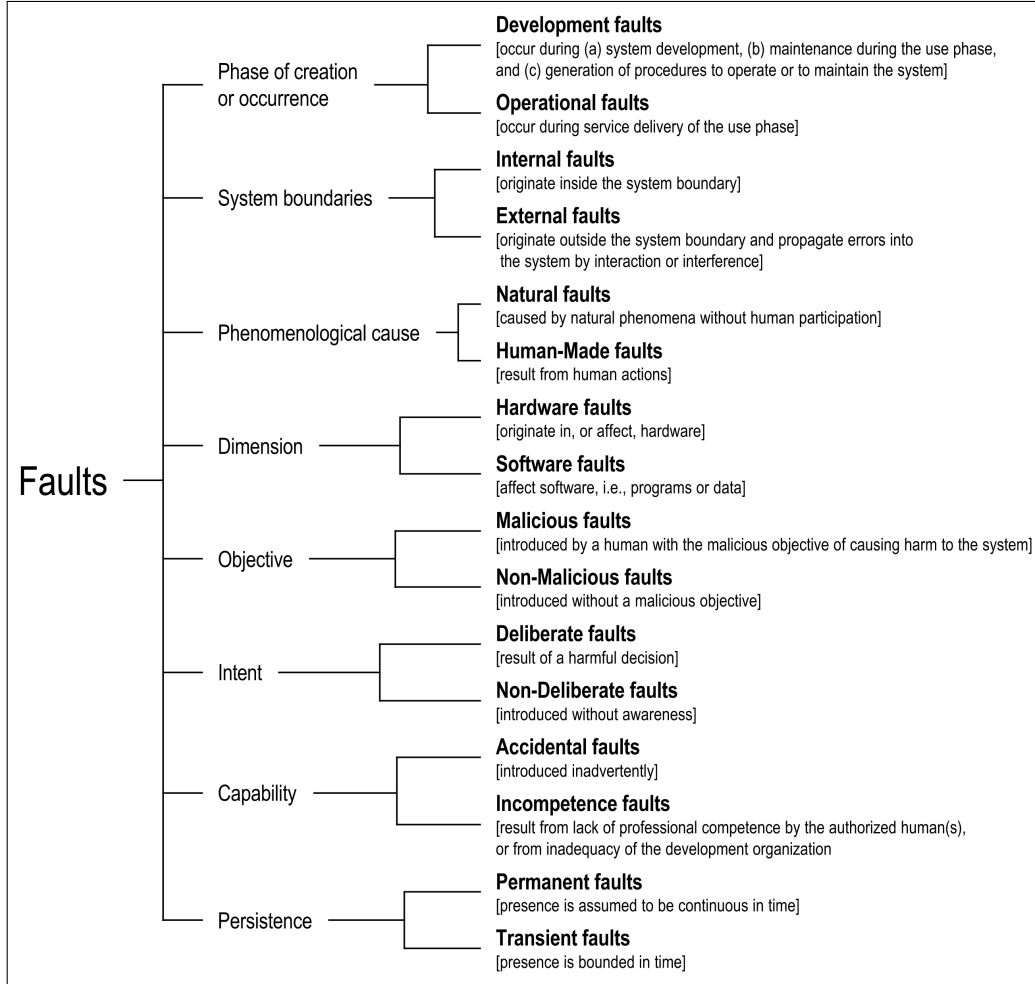


Figure 2.2: The elementary fault classes by Laprie [5].

the memory is not in the CPU, this is an *external* fault to the CPU.

This hints to an interesting property, when looking at larger networks of systems that rely on each other: The failure of one system becomes the external fault of another system, which relies on it. We call this the **chain of dependability threats** (after the “chain of threats” by Laprie [5]).

Figure 2.3 visualizes the relation of fault-error-failure, as well as the chain of dependability threats. The arrows depict causation, from cause to effect. The boxes depict the system boundaries. *System B* relies on *System A*, therefore the failure of *System A* may be seen as a fault in *System B*.

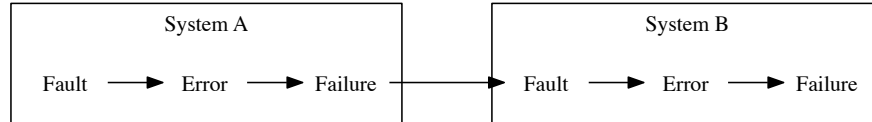


Figure 2.3: Chain of dependability threats

2.1.2 Failure semantics

Failure semantics denote how a system and specifically its service to other systems behave in case of a fault (after Knight [9] section 3.8). They therefore set expectations for correct and incorrect service such that other systems may be able to guard against these.

One specific model for failure semantics was proposed by Cristian [10]. His model is based on the notion of services (take input, execute operation, deliver output), servers who implement these services and a “depends upon” relation, in which the correctness of one server depends on the correctness of another server. The model then distinguishes between the following types of failures, notable within the “depends upon” relation:

Response failure The response is incorrect. It could be in a wrong format or carry incorrect data.

Timing failure The response happens outside of a specified time interval

Omission failure No response is ever delivered for a request

Crash failure After an initial omission failure all subsequent requests exhibit omission failures as well until the system is restarted

We will build on this model later, when we propose our algorithm for constructing fault trees in chapter 5 and explain our concrete assumptions regarding the failure semantics of the systems.

2.1.3 Dependability means

Dependability means are ways to limit the impact of dependability threats on the dependability of a system. Following the list of means from Figure 2.1, they include

fault prevention, fault tolerance, fault removal and fault forecasting. All means focus on faults and ways to mitigate them, in order to not have them propagate to failures.

Dependability means are not in the scope of this work. For more information on this topic we recommend the work of Knight [9].

2.1.4 Dependability attributes

The attributes of dependability define the different aspects, as to which the dependability of a system can be evaluated. The dependability means are there to assure these attributes against the dependability threats. In previous Figure 2.1 six attributes are mentioned. In the scope of this work we are only interested in *reliability* and *availability*:

Reliability is defined by Laprie [5] as the “continuity of correct service”. A more concrete definition is delivered by Knight [9] as “ $R(t)$ = probability that the system will operate correctly in a specified operating environment up until time t ”. We assume that at the time $t=0$, the system was operating correctly. $R(t)$ then denotes the probability that in the interval $[0,t]$ the system has not suffered a failure. An underlying assumption of this reliability definition is that the system is not repairable: given that a failure occurred, the system stays in an incorrect state and may never return into a functioning correct state again.

Availability is defined by Laprie [5] and Goloubeva [11] as “readiness for correct service”. In contrast to reliability, availability allows for systems to be repairable after a failure occurred. Thus, a system might fail, be repaired and then deliver correct service again. During the time of repair we assume the system to deliver incorrect service.

Please note that in later subsection 6.2.1 we introduce more theoretical background, which is utilized when calculating failure probabilities from historical availability.

2.1.5 Fault tree analysis

Introduction

When we talk about dependability in systems, we eventually aim to prevent failures. For that we have to analyze the system. To do this in a structured way, many methods exist. Three notable are *Failure modes, effects and criticality analysis (FMECA)* [12],

Reliability block diagrams ([6] section 3.10.), and fault tree analysis, which we will treat further in this section. All methods share the common assumption that faults happen. We may then separate faults into two categories: *anticipated* faults and *unanticipated* faults ([6] section 4.1). Against anticipated faults we are able to deploy dependability means. But against unanticipated faults we can by definition not defend, thus these are the most likely to propagate to failures. Therefore, one of the goals of structured methods for dependability analysis is identifying these faults as comprehensively as possible, so that they can be anticipated.

Next, we will introduce fault tree analysis further. It originated in the Bell Telephone Laboratories in 1962 for evaluating systems of the *Minuteman* missile program [13]. It also found application in nuclear power stations [14] and aerospace missions [15].

Definition

A **fault tree** is defined by Rausand et al [6] as a “logic diagram that displays the interrelationships between a potential critical event [...] in a system and the causes for this event”. Fault tree analysis is the process of constructing such a fault tree. A fault tree may be qualitative or quantitative, which we both will describe next.

Qualitative fault trees

A fault tree analysis is started by defining the system failure to investigate. We call that the *TOP event* of the fault tree. From that event, all events that might contribute to its occurrence are identified as new events. They are connected to the *TOP event* via the *logic gates AND-gate* or *OR-gate*. This process is continued for all new events recursively, until a sufficient level of detail is reached. Thus, we acknowledge that any event can likely be seen as consisting of a more detailed fault tree, but we make the design decision for the fault tree to stop investigating this greater detail. Since the analysis method is from top down, we call it “deductive”. The leaves of the resulting tree are called *basic events*. If an event is neither *basic* nor *TOP event*, it is called *intermediate event*.

Table 2.1 shows all the fault tree elements with their graphical symbols from the Fuzzed Editor [16]². More elements for fault trees exist (e.g. explained in [14] and [15]), but for our case the introduced elements are sufficient.

²All fault trees in this work were generated using the Fuzzed Editor [16] and therefore share the graphical symbols introduced here

Table 2.1: Fault tree elements






Name	Symbol	Description
TOP event		The TOP event, is the event whose causes are investigated with the fault tree. A fault tree may only have one TOP event.
AND-gate		An AND-gate connects many input events with one output event. The output event occurs only when all input events occur.
OR-gate		An OR-gate connects many input events with one output event. The output event occurs if any of the input events occurs.
Intermediate event		An intermediate event may be used as event between logic gates.
Basic event		A basic event represents the occurrence of a specific failure event. A basic event may have a failure probability assigned.

Figure 2.4 shows an example qualitative fault tree. It can be seen that there is one TOP event. “Intermediate Event” occurs if either “Basic Event 2” or “Basic Event 3” or both occur. “TOP event” occurs if “Basic Event 1” and “Intermediate Event” occur.

When a qualitative fault tree exists, it may be analyzed for possible combinations of basic events that result in the TOP event (called “cut sets”, for example explained by Rausand et al [6] section 3.6.4.). For example this analysis then allows for the identification of single point of failures in the system.

Another value of fault trees is running the fault tree analysis itself. The analysis process involves gathering experts and addressing the problem of dependability of a system from many different viewpoints. During the process, many problems and maybe even their respective solutions may become apparent. The resulting fault tree may then be used as basis for discussion, for example to communicate the current state of the system or discuss potential changes.

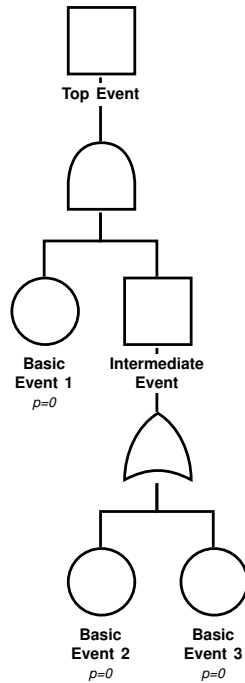


Figure 2.4: Example qualitative fault tree

Quantitative fault trees

Quantitative fault trees allow for calculating the probability of the TOP event based on the probabilities of the basic events. A quantitative fault tree analysis may use the same fault tree structure as the qualitative fault tree. Next, we will explain how such a calculation may be carried out. We base the following calculations after Limnios [17] chapter 5.

We operate under the following assumptions:

- Every basic event may exist in the fault tree only once. Repeated basic events are not allowed.
- The basic events are independent events.
- The calculations are done for repairable systems.

Furthermore we assume that all basic events have a failure probability. We may now calculate the probability per logic gate: We assume A and B to be the logic gate inputs with $\Pr(A)$ and $\Pr(B)$ being their respective failure probabilities. E is the logic gate

output, with the resulting failure probability $Pr(E)$.

Equation for an AND-gate:

$$\begin{aligned} E &= A \cap B \\ Pr(E) &= Pr(A \cap B) = Pr(A) * Pr(B) \end{aligned}$$

Equation for an OR-gate:

$$\begin{aligned} E &= A \cup B \\ Pr(E) &= Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A) * Pr(B) \end{aligned}$$

Both equations may be extended for more than two inputs. To calculate the TOP event probability, we may use the “recursive inclusion-exclusion method” (after Limnios [17] section 5.5.1): We start at the TOP event and resolve the immediate logic gate. We then recursively resolve all input events after the same method. Following is the equation for earlier example fault tree 2.4:

$$\begin{aligned} Pr(TOPEvent) &= Pr(BasicEvent1 \cap IntermediateEvent) \\ &= Pr(BasicEvent1) * Pr(IntermediateEvent) \\ &= Pr(BasicEvent1) * (Pr(BasicEvent2 \cup BasicEvent3)) \\ &= Pr(BasicEvent1) * (Pr(BasicEvent2) + Pr(BasicEvent3) - \\ &\quad Pr(BasicEvent2) * Pr(BasicEvent3)) \end{aligned}$$

As we can see, the resulting equation only has probability operators and therefore the probability of the TOP event is resolvable, given that each basic event has a probability assigned.

The major value we see in creating quantitative fault trees is for comparing changes to basic event probabilities and respective impact on the TOP event probability.

2.2 Software environment terminology

In this chapter we propose our software environment terminology model describing the microservice architectural style. We present the individual terms and the connections between them, and reflect on some related work. Extending the architectural model, we also introduce related infrastructural systems, which are commonly utilized when realizing such an architecture.

2.2.1 The microservice architectural style

Our terminology model is similar to the field of “Service-oriented architecture” (short: SOA) (e.g. described by Krafzig in [18] and standardized by OASIS in [19]). We did not find the SOA model to be a fitting description for the microservice architectural style, which is why we propose our own model here. This chapter is not an exhaustive survey over the usage of the introduced terms, but rather sets the stage for the remainder of our work.

Software architecture

To define *software architecture*, we follow the model from Fielding [20]:

“A software architecture is an abstraction of the run-time elements of a software system during some phase of its operation” [20]. Run-time elements are usually separated into components and connections. Since a *software architecture* is an abstraction, the exact definition of what constitutes a component and a connection depends on the level of detail and purpose of the current view on the software system. For example in this work we will often use applications as the granularity level for components and application dependencies as the granularity level for connections.

“A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture” [20]. The different levels manifest themselves most prominently in the fact that each component in an architecture may consist of another architecture in itself. Continuing with the example from above, one application might in itself have an architecture consisting of its deployed services, or how its internal code modules relate to each other. The phases of operation may be seen as changing over time (for example the architecture changed between an hour ago and now) or states (for example the architecture changed when a new application was introduced).

Software architecture style

Software architecture styles try to categorize common aspects of *software architectures*. These aspects might be similar patterns in structural organization, the same granularity of components and connectors or similar constraints on how these work together as one system [21] [22].

Software as a Service

Software as a Service (SaaS) is a software delivery model, in which the software code, computing resources and user data are managed by a provider. The software is remotely accessed by the consumer via a thin client, most commonly a web browser [23].

Traditional software delivery models involved shipping executable software to the users, who then operated them on their own computing resources [24]. Since SaaS software is operated by the vendor, it is possible to release software at a higher rate, sometimes several times in a day [25] [26].

In this work, we are concerned with multi-user SaaS, thus all users use the same computing resources supplied by the vendor, they have the same service level agreements and limited customization options [27] [28] [29]. One promising architectural style for building a multi-user SaaS is the *microservice architecture style*, which we will introduce next.

Microservice architectural style

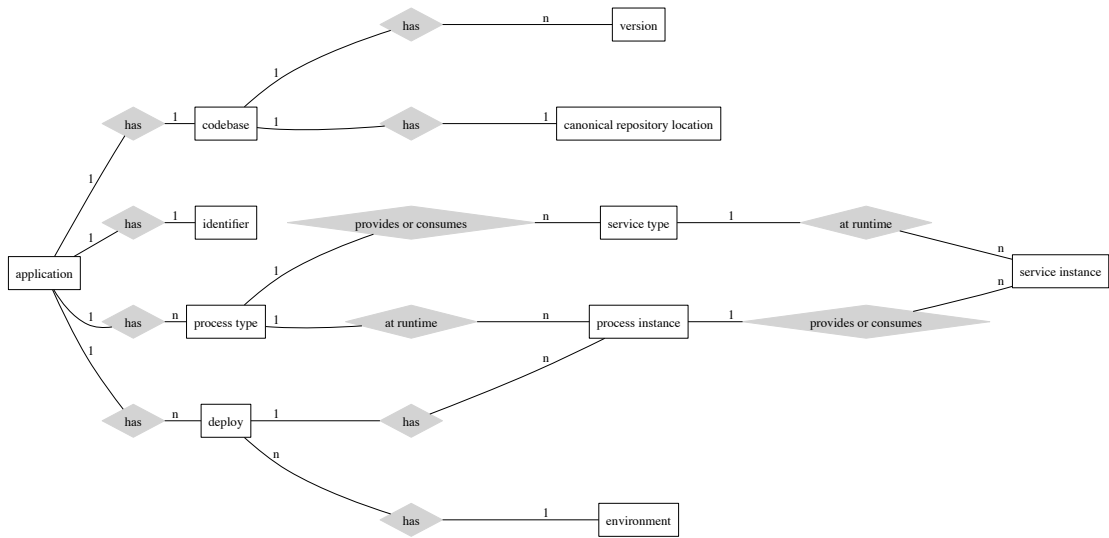
Fowler et al [22] define the *microservice architecture style* as “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API”. In our case, the “single application” is the *SaaS* as experienced by the user.

Microservice architecture shares concepts with the “Service-oriented architecture” (SOA) style, but we decided to not use the term SOA directly due to the ambiguity of its meaning (as described by Fowler et al [30] and Partridge et al [31]).

The terms following next are all to be seen in the context of microservice architecture, and therefore define it further.

Application

Figure 2.5: Entity-relationship model of the terms defined in this chapter



An *application* is the implementation of capabilities. Capabilities manifest themselves as having a real-world effect [31]. Usually the users of an application are able to observe its real-world effects during operation.

The following technical descriptions of *applications* are related to the observations made by Wiggins [32] as part of “12 factor applications”, but have been customized by us as core part of the terminology model, especially regarding the interrelation to other concepts. We visualize the interrelation of the described concepts in Figure 2.5.

Every application has exactly one *codebase*. The codebase may consist of files that are required to execute the application (e.g. source code) as well as meta data (e.g. documentation). The codebase is tracked in a source code version control system like Git [33] or Subversion [34]. We assume that within that system there is a canonical location for the codebase repository. Within that repository, the codebase might exist in different versions. We assume that for every repository, there is always a canonical version that represents the current state of development.

When an application is executed at runtime, its concrete representation are *processes*. One application may have many process types, each of which may have many process instances at runtime. We explain processes further below.

Every application has at least one identifier, which is unique in the current scope of investigation, which in our cases here is the organizational boundary of the company.

We distinguish between *internal* and *external* applications, based on the boundaries of an organization. The development of an *internal* application is driven within the organization. The development of an *external* application is driven externally to the organization. We assume both application types to be executed within the organization, thus on the machines under the control of the organization. An *external* application may also be executed outside of the organization if specified (e.g. services provided by other SaaS platforms).

Another categorization of applications we propose is between *client-facing* and *infrastructure* applications:

Client-facing applications serve a *client-facing* functionality. Thus, if such an application would be removed, the clients (and the human users of these) would notice a lack of functionality. Often these applications are very specific to the business they support.

Infrastructure applications support the *client-facing* applications in their operation, but in themselves do not provide a client-facing functionality. Thus, if the infrastructure application would be removed, the clients (or human users of these) would not notice a lack of functionality (given that the *client-facing* applications were still able to operate). We will discuss infrastructure applications further in the following subsection 2.2.2 *Infrastructure systems*.

In our work, we focus on *client-facing* applications. Forth going when we talk about “applications”, we implicitly exclude *infrastructure* applications.

An application may have many *deploys*, as explained next.

Deploy

A *deploy* of an application consists of the application’s process instances executed in a specific environment. It may also be referred to as an application at runtime. But since an application itself is an abstract entity, the concrete representation of it at runtime are its processes instances.

Every deploy is executed within a specific environment. Common names for such environments are “production” or “staging”. For each environment, a deploy might be configured differently, for example with regards to runtime parameters like queue lengths and timeout thresholds, or the location of services to consume.

The deployment of an application is often facilitated by deployment systems, which we introduce later in subsection 3.2.1 *Deployment systems*.

Process

A *process* is the concrete representations of an application at runtime. One process always belongs to one application. One application may have many processes. A process may be seen from two viewpoints: as process instance and as process type.

A *process instance* exists during execution, where it might use resources like CPU time, memory, files or I/O devices. The implementation details of how a process instance is managed during runtime is not of interest for our work, but we assume that its resources and lifecycle are managed by an operating system, for example as described by Silberschatz et al [35].

A *process type* is a logical entity embodying the opportunity to execute process instances of it. A process type usually manifests itself in some way in the source code of an application, for example by being a dedicated software module.

Every process instance always has exactly one process type and one deploy, whereas a process type may have many process instances, which in may even belong to different deploys. When we speak of “a process” it embodies both concepts of *process instance* and *process type*, which may be used interchangeably depending on the current context of discussion.

A process may consume and provide many services, as explained next.

Service

A *service* provides access to the capabilities of an application to other applications via a network. The possible ways of access are defined through a service interface. A service interface might be standardized in a machine-readable format like a WSDL definition [36], or just be defined through the implementation itself.

Services are implemented as processes. Just as with processes, we may speak about service types and service instances. Since each service may be consumable via a network, a service also needs to be addressable. Therefore each service instance may have a URI, via which it may be consumed by other processes.

If a process exposes its functionality via a service interface, it provides a service. If a

process starts a connection to a service, it consumes that service.

The access to services is facilitated as machine-to-machine interaction over a network, as described next.

Service communication

Service communication happens between a service consumer and a service via a network connection, over which they may exchange messages.

We assume that service consumer and service are interoperable over an agreed protocol stack which should adhere to the standards set by the Internet protocol suite [37].

We do not assume anything on the communication pattern. Examples might be request-response [38] or publish-subscribe [39]. The service provider is always the process that accepts the connection. The service consumer is the process that established the connection.

We do not assume anything on the locality of the service and the service consumer. They might be on the same physical machine, in different datacenters, or within different organizations.

Service dependency

A *service dependency* is the fact that a process is the consumer of a service. The service dependency is always directed from service consumer to service provider.

Given that a service dependency exists, it may also be noted with the related applications (as either service dependency or application dependency).

Please note that a service dependency is not to be confused with a shared library dependency: Both concepts have in common that the dependency is to another software component. These components are often developed in a different organizational context than the program that depends on it. This manifests itself in decoupled software life cycles. The difference between both concepts is that a shared library dependency is executed in the same context as the program (e.g. the same process) and usually used via an in-memory function call. On the other hand, a service dependency is executed in another context (e.g. another process, machine or datacenter) and therefore the communication may not happen via in-memory function calls, but through mechanisms like remote procedure calls [40] or web services [36].

Machines

A *machine* consists of a CPU, memory, permanent storage and an operating system. Every machines is connect to a network and may be addressable and accessible over it. A machine might be physical or virtual, but is always under the full control of the organization. A machine provides the execution context for processes and might be used by *client-facing* and *infrastructure* applications alike.

2.2.2 Infrastructure systems

When building a microservice architecture, certain types of systems are commonly used to support its realization. We call these *infrastructure system*, since they form the infrastructure for developers to build *client-facing* applications on. In this section we will describe three types of infrastructure systems: *deployment*, *service discovery* and *telemetry* systems. We introduce their concepts here and will describe the concrete implementations we found for them in our case study in later chapter 3 *Case study*.

Deployment systems

For an application to be actually used at runtime, it needs to be deployed. A deployment procedure usually starts with the application's codebase and ends with process instances running.

The foundation for each deployment system is that there are available computing resources under control of the organization. The deployment system then matches available resources with processes of application deploys and cares for starting the process instances. This might include building the application, distributing the build artifacts to the correct machines, setting up the runtime and configuration environment as well as starting and supervising the processes at runtime. This implies that the deployment system has knowledge about existing applications and about currently running process instances.

It is possible to execute a deployment procedure manually without utilizing a dedicated deployment system. Engineering experience has shown that it is desirable to automate deployment procedures as much as possible in order to increase development speed and lower potential for failure. Humble et al [26] motivate and describe this subject in detail.

Service discovery

When a process wants to consume the service of a certain application or deploy, it has to know the service's location. Service discovery answers this question, mapping an application name (and possible more qualifying information like deployment environment or datacenter) to the location of the services. Listing 2.1 shows an example of such a mapping. Since a deploy might have many process and service instances, service discovery might return the locations of all of these. Since we assume all service communication to happen via the TCP/IP protocol stack, the eventual format of a service locations is always an IP address and a port number.

```
"myapplication" -> [ "10.20.30.1:123", "10.20.30.2:321" ]
```

Listing 2.1: Example for a service discovery mapping from service discovery key to service locations with IP address and port number

Telemetry

Telemetry is defined in [41] as “the process of using special equipment to take measurements of something (such as pressure, speed, or temperature) and send them by radio to another place”. Telemetry systems are the basis for reasoning about the current state of the system, since they collect measurements and present them to engineers in a queryable format. Telemetry systems are also used for setting up alerting. Since in a microservice architecture process instances are by definition distributed over many machines, a dedicated telemetry system is needed for effectively providing engineers with the relevant information about the systems state.

2.3 Summary

In this chapter, we first introduced aspects of dependability theory based on the “dependability tree” by Laprie (visible in Figure 2.1). We described the dependability threats *fault*, *error* and *failure* and how these relate to each other. Based on the categorization of faults into *internal* and *external* faults, we mentioned the *chain of dependability threats* as a model to understand propagation of failures between interdependent systems. As attributes to qualify dependability we described *reliability* and *availability*.

Afterwards we introduced qualitative and quantitative *fault trees* as ways to model dependability.

To describe the microservice architectural style further we proposed an own terminology model. At the core it defines what an *application* is and how applications may depend on each other through services. The interrelation of the introduced terms can be seen in Figure 2.5. We ended the chapter by introducing infrastructure systems that support the realization of microservice architectures.

3 Case study

In this chapter we describe the case study and explain how the terminology model we proposed in previous section 2.2 *Software environment terminology* relates to it. We restate our problem in the context of the case study, highlight the evolution of architecture in the case study and introduce the concrete implementations of infrastructure systems, which are relevant to our work in later chapters.

In the first quarter of 2014, the author worked with SoundCloud [42] in the context of this work. SoundCloud produces one product, which they describe as “the world’s leading social sound platform where anyone can create sounds and share them everywhere” [43]. In October 2013 it had 250 million monthly active listeners [44]. On May 9th 2014, the Alexa Global Rank of the product was 166 [45]. In April 2013, the company had 190 employees in four offices (Berlin, London, San Francisco, Sofia) with 80 engineers mainly located in the Berlin office [46].

In the deployed microservice architecture of the case study we found hundreds of applications and thousand of processes deployed in the productive environment. One of the problems that became apparent to us while interviewing engineers as well as through observing the day-to-day work in the company was that there did not exist a holistic understanding of how applications and processes relate to each other. Often not even the immediate dependencies of applications were clear to engineers, manifesting themselves in the two questions “Which applications does my application depend on?” and “Which applications depend on my application?”. Resulting from these observations, we especially found the question of how failures propagate through the system to be an interesting angle for research.

Before we discuss our solutions to these problems in the following chapters, we will discuss the technical environment of the case study more closely.

3.1 Architectural style

We separate the product into two big categories: *clients* and the *platform*. The clients are software that is executed within the realm of the human users. This software may be the website on *soundcloud.com* running in a web browser, native applications (mostly on mobile devices) or consumers of the public REST API [47] [20]. The client software may be developed by the company or by third parties. All clients interact with the platform via APIs based on the HTTP protocol [8]. Usually the clients functionality is tightly bound to the functionality provided by the platform. In fact they might provide no functionality at all if there is no connectivity to the platform. Clients are not the focus of this work and will not be represented in further investigations. This is based on the notion that due to the context of the execution of clients we do not see them as part of the microservice architecture, even though they technically consume services provided by the microservice architecture. Figure 3.1 shows how clients and platform relate to each other.

The *platform* is implemented as a *multi-user Software as a Service*. Thus, the software, the runtime resources and the user data are managed by the company. The platform is implemented as *multi-user*, meaning that all users use the same computing and data storage resources.

The development of the platform started in 2007. Since then, it has been incrementally developed further in evolutionary steps, which we will describe next:

- 1. Monolithic application** Initially, the platform development started with the *Ruby on Rails* [48] web development framework. All presentation and business logic was concentrated in one monolithic application, which was executed with one process type and many process instances. One database system held all data.
- 2. Splitting the monolithic application** Starting 2010, functionality started to be moved out of the monolithic application and into independent applications [49] [50]. These applications might still have major functional dependencies on the business logic of the monolithic application. New features are not developed in the monolithic application anymore, but as new applications. We see the case study being currently in this stage, since the monolith still carries major significance.
- 3. Full microservice architecture** The goal of the architecture evolution is to reduce the significance of the monolith to “just being another service”. Thus, the functionalities of the platform as a whole should be split relatively equally over all applications. A platform that is developed further towards this style is Twitter, introduced in [2].

Figure 3.1 visualizes these three evolution steps.

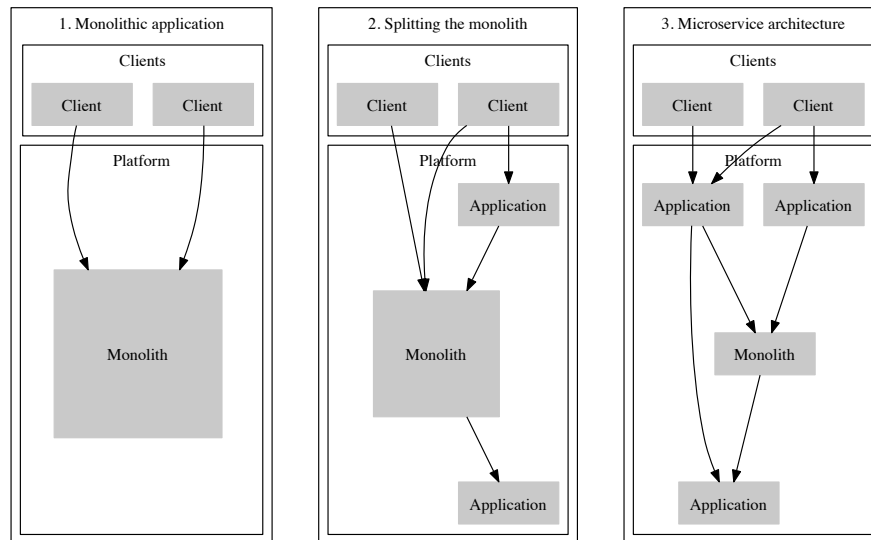


Figure 3.1: Visualization of the architecture style evolution from monolithic architecture style to microservice architecture style. Boxes represent applications and directed edges represent service dependencies. Size of the “monolith” represents its significance in the architecture.

We see two major drivers for this evolution:

"The right tool for the job" Different functionalities require different approaches to implement them efficiently. These approaches may be hard to consolidate in one application, most notably since different programming languages or frameworks might be used.

De-coupling of development and deployment Small teams of 2 - 10 people are believed to work more effectively than bigger teams [51]. Conway’s law [52] suggests that the structure of a system designed by an organization will follow the organization’s communication structure. Following this law, it is logical to assume that as an engineering organization grows with many small teams, the architecture will follow the same style. As a side benefit, small teams and applications allow for easier and therefore more often application deployment,

which in turn increases the velocity and quality of software [22] [26].

During our investigations we noted that the development of the architecture was not centrally controlled but rather happened to the discretion of individual engineers. Every engineer was able to decide how functionalities should be split into different applications. This led to a lack of oversight over the architecture development in the engineering organization. During our investigations we also found many different technologies in use for developing applications. For example we found 9 different programming languages used in the case study (as further explained in subsection 4.2.2 *From service interface modules*, especially Table 4.1).

3.1.1 Technical details

In this section we cover some technical details of the case study.

All codebases were organized with the Git [33] source code versioning system. The canonical location for all repositories is at GitHub [53], which is a SaaS providing Git repository hosting. In our case study, all repositories were organized within one “GitHub organization” [54], which is a logical grouping of hosted repositories, providing centralized access control and billing. We provide more statistics over repositories and their usage in later subsection 4.2.1 *Manually creating a dependency graph*.

All repositories on Github have an identifier. Since each application has a canonical Github repository and since these are unique, we may use the repository identifiers as application identifiers as well. We found two cases where a repository contained several applications. In these cases we assigned that repository to several applications with different identifiers¹.

In our case study, machines were distributed over several company-operated datacenters as well as rented cloud platforms. Several different Linux distributions were in use.

3.2 Infrastructure systems

Next we will describe the systems for *deployment*, *service discovery* and *telemetry* that we found in our case study.

¹One of the cases is described later in section 5.2

3.2.1 Deployment systems

In our case study we found various systems facilitating deployments. We introduce *Bazooka*, which is a “Platform as a Service” (short: PaaS) for deploying stateless applications, and *Chef*, which is a machine configuration system that is used for deploying stateful applications.

Bazooka

Bazooka [55] is a “Platform as a Service” deployment system. It is similar to the hosted PaaS Heroku [56] and Google App Engine[57], as well as the self-hosted PaaS Mesos [58], Flynn [59] and Deis [60]. Among others Bazooka handles the building of deployment artifacts, resource allocation for job scheduling, process supervision, service discovery setup, request handling and load balancing, as well as monitoring and logging.

Bazooka facilitates the deployment of applications on internal resources. It was built for applications that expose services via HTTP. The applications need to adhere to the constraints of “12 factor” [32]. “12 factor” is based on observations made by Wiggins regarding the design of applications in the context of the PaaS Heroku [56]. The developers of Bazooka adopted these constraint for their platform as well. The constraints relevant in the context of our work are as following:

- All applications’ codebases are tracked in Git [33].
- All processes are “stateless”, meaning that they hold no persistent data. This alleviates the need for a data consistency model between processes. Also, data locality and durability are no concerns. This constraint allows for near-linear horizontal scaling, meaning that double the amount of service instances should allow for handling double the amount of requests per second.

Bazooka consists of the following components:

Bazooka repository manager handles the building of the applications.

Bazooka application host provides the computing resources to run the processes and manages the life cycle of the processes.

Bazooka proxy manages service discovery and request routing to processes.

If an engineer wants to deploy an application with Bazooka, they first have to go through an initial setup. They create a Bazooka application on the Bazooka repository manager. The identifier of the Bazooka application is the same as the repository

name and has to be unique. Next, a Bazooka configuration for the application is created. Each configuration is a set of key-value pairs, which at runtime are available to the application via operating system environment variables. The current state of the Bazooka system, including applications, configurations and running processes, is managed via Doozer [61], a distributed, consistent data store.

The process of deploying an application with Bazooka is as follows: The engineer pushes the codebase to a git remote² on the Bazooka repository manager. There a deployment artifact is built (usually described via a Makefile [62])³. Once the building procedure is completed the engineer may start processes from the deployment artifact. Each deployment artifact is referenced with a revision⁴. When a process instance is started, the deployment artifact for it is transferred to the appropriate Bazooka application hosts, the process instance is started in the operating system and the running service is included in the service discovery.

At the time of writing, Bazooka was the preferred way of deploying microservices in our case study. On March 14th 2014, the deployment of Bazooka in our case study had 1 Bazooka repository manager, 2 Bazooka load balancers and 45 Bazooka application hosts, with 211 application deploys running 1333 processes⁵.

Chef

Chef describes itself as “a systems and cloud infrastructure automation framework that makes it easy to deploy servers and applications to any physical, virtual, or cloud location [...]” [63] as well as “a configuration management tool designed to bring automation to your entire infrastructure.” [64]. A configuration is the state of an architecture at a point in time. The configuration that Chef uses is described in code. This might include setting up a file structure, installing and managing applications or collecting and rendering configuration files.

Chef has a central Chef server that manages the configuration code, and a Chef client process on each managed machine. The highest granularity in Chef to manage machines is “roles”. A machine might have many roles assigned. Each role has code associated

²A git remote is a git repository that is hosted somewhere else than the current repository. It is possible to push and pull changes from git remotes, in order to share file changes. For more information, see [33] Section 2.5.

³It is also possible to deploy already-built artifacts.

⁴This is either the git commit identifier hash string or a revision string assigned by the Bazooka repository manager for the deployment artifact.

⁵This includes staging and testing deploys as well as prototypes and infrastructure applications

which gets executed in order to configure the machine. The Chef server allows to query which roles are applied to which machines. Conversely it is possible to fetch the applied roles from each machine via its Chef client.

In our case study, Chef is used to deploy all infrastructure applications. For example the components of the *Bazooka* deployment system get deployed themselves via Chef. *Chef* is also used for deploying applications which are stateful and therefore can not be deployed via *Bazooka*.

3.2.2 Service Discovery

Service discovery is the procedure of finding the process instance locations for a specific deploy. In our case study we found four service discovery mechanisms: *physical addresses*, *semantic DNS CNAMEs*, *Bazooka* and *Glimpse*. When describing service discovery mechanisms, one is often interested in attributes like propagation delays, load balancing or fault tolerance [65]. In our investigations, we are only interested in how the mappings between service discovery key and locations work. We will make extensive use of mechanisms introduced here in later chapter 4 *Constructing dependency graphs*.

Physical Addresses

Under physical address we understand two methods:

1. the **IP address** directly like `10.20.103.02`; or
2. a **physical domain name** denoting the machine directly without application semantics like `app08.internal.example.com`.

In both cases, the port numbers of the services for each address have to be known as well, in order to make a connection to a service. Physical domain names are registered and resolved via the DNS Domain Name System [66].

IP address and physical domain name are usually assigned when a machine is initially integrated into the company environment. We expect these assignments to not change⁶.

⁶Physical machines may get re-assigned. In our terminology this would constitute the decommissioning of the old machine and integration of a new machine. Also *Virtual IP addresses* were in use in the case study to provide fault tolerance capability on the traffic layer as well as for load balancing reasons. These were exceptions and we did not include them in our investigations.

Both methods may be seen as not constituting real service discovery. Even though for the physical domain name there is one DNS resolution step, it has no semantic meaning and therefore in our investigations does not add service discovery value.

We found both methods to be commonly used in our case study. They also are the final resolution step for the other methods, since they allow the actual addressing of machines and services via the network.

Semantic DNS CNAME

Apart from the physical domain names, we also found semantic domain names in use, defined via DNS CNAME records following the pattern of `{name}.internal.example.com`. These are configured manually by engineers, and map to one IP address. As with physical domain names, these do not include port numbers, which have to be known already.

Even though the usage of this method is discouraged, it is still in use (as we will show in later subsection 4.2.3 *From deployment configuration*). We attribute its use to the simplicity of registering and querying the domains.

Bazooka

The preferred way of reaching services deployed with Bazooka is via the Bazooka load balancers. When an application is initially registered with Bazooka, each of its services gets assigned a port number. The services keep their port numbers indefinitely, therefore these are the service discovery contracts with the consumers. A consumer may use the service by requesting a Bazooka load balancer on the application's port number. The load balancer will then forward the request to one of the service instances.

Bazooka has several load balancer instances (at the time of writing it were 2 with plans to use more in the near future). All instances share the same configuration mapping between port numbers and the service instances available for each of these port numbers.

The Bazooka load balancers may be addressed via one of the other service discovery methods. The most common method we found during our investigations (as explained later in subsection 4.2.3 *From deployment configuration* and subsection 4.2.4 *From network connections*) was via semantic CNAMEs. Figure 3.2 shows an example of addressing a service instance with the request flow from the service consumer to a service instance.

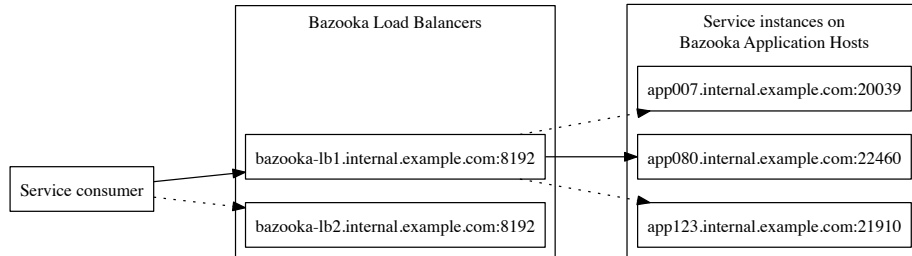


Figure 3.2: Visualization of the request flow from client to service through the Bazooka load balancer. Boxes represent machines with the domain name and port number as names. Solid edges represent the request flow. Dotted edges represent potential opportunities for different request flows, which would still yield the same results.

Glimpse

Glimpse is an application for service discovery, developed in-house of our case study. It is based on DNS SRV records (specified in RFC 6763 [67]), which for one domain name allow to request many pairs of domain name and port number. The query domains have the following format:

```
{protocol}.{job}.{env}.{product}.{zone}.internal.example.com
```

protocol denotes the application-layer protocol with which to access the service.

Examples are *http* or *ssh*.

job is the name of the service in the context of the application. Examples are *web* or *admin*.

env is the environment of the application deploy. Examples are *production* or *staging*.

product denotes a grouping of user stories. It might match an application and repository name, but there is no convention enforcing that.

zone denotes the zone, which a logical grouping of computing resources. Zones might for example be used to differentiate data centers.

An example domain name is

```
http.app.prod.soundcloud.ca.srv.internal.example.com
```

An example query is shown in Listing 3.1.

```
$ dig +short SRV http.app.prod.soundcloud.ca.srv.internal.example.com
0 0 5023 app117.internal.example.com
0 0 5023 app118.internal.example.com
0 0 5024 app097.internal.example.com
0 0 5024 app117.internal.example.com
```

Listing 3.1: Service discovery via Glimpse; Query example

The service discovery requests of Glimpse are handled via BIND [68] DNS server. At the time of writing, two existed, which are the authoritative sources for the Glimpse DNS namespace. The namespaces are initially configured by engineers, but the actual service instance information for each glimpse domain is gathered and distributed to the DNS server automatically.

3.2.3 Telemetry

Telemetry systems measure the state of a system, collect this information and make it available to engineers. During our investigations they were crucial tools for understanding and reasoning about the deployed architecture. Here we present three telemetry systems that were in use in our case study. More systems do exist, but we limit our descriptions to the ones that were relevant to our investigations.

Graphite [69] describes itself as doing two things: “1. Store numeric time-series data
2. Render graphs of this data on demand”. Every time series consists of many datapoints. A series has a resolution, defining how many seconds of time each datapoint covers. In our case study, the standard resolution was 60 seconds. Data is sent to *Graphite* via HTTP, setting the value for individual data points in time series. During querying, time series may be transformed and combined through functions. Graphite allows the extraction of the raw time series data as well as displaying visual graphs. In our case study, the majority of measures were collected and displayed with Graphite.

StatsD [70] describes itself as “[a] network daemon that [...] listens for statistics, like counters and timers, [...] and sends aggregates to [...] backend services (e.g. Graphite)”. *Graphite* only allows setting absolute numeric values for each datapoint. *StatsD* allows for aggregating data for each datapoint before, for example when an event (like an HTTP request) happens, its result might be sent

to *StatsD*, which aggregates all of these events for the current datapoint and then sends that aggregated datapoint to *Graphite*.

Prometheus [71] describes itself as “a generic time series collection and computation server”. Different to *Graphite*, its data collection mechanism is pull-based: in regular intervals the *Prometheus* server queries a telemetry HTTP resource on each measured instance. The instances internally aggregate the metrics.

In our case study, *Prometheus* is set to eventually replace the combination of *Graphite* and *StatsD*. At the time of our investigation, both systems were in use and served different measurements.

3.3 Summary

In this chapter we have introduced our case study, the berlin-based company “Sound-Cloud” in which the author was embedded while investigating this work. We have shown how the architectural style used in the case study has evolved from a monolithic multi-tier web application towards a microservice architecture (as visible in Figure 3.1). We have also described concrete implementations of infrastructure systems, on which we will rely in the following chapters. As a problem in assessing the dependability of the deployed microservice architecture we identified a lack of visibility regarding the dependencies between applications.

4 Constructing dependency graphs

In this chapter we investigate how to construct a graph of application dependencies from a deployed and running microservice architecture. We first introduce the assumptions regarding the properties of the graph. Then we propose four methods for graph construction. We end the chapter by discussing and summarizing our findings.

4.1 Theory

In literature dependency graphs often are understood as “program dependence graphs” [72] which model how a program works internally on the granularity of software modules, control flow or data flow. In this work, we use the granularity level of applications and dependencies between these as the basis for dependency graphs.

Directed graph

A **directed graph** (or “digraph”) consists of *vertices* and *directed edges* [73]. The edges connect the vertices. Every edge has a direction, leaving from one vertex and arriving at another vertex. All edges leaving from a vertex are called *outgoing* edges. All edges arriving at a vertex are called *incoming* edges.

Vertices represent applications in a microservice architecture.

Earlier in section 2.2 *Software environment terminology* we established that one application has many processes and each process might implement or consume many services. Thus if there are many processes (or services) for one application in the graph, these would all be represented by one vertex for the application they belong to.

Directed edges represent service dependencies between applications.

A service dependency is the fact that a process is the consumer of a service (as explained earlier in section 2.2.1). The direction of the directed edge follows the direction of the service dependency, from service consumer to service provider. If the vertices represent

applications, the service dependency may exist for these as well, since processes and service always belong to exactly one application

Example graph

Figure 4.1 *Dependency graph: example* shows a graphical example. It shows the following information:

- Application A is dependent on Application B
- Application A is dependent on Application C
- Application B is dependent on Application D

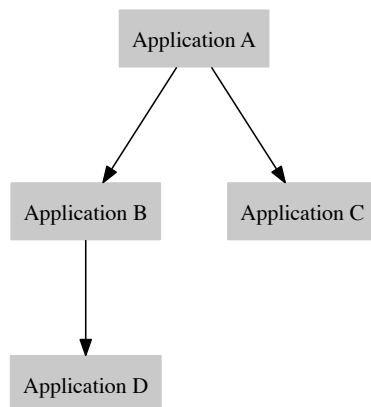


Figure 4.1: Dependency graph: example

4.2 Methods for construction

To construct a dependency graph from a deployed microservice architecture, we propose four methods in this chapter:

4.2.1 *Manually creating a dependency graph*

4.2.2 *From service interface modules*

4.2.3 *From deployment configuration*

4.2.4 *From network connections*

We will discuss each of these methods after the following section structure: We first introduce the theory of the method based on our terminology model for microservice architectures from section 2.2 *Software environment terminology*. The theory includes the assumptions the method makes against the environment it is executed in, and the high-level process of executing it. Afterwards we describe how we executed the method in the context of our case study and present the respective results. To conclude each section we discuss our findings. For that we evaluate each method after the following criteria:

Feasibility Is it possible to execute the method at all?

Correctness How well does this method detect all applications and dependencies?

Automation To what extent can this method be automatically executed?

4.2.1 Manually creating a dependency graph

This method is based on people manually creating the dependency graph.

The naive way of doing this is without a formalized process, but rather ad-hoc to the discretion of some people. This approach is limited by how much an individual person or a group of people may know about the whole architecture. With an increasing size of an architecture (quantified by number of applications and service dependencies) the generated graph will be more prone to include errors, since it will be less likely that the involved people have enough holistic knowledge about the architecture. In our case study we found several manually created diagrams representing application dependencies. They lacked a well-defined definition of the elements of the diagram and represented the architecture from a specific viewpoint only, but never showed a holistic view.

In this section we propose a semi-automated approach for generating a dependency graph for a deployed microservice architecture. It is based on people manually

annotating the service dependencies of individual applications in a structured format. These annotations are then gathered in order to generate the dependency graph.

Theory

We base this approach on the following assumptions:

1. Every application has exactly one codebase
2. Every codebase is tracked in a source code version control system
3. Every codebase has one canonical repository location
4. Every canonical repository location has a unique identifier within the scope of the microservice architecture
5. It is possible to fetch a list of all repositories within the scope

Given these assumptions, we may do two steps: First we manually annotate each repository with its respective service dependencies. Then we gather all repositories and annotations to create the dependency graph.

We start by fetching a list of repositories. For each repository, we decide, if it constitutes an application that fits the criteria for being in the dependency graph. Example of applications that might be excluded are prototypes or historical codebases. If an application does not fit the criteria, it is not annotated. If it fits the criteria, an annotation for it is created. Each annotation holds a list of applications that the current application has service dependencies to. The application identifiers used are the repository identifiers.

Given that all annotations are created, we may create the directed graph from them. Each annotated application becomes a vertex in the graph. From the annotations of that application, each service dependency becomes an outgoing directed edge to the respective application providing that service.

Next, we will describe how we executed this method in our case study.

In the case study

All application repositories of our case study were kept under one GitHub organization. This provided a central unique naming scope for the repository identifiers. It also made it possible to fetch a list of all repositories. In our case study, we fetched 545 repositories.

We then decided for each repository, if it should be part of our investigation. We used the following criteria:

1. The application is currently deployed in the production environment
2. The application is client-facing, thus it implements a functionality visible to users

This excluded many repositories, for example:

- applications, which do not consume a service (like data storage system, which only provide a service)
- infrastructure applications, which do not directly satisfy user requests (like deployment and telemetry applications)
- deprecated applications, which are not deployed anymore
- prototype applications, which are not deployed in the production architecture
- shared libraries, which are not stand-alone applications themselves
- documentation, which are no applications at all
- one-off jobs, which do not directly satisfy user requests (like analytic aggregations)

We identified 75 applications fitting our criteria. For each of these, we manually created an annotation file in JSON¹ format. We defined our own annotation format, since existing dependency annotation systems like the dependency management tools *Bundler* [75] or the dependency management system of *Go* [76] (which we both describe in detail in the following subsection 4.2.2 *From service interface modules*) were created with a focus on shared code library dependencies. Still we believe it would be possible to map our annotation format to other formats, given that these work with a “manifest file” like *Bundler* does.

An example of an annotation file in our format can be seen in Listing 4.1. It annotates the *threaded-comments* application, which we will also work with in chapter 5 *Constructing qualitative fault trees*. As application identifiers we used URIs (as specified in [77]). In the example, we see two kinds of application identifiers: The first one are URIs of web URIs of Github repositories. The second kind is a URI to the software project’s homepage.

¹“JSON is a text format that facilitates structured data interchange between all programming languages” [74]

```
1 {  
2   "dependencies": [  
3     "http://github.com/soundcloud/soundcloud",  
4     "http://github.com/soundcloud/authenticator",  
5     "http://www.memcached.org"  
6   ]  
7 }
```

Listing 4.1: Dependency graph: the manual annotation file in JSON format for the threaded-comments application

Given all annotation files, we created the dependency graph out of that. Figure 4.2 shows the graph for our case study. We created the graphical representation with Graphviz [78], which also orders the vertices hierarchically based on the edges. The hierarchical ordering has no semantic meaning applied by us. Furthermore we masked the application identifiers with numbers, in order to protect confidentiality of the case study company.

The following things can be seen in the visualized graph: In total, we identified 90 applications and 188 service dependencies. Towards the top of the graph are applications with no incoming edges. These applications are directly used by the clients. One can argue that the clients have service dependencies on these applications. We excluded clients from this investigation, thus these are not present. Several application show a high concentration of incoming edges: Vertex 2 is the database application, vertex 3 is the message queueing application and vertex 5 is the former monolithic application.

Discussion

In our case study we executed the application identification and dependency annotations by looking at the information in the repositories. This includes the source code, documentation, build information, required environment variables and packet management information. We also asked engineers and used information from the deployment system. Still, we assume that our method and our execution of it has a non-determined percentage of mistakes. We assume one way to reduce the percentage of mistakes is by deferring the responsibility to annotate the service dependencies to the teams that are responsible for the individual applications, since these are the individuals who probably know their applications best. We propose that the actual annotation

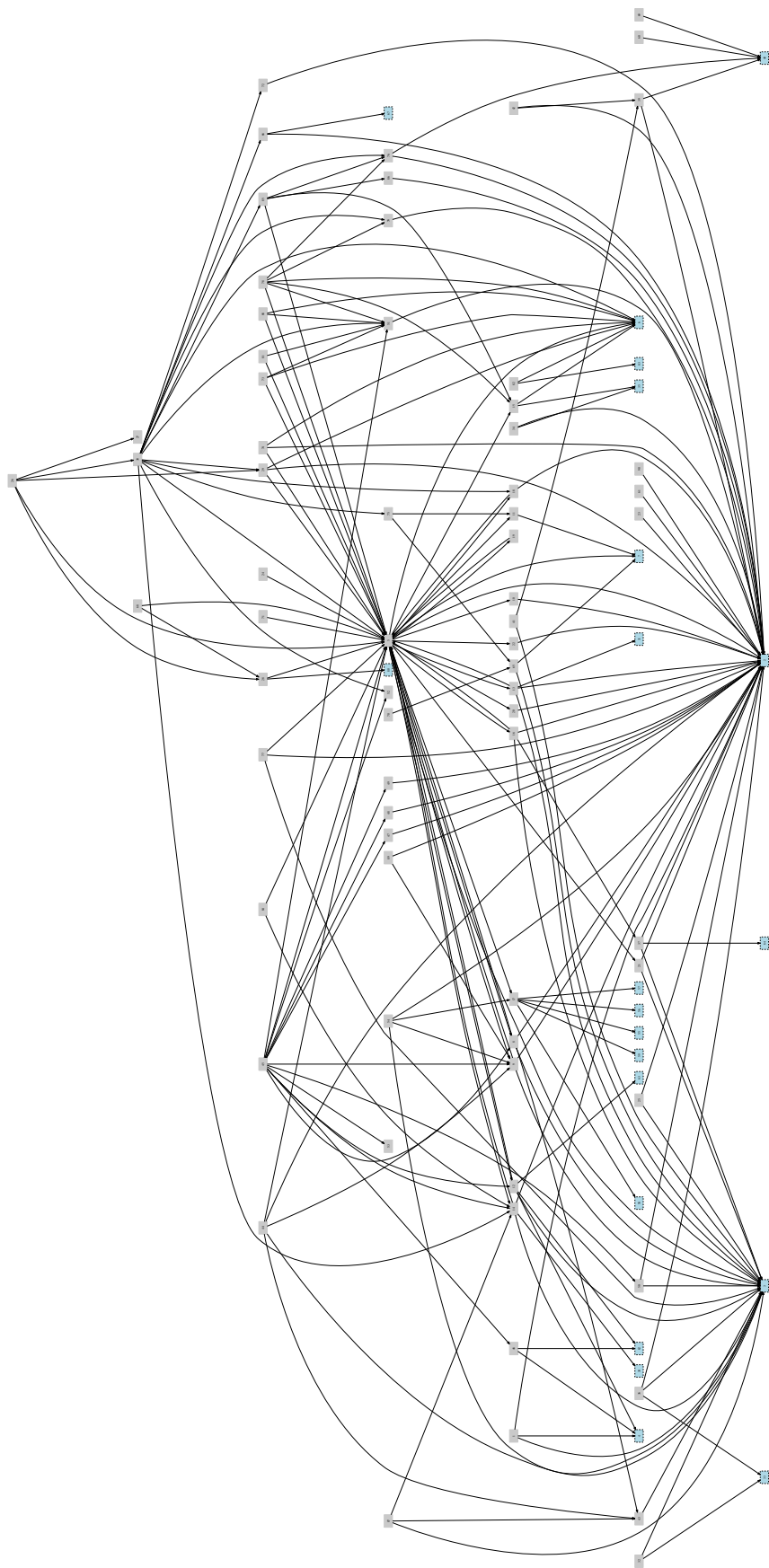


Figure 4.2: Dependency graph: Manual annotation from case study with masked application names. Grey boxes are internal applications. Blue boxes with dotted outline are external applications.

file should then be kept within the application repository, e.g. as a file like “service-dependencies.json”.

A problem with manual annotation is to keep the information up-to-date, e.g. when applications are retired or service dependencies change. This gap impedes on the correctness of the dependency graph and therefore raises the question of how much the generated graph may be trusted by engineers.

Given that several people annotate the dependencies, using unambiguous application identifiers becomes a problem. Getting unique identifiers is solved by using URIs that are reachable via the global DNS system. The more difficult requirement is if two people individually encounter an application, they need to assign the same identifier to it. This is easy for applications within the organization’s scope, given that we assume unique repository identifiers, which can be used directly as application identifiers. But for external applications, we did not find an unambiguous convention, given that not all applications have a canonical repository location and not all applications have a canonical project website.

To conclude the discussion, we evaluate the method after the three criteria:

Feasibility Given our experiences from executing the method in the context of the case study, we believe this method to be well feasible.

Correctness The **correctness** of the generated graph directly depends on who creates it. If few people create it, the organizational overhead is low but the people involved may not have enough knowledge to create a holistic view of the architecture. If many people are involved in the creation, enough knowledge over the architecture may be contributed but the organizational overhead may be a significant burden. We believe that distributing the creation of the annotations to the teams building and maintaining the applications might be a way to provide an acceptable level of correctness, which should be investigated more in future work. The correctness of the method should also be seen over time: The dependency graph not only needs to be created once, but also maintained over time, given that architectures are very likely to change. We believe that the correctness of the generated graph eventually is a cultural problem of the engineering organization.

Automation By the nature of the method, manual annotation is not **automated**. But given that distributed creation of annotation information might exist, the gathering of the annotations and subsequent generation of the dependency graph may happen automatically.

Future work

We see manual annotations as a cultural problem. Future work might explore, how the creation and maintenance of manual annotations in an organization may be incentivized. This is especially a problem, since the benefits of annotations might not be immediately visible to their creators.

As another opportunity we see the possibility to combine manual annotations with the other methods introduced in this chapter. Other methods could provide indicators for applications and dependencies, but the eventual decision over an annotation would still be with a person.

4.2.2 From service interface modules

In this section we propose another approach for generating a dependency graph. It is based on the assumption that all service dependencies of an application are encapsulated in service interface modules and thus extractable from the source code. Therefore this method would allow us to extract service dependencies exclusively from the source code by static code analysis.

Theory

To begin with, we assume that there is a list of applications and their respective codebases. An example for generating this list is through manual creation (as we also did in previous subsection 4.2.1 *Manually creating a dependency graph*). Given that we have these applications, we may analyze their codebases.

The central assumption of this method is that access to service dependencies is encapsulated in “service interface modules” (short: SIM). A SIM is a software module within an application. It exists to encapsulate the access to specific service.

We see two kind of SIMs:

1. shared libraries, which usually encapsulate access to external services
2. internal modules, which usually encapsulate access to internal services

Furthermore we make the following assumptions about SIMs:

- From each applications’s codebase it is possible to extract the used SIMs.
- It is possible to map a SIM to the application identifier it encapsulates.

Table 4.1: Approximate percentual statistics over languages of relevant applications in our case study.

Ruby	Go	Scala	Clojure	JavaScript	Python	Java	C++	C
37%	16%	15%	8%	15%	2%	2%	4%	1%

Given these assumptions, for each application the SIMs may be extracted and matched to the accompanying applications. This results in a list of service dependencies per application. These may then be used to construct a dependency graph, by taking the applications as vertices and the service dependencies of each application as outgoing edges to the respective service-providing applications.

In the case study

To execute the approach in our case study, we start with a list of relevant application identifiers. We manually created this list after the same criteria as in previous subsection 4.2.1, which means we continue to work only with applications that are deployed in the production environment and serve a user-facing purpose. This list of application identifiers allows us to access the codebase for each application, since it coincides with the identifier for the repositories. Therefore, we may acquire the codebases for all applications and continue with the next step.

Given that the codebase is accessible, the process of extracting SIMs from it may be executed. In our case study, applications are written in a variety of programming languages [79]. Table 4.1 shows a percentual breakdown of the application languages. We derived this information from the Github repository language determined via Linguist [80]. From the table it is visible that *Ruby* and *Go* are the most common programming languages, thus we started to investigate them first.

Ruby

In Ruby “shared code libraries” are packaged and distributed as “Gems” [81]. In our case study we found all Ruby applications to use “Bundler” [75] to manage their Gems. Bundler is a dependency manager that facilitates loading the correct Gems as well as keeping these configurations portable when changing the execution environment. It does so by creating a manifest file (called “Gemfile”) in the application’s root directory. The Gemfile holds information about the needed Gems. Listing

4.2 shows a shortened Gemfile from an application of our case study. We did not find an automated way to determine the application identifiers for the encapsulated service dependencies from Gems. We believe this information to also be hard to generically exist, since the application identifiers might be dependent on the context of the individual organizational context. In our case study, we manually mapped Gem names to application identifiers by looking at the name of the Gems, the documentation for the Gems as well as our experience and asking engineers of our case study. Looking back at the example in Listing 4.2, the 2 gems “haml” [82] and “rack” [83] do not encapsulate a service dependency, thus are no SIMs. The gem “jberkel-mysql-ruby” [84] encapsulate access to the *MySQL* [85] application. The gem “amqp” [86] encapsulates access to the *RabbitMQ* [87] application.

```
1  [...]
2  source 'https://rubygems.org'
3
4  [...]
5  gem 'amqp',
6  [...]
7  gem 'haml',
8  gem 'jberkel-mysql-ruby',
9  [...]
10 gem 'rack'
11 [...]
```

Listing 4.2: Shortened Gemfile from an application of the case study

For “internal modules”, Ruby offers the constructs of classes, objects and modules to encapsulate software components. Due to its dynamic nature and facilities for sophisticated meta programming, we found it unreliable to extract these as SIMs from the source code. Furthermore, even if these could be reliably extracted, the mapping between SIM and encapsulated application would not be present.

Go In Go the preferred way of separating software components is via packages. It is possible to extract a list of all imported packages from a Go application². Packages are identified during import via their path within the \$GOPATH directory. With this

²In Go version 1.2, the following code returns a list of imported packages:

```
go list -f 'join .Imports "\n"'
```

method, there is no distinction between “shared code libraries” and “internal modules”, thus they may be treated the same.

As an example Listing 4.3 shows the shortened output of the list of imported packages for an application with identifier *threaded-comments*. The packages `flag` and `fmt` belong to the standard library of Go. The package `github.com/streadway/handy/breaker` is a shared library that does not encapsulate access to a service dependency. The package `github.com/ianoshen/gomemcache/memcache` is a shared library that encapsulates access to the memcached [88] application. We were able to determine these mappings by manually looking at the packages’ documentation. We did not find opportunities to automatically determine the mapping between package and application identifier. We were however able to identify, if a package opens a network connection³. This might allow identifying candidates for SIMs, which then in turn could be mapped to application identifiers manually by a person.

```
1  [...]
2  flag
3  fmt
4  github.com/ianoshen/gomemcache/memcache
5  github.com/streadway/handy/breaker
6  [...]
```

Listing 4.3: Shortened list of imported packages of the thread-comments application

After the experience with these two languages, we did not investigate the extraction of SIMs for other languages. We expect them to work in similar ways with similar conclusions.

It is to note that in our case study, we did not find occurrences of access to internal services being encapsulated in shared libraries. Some service dependencies were encapsulated in internal modules though, but we did not find a way to map the respective application identifiers to them.

Given these experiences, we did not generate a dependency graph via this method.

³We did identify potential network connections by analyzing the abstract syntax tree of the codebase for calls to the “net” package’s methods for opening network connections

Discussion

Looking at the results from earlier subsection 4.2.1 *Manually creating a dependency graph*, we see that the majority of service dependencies are between internal applications. In our case study execution we found that dependencies between internal applications are nearly never encapsulated in identifiable SIMs. Therefore, a dependency graph generated with this method would not include a large amount of existing service dependencies, impeding heavily on the completeness of this method.

When we identified SIMs in the case study execution, we were not able to match these to their respective application identifiers. The only option was to manually map them. Therefore, it does not seem possible to execute this method automatically.

One aspect that impedes on the feasibility of this method is the needed implementation effort. This method requires extensive knowledge about the used languages and their dependency management systems. Thus, there needs to be an individual implementation for each configuration, which in a polyglot environment constitutes significant effort.

To conclude the discussion, we believe that this method is not well **feasible**. Given the problems with identifying internal applications, we believe the resulting dependency graphs to significantly lack **completeness**. This method may be executed **automatically**, given the problem of mapping SIMs to application identifiers is solved in an automated way.

Future work might investigate the creation of a database of SIM-to-application-identifier mappings. When a person manually annotates a “shared code libraries” or “internal modules”, this mapping could be then used in subsequent executions of the method, which would reduce the total amount of manual mapping work needed in many executions of the method.

4.2.3 From deployment configuration

With the method we introduce in this section, a dependency graph may be generated based on the service dependencies extracted from application deploys. To do so, the values of the deploy’s configuration are examined and mapped to application identifiers via reverse service discovery.

Theory

We base this method on the following assumptions:

Applications are configurable Applications are built in a way, such that external configuration, which tailors the application to the current execution environment, might be injected when the application is started. Examples of configuration are enabling or disabling features (e.g. logging, debugging, choosing algorithms) and setting thresholds (e.g. timeouts, queue sizes).

Applications are configured during deployment We assume that all applications are deployed with the help of a deployment system (like outlined for our case study in earlier subsection 3.2.1 *Deployment systems*). The deployment system compiles the configuration for the deploy and makes it available to the processes. The format of the configuration is not relevant here; examples are operating system environment variables or configuration files. The exact configuration details may be gathered from different sources, like a database or the runtime information from the currently deployed architecture.

Service discovery is part of the configuration Service discovery is the process of finding the service instances of a certain application deploy (as explained earlier in subsection 3.2.2 *Service Discovery*). We assume that service discovery happens to a certain extent during the deployment process and is present in the configuration, thus allowing the execution of *reverse service discovery* on the configuration values. We use the term reverse service discovery to describe the process of mapping a service discovery key, service location, intermediate value of the service discovery process to its respective application identifier.

Given these assumptions, we may execute the method as follows:

1. Fetch a list of relevant application deploys.
2. For each deploy, we need to have access to the currently deployed configuration.
3. For each configuration, we go through all values and identify the ones that are service locations. For each service location we then identify the respective application identifier via a reverse service discovery lookup.

Resulting from this method is a list of deploys and therefore also applications⁴. For each deploy, a list of service-providing applications exists. We may then generate a dependency graph out of that information: Every deploy is a vertex. Every service dependency becomes an outgoing edge from the deploy to the service-providing application.

In the case study

Next we describe how we executed the method in the context of the deployment system Bazooka. For the other deployment systems (explained in subsection 3.2.1 *Deployment systems*) we were not able to reliably identify deploys and their respective environments, thus we did not execute the method with them.

In Bazooka we may fetch a list of all application deploys from Bazooka's distributed data store. In our case study this list also included staging and prototype deploys, which we manually removed from the list since we were only interested in deploys in the production environment. Given this list we fetched the current configuration for each deploy⁵. In Bazooka the configuration is a list of key-value pairs. We extracted all values and used the methods explained next to map these to application identifiers. Listing 4.4 shows an example of such a configuration.

Through reverse service discovery, the values from the configuration may be mapped to application identifiers. Next we will introduce several methods for doing so. These relate strongly to the service discovery mechanisms we described earlier in subsection 3.2.2 *Service Discovery*.

⁴We assume that for every deploy identifier, the corresponding application identifier is derivable. A simplest case is, if the deploy identifier is the same as the application identifier. Another case is, if the application identifier is part of the deploy identifier and can be parse out of that (e.g. deploy identifier: "app-env" => application identifier: "app").

⁵In Bazooka, an application might have processes deployed with different environments. A typical case is a canary process [89]. When we encountered such a case, we chose the most recent environment.

```
1 {
2   "logLevel": "error",
3   "threadedCommentsHost": "bazooka-lb.internal.example.com:8485",
4   "soundcloudAppURI": "http.app.prod.soundcloud.ca.srv.internal.example.com",
5   "databaseTransactionalAddress": "10.20.30.10:1337",
6   "databaseAnalyticalAddress": "db04.internal.example.com:1233"
7 }
```

Listing 4.4: Example configuration of an application in Bazooka

Bazooka To consume services deployed with Bazooka, consumers may use the Bazooka load balancer. For a service consumer to use a service, it has to know the port on the Bazooka load balancer. Therefore, in the configuration of a deploy, the Bazooka load balancer address and port number may be used (as visible in 4.4 for the “threadedCommentsHost”). For our purpose of generating a dependency graph, we may map this port number to the application identifier, as described next:

Bazooka is using HAProxy as load balancing software. It is configured via a configuration file [90] generated by Bazooka. We may use that configuration file for doing reverse service discovery, from the Bazooka port number to the application name. Listing 4.5 shows a part of the Bazooka HAProxy configuration file for the threaded-comments application. The relevant part is the first line, which shows the port number (8485) on the bazooka load balancer as well as the application-process-type combination (*threaded-comments-web*) which, if split on the last hyphen, reveals the application’s identifier. Since the port number is used by the clients, we assume it to not change during the further development of Bazooka. On the other hand, splitting the application-process-type combination is only possible as long this string stays the same. Since it is an internal implementation detail of Bazooka, this might change in future versions of Bazooka.

```
1 listen threaded-comments-web bazooka-lb.internal.example.com:8485
2   bind-process 1
3   server threaded-comments-web0 app08.internal.example.com:1000 [...]
4   server threaded-comments-web1 app09.internal.example.com:3001 [...]
5   [...]
```

Listing 4.5: Bazooka HAProxy configuration file excerpt for the threaded-comments application

Glimpse With Glimpse, a service is identified via a URI following a well-defined format. In previous listing 4.4, the key “soundcloudAppURI” has such a URI as value. Given that the format is well-defined, it is possible to extract the *product* information from it⁶. The *product* information may then be used as application identifier.

Semantic DNS CNAMEs A service might be identified via a URI with a DNS CNAME on an internal DNS namespace. In our case study, these followed a format similar to *name.internal.example.com*. The CNAMEs were assigned manually by engineers. Given that these URIs follow a well-defined format, it was possible to identify URIs following it and extract the names from there. There is no convention around the usage of these names, therefore the name might or might not be an application identifier.

Physical Addresses In previous listing 4.4, the keys “databaseAnalyticalAddress” and “databaseTransactionalAddress” use physical addresses for denoting service locations. Our only way to map these to application identifiers is by manual intervention. This might involve semantically interpreting the URIs or key name, or connecting to the denoted machine to investigate running processes.

With these methods for reverse service discovery to our disposal, we investigated them in the context of our case study. We executed this investigation on May 30th. We fetched all current deploys from Bazooka (296 deploy identifiers) that had running processes (reduced to 136 deploy identifiers). We manually removed deploys that were not part of the production architecture, like staging and prototype deploys as well as internal productivity applications. This reduced the number of relevant deploy identifiers to 90. For each of the remaining deploys, we fetched the latest configuration.

For each deploy configuration value we executed the just-introduced automatic reverse service discovery methods. For method *Bazooka*, we had a list of addresses for the Bazooka load balancers. For method *Glimpse*, we used the *product* value as application identifiers. For method *Semantic DNS CNAMEs* we used the extracted names as application identifiers.

Subsequently we were able to generate the dependency graph: Each deploy’s application identifier became a vertex in the graph. Each of the discovered dependencies became an outgoing vertex from the deploy’s application identifier to the identified service-providing application. The resulting graph can be seen in Figure 4.3.

To cross-validate these results, we manually determined the application identifiers for each value, taking the value and key name into account. The resulting graph can be seen in Figure 4.4.

⁶Glimpse URIs are specifically meant to be parseable by regular expressions.

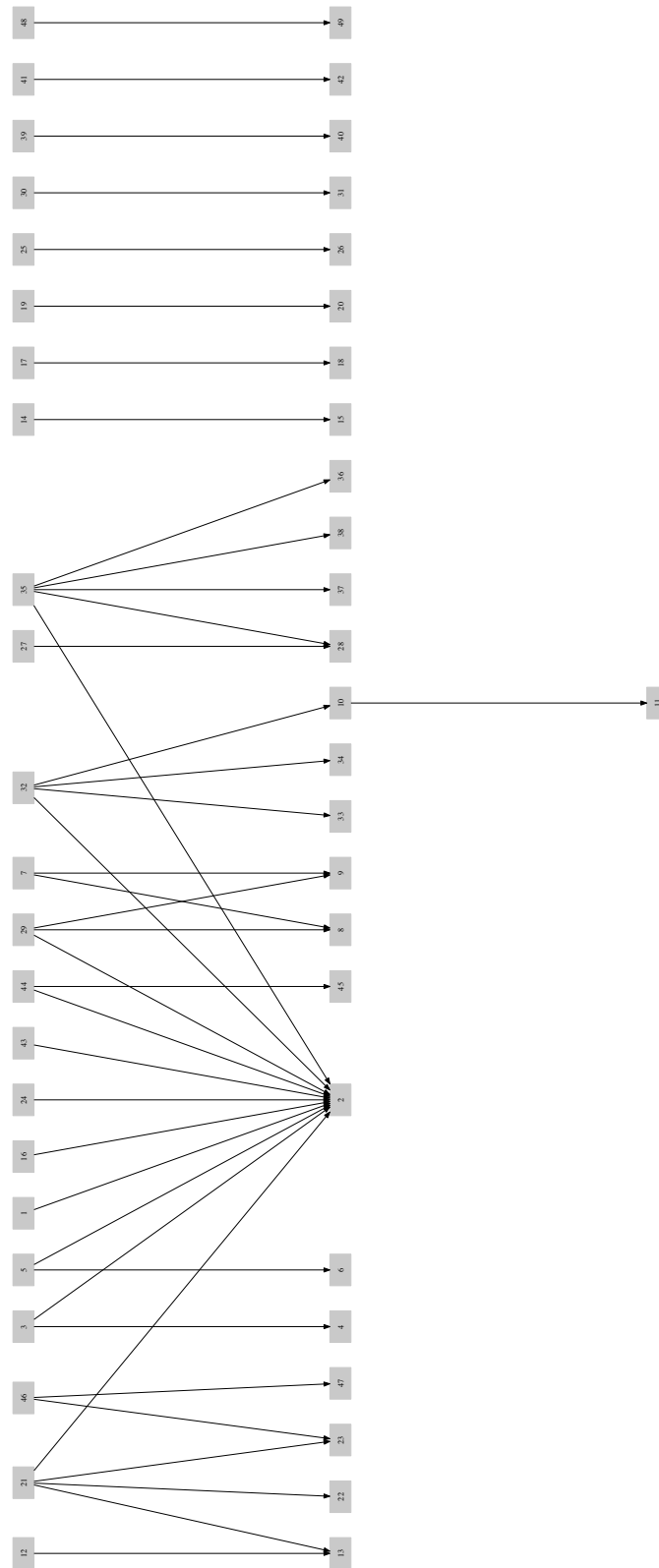


Figure 4.3: Dependency graph generated from deployment configuration and executed with **automatic reverse service discovery** in the case study. The application identifiers are masked with incremented numbers.

Discussion

We start the discussion by comparing the manually and automatically generated graphs with each other. The major difference between both are the sizes of the graphs. The automatically generated graph 4.3 has 49 vertices and 41 edges, whereas the manual graph 4.4 is about double as big with 88 vertices and 99 edges⁷. We attribute this difference to the fact that in the manual execution we have all reverse service discovery methods of the automated version available, plus are able to deal with edge cases manually. These included the following:

Decomposed URIs We found 5 URIs that were decomposed into several configuration variables. One example for that is splitting the hostname and the portname into two variables, as visible in Listing 4.6.

Incomplete URIs 7 times we encountered URIs that did not fully denote a service. One example is the bazooka load balancer address as configuration value without the port number, as visible in Listing 4.6, when only taking the “bazookaLB” variable into consideration.

Physical addressing not resolvable We encountered several physical addresses, which were not automatically resolvable. We were able to resolve these manually with the help of the variable name as well as by logging into the respective machine and investigating the applied chef roles.

Both graphs show the monolith of the architecture with a big number of incoming edges (vertex “2” with 11 incoming edges in the automated and vertex “5” with 13 incoming edges in the manual graph). We have seen the monolith to be mostly addressed via CNAMEs with the application name. The database *MySQL* (vertex “2”) makes up for a significant share of the edges in the manually-created graph Figure 4.4. These are not represented in the automatically-addressed graph, since they addressed specific deploys of the application by IP address or physical machine name.

```
1 {
2   "bazookaLB": "bazooka-lb.internal.example.com",
3   "threadedCommentsBazookaPort": 8485
4 }
```

⁷Please note that between the different graphs, the numbering of the applications is not consistent. Thus, application “1” in one graph is not the same as application “1” in another graph. This is due to the mechanism of masking we used, which did not sync naming between the generation of the graphs.

Table 4.2: Distribution of configuration cases for the dependency graph generated with manual reverse service discovery in Figure 4.4.

Metric name	Value
Total number of configurations	90
Configurations without any variables	17
Configurations with at least one variable	73
Configurations without any reverse service discoverable variables	39
Configurations with at least one reverse service discoverable variable	51
Total number of dependencies discovered	99
Dependencies reverse service discoverable via Bazooka	7
Dependencies reverse service discoverable via Glimpse	1
Dependencies reverse service discoverable via CNAME	65
Edge cases only service discoverable manually	26
CNAME discovered services directly mapping to an application name	39
CNAME discovered services being deploys of the MySQL application	22
CNAME discovered services being deploys of the RabbitMQ application	4

Listing 4.6: Example configuration of an application in Bazooka

Table 4.2 shows more information about the distribution of the configuration. It can be seen that CNAME reverse service discovery makes up the majority of the discovered services. One of the assumption around CNAME discovery was that the *name* is an application identifier. We found this assumption fulfilled in 60% of cases, with the other cases denoting specific deploys of the *MySQL* or *RabbitMQ* applications (e.g. `broker001.m.internal.example.com`).

In our investigations, we only found 1 case using the Glimpse service discovery. In that case, the “product” part of the URI did map the application identifier. Still, our method would benefit from either the convention that the “product” has to map the application identifier, or from another URI part that would include the application identifier. Among all methods of reverse service discovery, Glimpse proved to be the easiest to use. This is due to its well-defined URI format, which can be easily identifier via a regular expression, as well as the possibility to extract the identifiers without querying another system.

When comparing the graphs of this method with the earlier graph subsection 4.2.1 *Manually creating a dependency graph*, we can see that the earlier graph has more vertices and edges (90 vertices/188 edges) than our graphs here (manual 88 vertices/99 edges and automated 49 vertices/41 edges). The earlier graph also has 7 hierarchy levels, whereas the graphs of this method only have 4 hierarchy levels. We attribute this to the fact that the graphs generated with the method here may only take applications into consideration which have been deployed with Bazooka, whereas the earlier graph may also take differently-deployed applications into consideration. It is to be noted that the comparability of the graphs is limited by the fact that there were several months between the generation of these graphs and more applications were created within this time period which are not present in earlier graph subsection 4.2.1 *Manually creating a dependency graph*.

One limitation during our investigation was the reverse service discovery for applications outside of the organizations scope like third-party services. Two practical examples we encountered in our investigation were the REST API of Twitter [91] as well as the Amazon S3 web services APIs [92]. It would theoretically be possible to put these behind the internal service discovery mechanisms, but doing so is impractical, since it increases complexity without providing tangible direct benefits especially since the location of these external services is highly unlikely to change. Therefore we believe, these have to manually be mapped to application identifiers then.

To conclude, we have seen that the approach introduced in this chapter is **feasible**, since we were able to execute it in our case study. Its **correctness** and the degree of **automation** are influenced by how many of the requirements are met: Are all applications configured during deployment? Do all applications get their service dependency locations from that environment? Is it possible to obtain the configuration for the deploys? Is it possible to execute reverse service discovery for discovered service discovery locations? To structurally fulfill these requirements, we see deployment systems like Bazooka and service discovery systems like Glimpse as enabling factors.

Future Work

During the automated execution, we did not take the variable names into consideration. We did so during manual execution though. We found that variable names did not follow a convention. Still, they were useful in manually determining the service dependency application, even though this eventually meant “guessing” the actual application identifier. This could be improved by introducing a convention on environment variable names, which includes the application identifier. An example

format is “service_application_identifier”.

If the service locations are within the application’s codebases and not in the environment, it might still be possible to execute this method. This assumes that we can extract the service locations from the codebase. We may then execute the same reverse service discovery mechanisms as explained before.

One of the edge cases we identified was physical addresses (e.g. IP addresses like *10.23.0.10* or physical domain names like *app08.internal.example.com*). These do not hold service discovery information. If it is possible to connect to that address (e.g. via *ssh*), investigations on the server could happen. Examples are investigating running processes (assuming a way to map processes to application identifiers) or applied chef roles (assuming a mapping from chef role to application). Both examples assume that a machine only has one application running.

4.2.4 From network connections

With this method we generate a dependency graph from the service dependencies of applications. We derive these service dependencies by observing network traffic and matching the source and destination processes to applications.

Theory

We assume that all application dependencies manifest themselves as network communication between the applications’ processes. Thus if we detect network communication between two processes, we can derive that there is an application dependency between these applications.

To execute this method, we have to observe network communication. Assuming that communication is based on TCP/IP [37], we may take a TCP packet as atomic entity. A packet holds the information about the IP address and port number of the source and destination. If these can be mapped to their respective applications, we have determined the application dependency. If we are able to determine, which of the two communication parties started the communication, we can derive the direction of the application dependency, from source to destination.

Given all this information, the longer we observe the system, the more service dependencies may be collected. Each source or destination application becomes a vertex in the dependency graph and each observed service dependency becomes an edge.

In the case study

In our case study we executed this method in the context of applications deployed with Bazooka. We observed packets on the Bazooka application hosts. Each application host used a Linux distribution as operating system. We execute the packet observation using *netstat*, which allows to display currently active network connections.

```
1 192.168.100.1:39998 192.168.100.3:1001 10411/application
2 192.168.100.1:2001 192.168.100.2:35019 20471/python
```

Listing 4.7: Sanitized example output from *netstat*. Schema is: *local_address foreign_address process_id/process_name*

Listing 4.7 shows an example output from *netstat*, which we filtered to include only the essential information. The IP address *192.168.100.1* belongs to the machine where the capturing happened. The first line shows a connection that started from a local process to a remote service (*outgoing connection*). The second line shows a connection that started from a remote process to a local service (*incoming connection*). Next we will investigate for both cases, how to map the source and destination applications of the connection.

Source application When a process opens a new network connection, it usually does so by using network sockets [93] [94]. A socket needs a local port, which it usually gets assigned by the operating system ⁸. If we now observe an incoming connection on a machine, there is no way to map that incoming port number to a process and/or application, since the port number has been assigned just for that connection.

Therefore we may only be able to map outgoing connections to application identifiers. Next we will describe that process in the context of the Bazooka application host. In the outgoing connection in Listing 4.7 (first line) we can see that *netstat* returns a *process_id* and the *process_name*. The *process_name* does not reveal the application name, since processes have widely differing ways of being executed, resulting in no consistent mapping to application names. To do the mapping in our case study, we relied on

⁸It is also possible for a process to bind to a specific local port. From our experience, this is common for server processes, since the location of their service has to be known to its clients. It is uncommon for client processes, since any port number works equally well, and a client process binding to a specific port number might get the problem of the port number already being occupied. Letting the operating system handle the concern alleviates the client process from the complexity of finding a free port number.

implementation details of Bazooka: When a process is started on a Bazooka app host, it is executed within a Linux container (LXC [95] [96]). These containers are isolated using Kernel Control Groups (cgroups [97]). Via the `proc` virtual filesystem, it is possible to access the cgroup information for a specific process, which in turn contain the name of the container. Listing 4.8 shows an example output of the cgroup information. Bazooka uses the following scheme for the container names:

```
{application}-{process}-{git-commit-sha}.
```

We are able to extract the application identifier from that container name, and thus are able to map the source application for outgoing connections.

```
1 1:perf\_event,net\_cls,freezer,devices,memory,cpuacct,cpu,cpuset:
2 /lxc/threaded-comments-web-b3d7941-22160
```

Listing 4.8: Cgroup information example from the `proc` virtual filesystem retrieved via `cat /proc/20471/cgroup` in our case study

Destination application We have just shown that it is possible to obtain the source application identifier for *outgoing connections*. Now we will show how to obtain the destination application identifier for such connections. The destination of a connection (or *service location*) has the property that it has to be discoverable for the process wanting to connect to it. Commonly this discovery from a service discovery key to a service location is done via a service discovery mechanism. We are interested in doing this process in reverse, thus mapping from service location to the service discovery key and the to the application identifier. We call this process “reverse service discovery” and introduced several methods for it in the previous subsection 4.2.3 *From deployment configuration*. The difference to the earlier execution and here is that we now only have IP addresses to work with. For example in the earlier listing Listing 4.7, the destination application has the IP address/port number pair of `192.168.100.3:1001`. Next we will explain the execution for the Bazooka and Glimpse reverse service discovery:

Bazooka A service that is deployed with Bazooka is accessed via a port on the Bazooka load balancers. Given that we know the IP addresses of the Bazooka load balancer and have access to the HAProxy config file, we may map the application identifier the same way as introduced in earlier in subsection 4.2.3 *From deployment configuration*.

Glimpse When we introduced reverse service discovery for Glimpse in earlier subsection 4.2.3 *From deployment configuration*, we saw that the domain name format makes it possible to directly parse out the *product* and therefore application identifier from a

Glimpse service discovery domain.

To do the same in this method, we have to first map the IP address and port name to the Glimpse domain name. The Glimpse service discovery mechanism is based on BIND [68], which holds the information about the DNS mapping in a zonefile. Listing 4.9 shows an example excerpt from that zonefile. We can see that the product *threaded-comments* has three instances. Each of these is identified with a physical domain name (e.g. *app08.internal.example.com*) and a port number (e.g. *1001*). Let's assume that we try to resolve the IP address *192.168.100.3:1001* from earlier netstat example Listing 4.7. We first have to map the IP address to a physical domain name. We may do so with a reverse DNS lookup from address to domain name (as specified in RFC [98] section 5.2.1.). Our example IP address *192.168.100.3:1001* would then map to the physical domain name *app08.internal.example.com*. We may use that physical domain name and the port number for a lookup in the zonefile, which leads to the Glimpse domain *http.web.prod.threaded-comments.ca* and the product/application identifier *threaded-comments*⁹.

1	<code>http.web.prod.threaded-comments.ca 5 IN SRV 0 0 1000 app08.internal.example.com</code>
2	<code>http.web.prod.threaded-comments.ca 5 IN SRV 0 0 1001 app08.internal.example.com</code>
3	<code>http.web.prod.threaded-comments.ca 5 IN SRV 0 0 5671 app78.internal.example.com</code>

Listing 4.9: Glimpse zonefile example excerpt

In our case study we executed the connection gathering and analysis with both reverse service discovery methods on March 14 2014. Next we will describe the process we used:

- We executed the connection observations sequentially on all Bazooka application server. On each server, we executed netstat 50 successive times with a 5ms delay between the executions. We then collected the cgroup records for all processes on

⁹It might be possible, to do reverse DNS lookup for SRV records, similar to reverse DNS lookups for IP addresses to domain names. Following the definition in RFC 6763 [67] and our experience, we did not find a solution for that, other than the one introduced using the zonefile directly.

each application host¹⁰.

- We collected the zonefile from a BIND server by Glimpse¹¹.
- We collected the HAProxy configuration file from a Bazooka load balancer¹².
- We manually determined the addresses of all Bazooka load balancers.
- To analyze the data, we took each collected network connection and tried to map that to an application dependency with the methods explained above.

As a result of the investigation, we were able to derive 260 connection pairs between processes running on a Bazooka app host and a remote host. Of these, we resolved 9 application dependencies via the Bazooka load balancers and 2 application dependencies via Glimpse. Figure 4.5 shows the resulting graph.

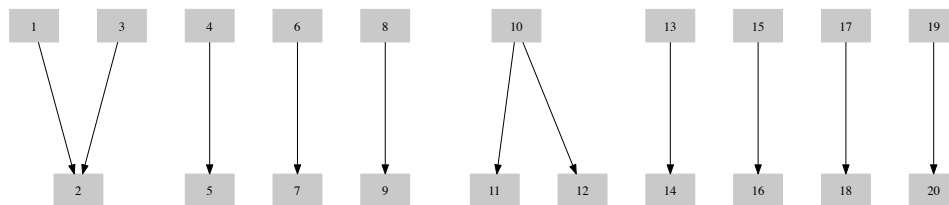


Figure 4.5: Dependency graph: Graph from network connections, resulting from execution in the case study. Application names are masked.

Discussion

Given the resulting graph in 4.5 we can see that only few applications and dependencies were identified. Also these are not well-connected. We conclude that at least the graph we created in the case study is not a satisfying representation of the actual state of the architecture we investigated.

¹⁰This is based on the assumption that processes do exist for an extended amount of time, but at least between doing the connection observation and retrieving the mapping between process ID and cgroup. We found that to be true, with process IDs only changing when a process was restarted (for example when it crashed or a new deployment was done) or when it was stopped for administrative reasons (for example when the number of running process instances per deploy is decreased)

¹¹All instances of BIND in Glimpse used the same zonefile.

¹²All instances of the Bazooka load balancers used the same HAProxy configuration file.

A major problem with this method is identifying the source application of a connection. In our case study investigation, we used an implementation detail of Bazooka to collect the information. This appears to be a brittle approach since there are no guarantees that this implementation detail might not change in future versions of Bazooka. It also is not possible to use the approach when other systems than Bazooka are used for deployment.

An easier problem to solve was mapping the destination application of a connection, given that we assume mechanisms for reverse service discovery to exist. During our investigations we were able to only resolve a small subset of destination applications (11 from 260). We attribute these to a low adoption of Glimpse. We also did not take “Semantic DNS CNAMEs” into consideration, which in earlier subsection 4.2.3 *From deployment configuration* made up for a significant share of services.

In our case study implementation, we probed for current connections at regular intervals. Therefore, we are not able to detect connections that did not happen at the times of probing. Examples might be application dependencies that have low traffic or only have traffic during certain times. The probability of detecting all connections increases if measurements are done over a longer time period.

We measured connections on all Bazooka application hosts. Depending on the number of hosts, this might be a substantial effort, with regards to orchestration of the measurement and impact on the running system. Also, the measuring itself might create big data volumes to be analyzed. These might be handled by analyzing the connections immediately and aggregating the results on the machine.

One problem we did not investigate are networking mechanisms like network address translation or proxy servers. Since these effectively hide the true origin IP address, they might impede on the possibility for reverse service discovery.

To conclude, we have shown that this method is **feasible**, even though in our case study we only detected a subset of the application dependencies. The **correctness** of the method depends on the mechanism to detect connections, as well the mechanisms for mapping source and destination IP addresses and ports reliably to application identifiers. In our case study, we have shown that it is possible to execute the method in an **automated** way.

Future Work

To measure connections, another option to investigate is measuring on the network directly. Networking equipment like switches do allow to the monitoring of connections and packets [99]. The problem here is deriving the source application, which in our case study involved knowledge that only exists on the applications hosts, thus making network capturing infeasible. Instead of the network, communication might also happen via other media. An example is the “Service Bus” in SOA as explained in [18] section 4.3.4.

We also see three options for future work regarding “reverse service discovery”:

- In earlier 4.2.3, we introduced the “reverse service discovery” method *Semantic DNS CNAMEs*. We did not use this method in the case study execution here but believe that it might improve the results significantly.
- More work can be done in exploring how to map IP addresses to application names. For example Chef might be used, in the same way as already explained in earlier 4.2.3.
- Packet inspection could be used to identify source applications. For example, the packet content could be annotated with the source application identifier. A practical implementation could be using the HTTP protocol with a custom header holding that information. This is similar to the way tracing frameworks work, which we will introduce in the following section 4.4.

4.3 Discussion

We introduced four methods for generating an application dependency graph from a deployed microservice architecture.

Manually creating a dependency graph We found this method to be the most complete and correct method. We showed that it is feasible to execute this method within an organization. By its nature, this method is only semi-automated. The gathering of annotations and graph generation may be automated. The creation and maintenance of dependency information is manual. This impedes on the feasibility of the method (due to lack of knowledge with the executing people) and the correctness (due to lack of knowledge with the executing people, differing willingness of people in the organization to create annotations, continuous effort to adapt annotations to change in architecture).

From service interface modules We found the assumptions of this method not to be fulfilled in our case study. To correctly identify internal application dependencies, these would have to be deliberately encapsulated, which appears to be a significant change to existing codebases. Also, the problem of mapping module names to application dependencies seems to only be solvable by manual mappings. We conclude that this method might be feasible, but would demand significant effort to change existing structures.

From deployment configuration We found this method to be feasible and potentially providing correct results. The execution of the method can be automated. In the case study execution, we found that only some service dependencies could be mapped to application identifiers, since reverse service discovery was not always possible. Also we only investigated deploys done with the *Bazooka* deployment system, thus we did not capture all deployed applications.

From network connections We found this method to be feasible within our case study. Problematic was mapping the applications, which limited the method in our investigations.

Since in our case study only *Manually creating a dependency graph* gave us a complete picture of the architecture, we will continue using the dependency graph we constructed from our case study with that method in the following chapters.

We did an informal qualitative evaluation of that graph with engineers of the case study company. A common reaction was to search for the applications the particular engineers were concerned with in their daily work. From there they analyzed and discussed incoming and outgoing edges. It occurred that existing edges were surprising to engineers and showed unexpected dependencies. Circular dependencies were a topic of concern for engineers, which were identified and discussed with the help of the dependency graph. A common concern for the dependency graph was as to which extent it represents the currently deployed architecture. We also learned that other engineers in the past tried to create similar graphs manually. They never created a meaningfully complete graph of all applications, but rather either only looked for a small subset, usually from the perspective of one application or they made a high-level architecture diagram which did not follow a consistent or holistic entity model. Other architecture overviews we encountered were data flow diagrams and request flow diagrams. Both were usually based on actual deploys of applications and only featured a limited subset of the architecture.

We believe that *From deployment configuration* and *From network connections* are feasible automated ways to generate application dependency graphs and might be fruitful for

future work. They both make assumptions towards the infrastructure tooling, especially with regards to reverse service discovery. It is to be investigated, how these can be satisfied without compromising other requirements, like low system complexity or scalability.

A problem with all methods was to distinguish applications that are currently deployed in the production deployment environment. We proposed criteria for manually filtering applications in manual creation (among them excluding infrastructure, deprecated and prototype applications). The same criteria were then used for manually filtering applications in the other methods as well. With all methods we hoped to find ways of running them highly-automated, but eventually did not succeed in doing so. All methods needed manual intervention at some point, for example when filtering for applications and deploys that are part of the production deployment environment.

If microservice architecture continue to grow in popularity, we expect “programmable infrastructure” to become more prevalent. We see programmable infrastructure as the attribute of systems to enable the planning, creation, maintenance and monitoring of an architecture. These systems will then also allow programmable interaction, which in turn allows for more sophisticated analysis of the architecture, just as we have executed them in this chapter.

4.4 Future work

A method worth investigate more is deriving service dependencies from within source code. We touched onto one option for this in the “Future work” section of subsection 4.2.3 *From deployment configuration*, where we proposed extracting service locations or service discovery keys from the source code. Another option would be to annotate dependencies to services in the code e.g. with a comment.

Tracing frameworks like Dapper [100] or Zipkin [101] allow for collecting information on how requests travel through a system. This assumes a request/response style communication originating in one part of the system (usually the interface to the user). Each of these request gets an identifier, which is carried through the architecture when more requests are made to subsystems. This tracing information is then collected and made available for analysis. Even though initially meant for performance monitoring, tracing would also allow to extract service dependencies.

An interesting angle for future work is using a finer granularity when discovering dependencies. In our work we used applications and application dependencies. A

more fine-grained approach would be by using processes. An even more detailed approach would be splitting applications or processes into functionalities and modeling dependencies between these. For example, if a service provides several HTTP resource locations, each of these could be seen as a functionality, to which other application functionality might have dependencies. Each resource location in turn would be a functionality with own dependencies. This would allow for more sophisticated dependency modeling, which in turn could also lead to new angles in dependency modeling.

4.5 Summary

The goal of this chapter was to evaluate ways of creating a dependency graph from a deployed and running microservice architecture. A dependency graph is a directed graph with applications as vertices and service dependencies as directed edges from service consumer to service provider. We proposed four novel methods for creating dependency graphs, all of which we executed in the context of our case study in a real and deployed microservice architecture. We then evaluated each method after the aspects feasibility, correctness and automation.

Of the four methods, 4.2.1 manually annotating service dependencies and semi-automatically collecting these for constructing the dependency graph was the most viable option, since it provided the most complete view on the deployed architecture. The graph we constructed from our case study with that method is visible in Figure 4.2.

5 Constructing qualitative fault trees

In this chapter we describe how a dependency graph can be transformed into a qualitative fault tree. First we lay the basis for our algorithm by describing the requirements towards the dependency graph and assumptions regarding failure semantics of applications. We then describe our algorithm and finish the theory section by introducing related work. Furthermore we describe how we executed our algorithm in the context of the case study and end the chapter by discussing our findings and future work.

5.1 Theory

In this section we explain our algorithm to transform a dependency graph into a qualitative fault tree. A dependency graph is a graph representing applications as vertices and dependencies between these applications as directed edges (see earlier chapter 4 *Constructing dependency graphs*). The algorithm has several assumptions which we will describe before we explain the algorithm itself.

5.1.1 Dependency graph requirements

For our algorithm to work, the dependency graph has to fulfill the following requirements:

Directed graph In a directed graph, all edges have a direction from a source vertex to a destination vertex. In a dependency graph, all edges fulfill this property, since dependencies between applications are directed. If this requirements is not fulfilled, our algorithm could not construct a fault tree, since it would be impossible to detect the direction of failure propagation. Interpreting undirected edges as symmetric (being directed in both directions) is not a feasible solution, since the graph needs to be acyclic, as explained next.

Acyclic graph An acyclic graph is a directed graph which has no cycles. This means that for every vertex, if one recursively follows the outgoing edges, one will never return to the start vertex. Fulfilling this requirement in a dependency graph might be hard, since service dependencies can be cyclic. If this requirement is not fulfilled, it will not be possible to generate a fault tree with our algorithm, since the generated tree will grow indefinitely.

Rooted graph In a rooted graph, one vertex is labeled as a special vertex, distinguishing it from others. In our case, this can be fulfilled by manually choosing a root vertex from the graph. If this requirement is not met, it is not possible to identify the vertex for the TOP event for the fault tree and therefore the algorithm can not be executed. Given a microservice architecture that implements a SaaS, interesting root vertices are applications that directly serve clients, since these represent entry points into the system.

When executing the algorithm, it will practically be executed on a subgraph of the dependency graph, which includes all vertices and edges reached when recursively following the outgoing edges from the root vertex. It is sufficient if only that subgraph is directed and acyclic.

5.1.2 Failure semantics

In this section we describe the assumptions about failure semantics that we base our algorithm for fault tree construction on. In short, these are as follows: 1. Applications might fail because of **internal** failures. 2. The failure of one application might **propagate** failure to all applications that depend on it. Next we will describe these assumptions in more detail.

Given that we have a dependency graph, each vertex represents one application that is part of the microservice architecture. These applications might fail individually¹.

In order to exhibit a failure, the application needs to have experienced a fault. We assume that there are only two possible fault types:

1. An application might experience an *internal fault*. We specify that this fault might not be because of another application's failure. Apart from that, the nature of

¹To be correct with the terminology from the earlier subsection 2.1.1 *Dependability threats*, it should be "might individually be in erroneous states, which manifest as failure events when the service is used". For reasons of comprehensibility, we only use the terms "fail" and "failure" to refer to these cases in this section.

these internal faults it not specified more here².

2. An application might experience an *external fault*. A external fault in an application A occurs, when an application B has a failure and application A has a service dependency on application B.

The second fault type is based on service dependencies. We defined service dependencies in earlier section 2.2 *Software environment terminology* as the fact that application A is the service consumer of another application B, which in turn is based on the fact that one process of application A at runtime opens a network connection to a service implemented by application B.

On a more abstract note, a dependency is defined by Knight [9] as following: “The dependency of system A on system B represents the extent to which system A’s dependability is (or would be) affected by that of system B”. In our case we assume that the dependence between systems is very strong, leading to the immediate propagation of failures.

In practice, the dependence might not be as strong as we assume it here. Commonly fault tolerance means are used to limit the impact of dependency failures. In our concrete example this might include using spatial or temporal redundancy, or continuing in a degraded state, with some functionality of the application being disabled but others being unaffected. We explicitly do not include these means of fault tolerance in our model of failure semantics.

The practical manifestation of failures is not of concern a here. Following the failure semantic model for distributed system by Cristian [10], failures could be response, timing, omission or crash failures. All result in the same failure propagation results as proposed here.

5.1.3 Construction algorithm

We introduced fault trees in subsection 2.1.5 *Fault tree analysis* as a technique for structured dependability analysis. Via a graphical diagram it helps to visualize which events may contribute to a TOP event failure and how these are interrelated with each other. We described earlier that fault trees are used to comprehensively describe the

²Examples for internal faults might be design or programming mistakes when considering the application itself or failures of the execution environment. We do not consider these internal faults further in our algorithm. As future work, these could be investigated further. For example, they could be modeled with own fault trees then.

causes for a TOP event failure. In this work, we limit the fault tree analysis to *internal* faults in an application and *propagated* faults due to failing service dependencies.

Algorithm Given that we have a rooted, directed, acyclic graph, we may construct a qualitative fault tree with the following algorithm:

1. Turn root vertex into TOP event
2. Create OR-gate and connect it to TOP event
3. Create basic event with name of root vertex and connect it to OR-gate
4. Follow each outgoing edge of the *current* vertex (beginning with root vertex) to the *next* vertex
 - a) Create intermediate event with next vertex name and connect it to OR-gate of the current vertex
 - b) Create OR-gate and connect it to intermediate event
 - c) Create basic event with next vertex name and connect it to OR-gate
 - d) Repeat step 4 recursively for next vertex

Figure 5.1 shows an example dependency graph. Vertex “A” is the root vertex. Figure 5.2 shows each algorithm step while generating the fault tree. Since vertex “A” only has one outgoing edge, algorithm step 4 is executed only once for vertex “B”. If vertex “A” would have more outgoing edges, algorithm step 4 would be executed for each of these. If vertex “B” would have outgoing edges, algorithm step 4 would be executed for each of these recursively.

It is to note that the dependency graph has the vertices “D” and “E”, which we did not utilize in the algorithm execution. This is due to the fact that neither “A” nor “B” have outgoing edges to “D” or “E”. Furthermore, “D” and “E” have symmetric edges and therefore form a circle. Since this does not meet our requirement of acyclicity, it is not possible to use our algorithm with them.

5.1.4 Related work

Software fault tree analysis after Leveson et al. [102] and Rausand et al [6] (section 4.3.8) is a software verification technique. It takes some undesired output of software as TOP event and then based on the internal structure of the software tries to derive if and how this output might occur. We do not use this technique in our work, since in our fault tree analysis we are not concerned with the internal structure of software but rather the system’s view between software applications.

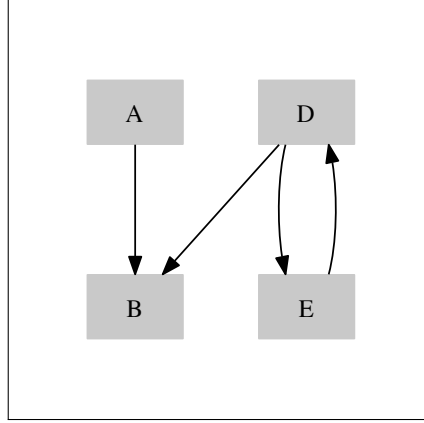


Figure 5.1: Example dependency graph for fault tree algorithm example. Root vertex is vertex “A”.

Lapp and Powers [103] proposed an algorithm for transforming directed graphs to fault trees that is similar to the one we propose here. It differs by taking more qualifying information into consideration and therefore constructing more expressive fault trees.

5.2 Case study execution

In this section we describe how we executed the algorithm in our case study. As basis for this we use the dependency graph in Figure 4.2 generated in subsection 4.2.1 *Manually creating a dependency graph*. From that dependency graph, we chose two root vertices to construct fault trees from: Vertex “26” and vertex “73”.

Vertex “26” This vertex represents an application that acts as interface between the microservice architecture and the clients of the platform. Specifically, this application implements an API with the HTTP protocol, which is then used by the users’ web browsers for functionality on the website <http://soundcloud.com>. Therefore it does not have incoming edges, since clients (in this case browsers) are not represented in our investigation³. When looking at the relevant subgraph spanning from vertex “26”, it can be seen that there are cycles in that subgraph, revolving around vertex “5”. We will comment in the *evaluation* paragraph below on why these exist. To resolve this issue, we manually determined, which of the cyclic edges was more relevant, and removed

³see earlier section 3.1 *Architectural style* for more details

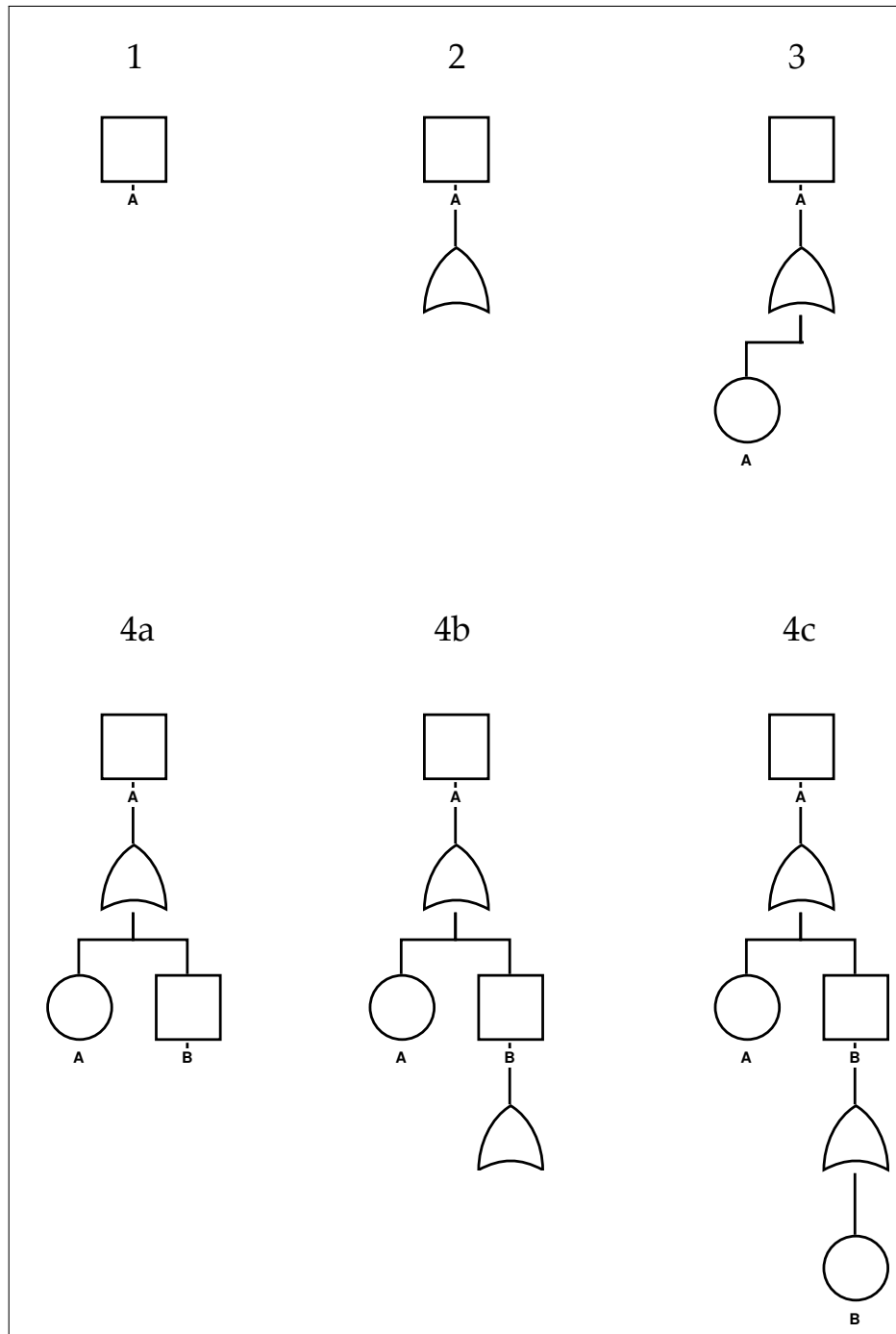


Figure 5.2: Visualization of the algorithm for transforming a dependency graph to a qualitative fault tree. This example is based on the dependency graph in Figure 5.1

Table 5.1: Shortened application identifiers of subgraph for vertex “73” shown in Figure 5.3.

Full application identifier	Shortened application identifier
github.com/soundcloud/soundcloud	soundcloud
github.com/soundcloud/threaded-comments	threaded-comments
github.com/soundcloud/authenticator	authenticator
mysql.com	mysql
memcached.org	memcached

the other edge. Figure 5.5 shows the resulting fault tree for this root vertex. Please note that the fault tree is cropped, due to limitations with drawing large fault trees in the fault tree construction software we used. For reference and evaluation, Table 5.3 shows statistics over the size of the fault tree with comparison to the size of the dependency graph it was based on. A second note is that for technical reasons the naming of the vertices in Figure 4.2 does not coincide with the naming of events in Figure 5.5.

Vertex “73” (threaded-comments) Similar to previous vertex “26”, vertex “73” represents an application that exposes an API with the HTTP protocol to users’ internet browsers. Therefore its vertex does not have incoming edges in our microservice architecture. We chose this vertex because we were interested in using a small subgraph for investigations in the following chapter 6 *Constructing quantitative fault trees*. For better visibility, we visualized the subgraph for vertex “73” in Figure 5.3. The vertices of the subgraph carry the original unmasked application identifiers as were assigned in subsection 4.2.1 *Manually creating a dependency graph*. When extracting this subgraph, we made one major simplification: We did not include the outgoing edges for the vertex “5” in the dependency graph respectively application “github.com/soundcloud/soundcloud” in the subgraph. Our intention was to keep the subgraph small, in order to allow for further investigation in chapter 6 *Constructing quantitative fault trees*. We were not able to find another suitable subgraph which would allow these investigations without making the simplification we did here. The fault tree resulting from executing our algorithm on the subgraph can be seen in Figure 5.4.

Please note that for brevity in this work, we have shortened the application identifiers as shown in Table 5.1. Furthermore, Table 5.2 shows descriptions for the applications involved.

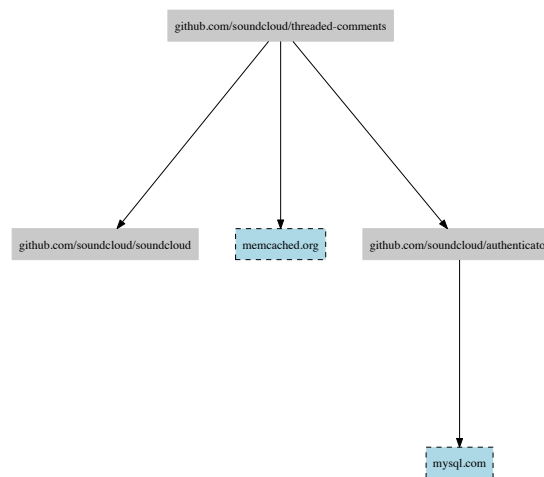


Figure 5.3: Subgraph for vertex “73” from case study figure Figure 4.2. The outgoing edges for vertex “5”/application “github.com/soundcloud/soundcloud” have been removed. The vertex names have been unmasked to carry their original application identifiers.

Table 5.2: Description of applications of subgraph for vertex “73” shown in Figure 5.3.

Shortened application identifier	Description
soundcloud	Former-monolithic main business logic application
threaded-comments	Application to group and cache comments
authenticator	Authentication and authorization of requests
mysql	Relational database
memcached	In-memory key-value, used as cache store

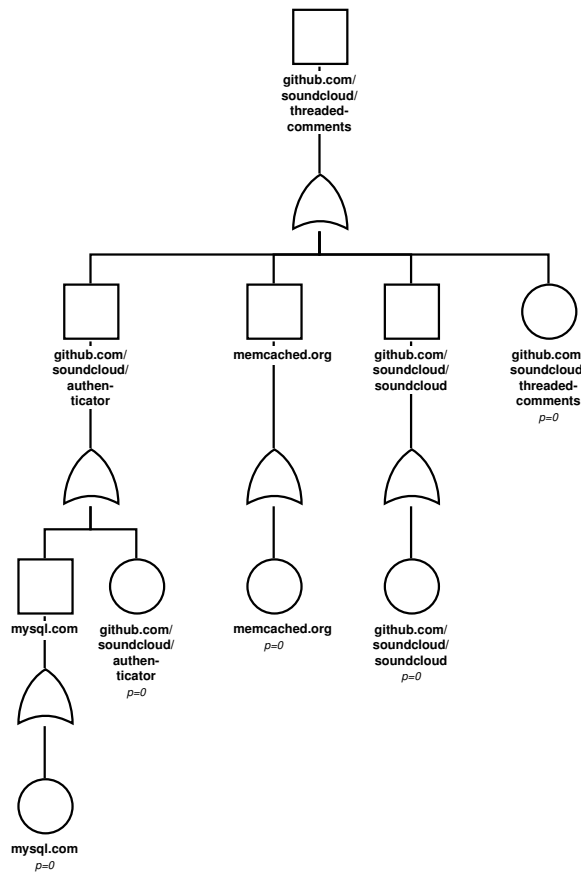


Figure 5.4: Qualitative fault tree for vertex "73"/application "github.com/soundcloud/soundcloud" following the dependency graph from Figure 5.3.

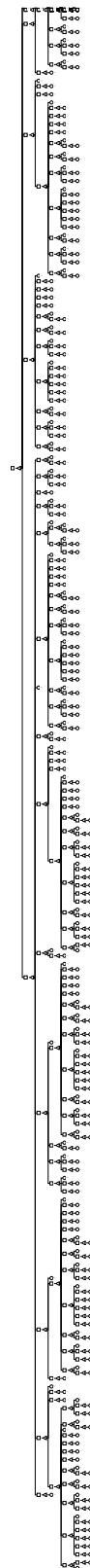


Figure 5.5: Qualitative fault tree for vertex "26" from case study dependency graph figure Figure 4.2 (cropped)

5.3 Discussion

As we have seen with Figure 4.2 and Figure 5.3, it is possible to generate fault trees from dependency graphs with our algorithm. In this section we will discuss these and the algorithms.

The most notable fact of our algorithm is that it only uses OR-gates as logic gates. This leads to the fact that each basic event is a minimal cut set and therefore each basic event is a “single point of failure” for the TOP event. As a result of this fact, employing analytical methods that are based on cut set analysis will yield low insights. We believe that the architecture of the system is not holistically expressed in the generated fault tree, since all basic events have the exact same influence on the TOP event. As major reason for that structure we see the exclusion of fault tolerance in the construction process. We will discuss opportunities for including these during automatic execution in the following future work section.

Table 5.3 shows the sizes of the dependency graphs and the fault trees that our algorithm generated for them. It can be seen that the fault trees have magnitudes more elements and edges than the dependency graphs. One reason for the size increase is the duplication of dependency subgraphs in the fault tree: If a vertex has several incoming edges, the subgraph spanned by its outgoing edges will be present several times in the fault tree, since it will be duplicated for each incoming edge. Another reason for the size increase is the usage of intermediate events and logic gates, which are not necessary in the dependency graph.

One of the assumptions of our algorithm is that there are no circles in the graph. As we have seen in the case study, circles do appear in reality, which hinders using the algorithm. We manually resolved the dependencies in the case study, which we believe is a poor solution.

We believe that for an engineer to get an overview over the architecture, dependency graphs are better suited than fault trees, due to their significantly smaller size. We see potential in improving the algorithm as explained in the following Future work section. We also see potential of the fault trees as basis for quantitative fault trees, as explained in the following chapter 6 *Constructing quantitative fault trees*.

Table 5.3: Sizes of dependency graphs and the generated fault trees

Dependency graph with root vertex "26" (not visualized)	
Vertices	49
Edges	97
Fault tree for vertex "26" Figure 4.2	
Elements	1146
Basic Events	382
Edges	1145
Dependency graph with root vertex "73" Figure 5.3	
Vertices	5
Edges	4
Fault tree for vertex "73" Figure 4.2	
Elements	15
Basic Events	5
Edges	14

5.4 Future work

In the discussion we have seen that the size of the fault tree is problematic. We see two opportunities for future work to improve on that:

- To reduce the number of elements and edges in the fault tree, we see the following simplification: All vertices with no outgoing edges will follow the same pattern of being represented by one intermediate event, one OR-gate and one basic event. These can be compacted by removing the intermediate event and OR-gate, and connecting the basic event directly to the parent OR-gate.
- We have discussed that the same subgraph may appear several times in the fault tree. To improve on that, a "Transfer In" element⁴ may be used.

The fault trees our algorithm generated are limited to OR-gates as logic gates only. This limits the applicability of analytical methods based on cut sets. Thus the goal of future work should be to improve the expressiveness of the fault tree. Next we will introduce

⁴A "Transfer In" element is a fault tree element that allows to include another fault tree. The "Transfer In" element then represents the TOP event of the included fault tree.

several opportunities for that:

Manual extension The generated fault tree could be taken as a basis for manual extension by engineers. Examples for extensions are including runtime environments of applications, non-technical influences like human interaction, or the methods introduced below.

Application fault tolerance Future work should investigate representing fault tolerance of applications in the fault trees. Examples for application-internal fault tolerance mechanisms to investigate are temporal and spatial redundancies (which might result in using AND-gates or K/N-gates in the fault tree) as well as degraded service (which might allow for splitting applications into finer-grained functionalities).

Different application granularity In earlier section 4.4 *Future work* for generating dependency graphs, we already discussed potential for different granularities for applications. These might be deploys, processes or functionality. Each of these would allow different algorithms, which in turn would result in different fault trees. These could aid the expressiveness of the fault tree.

Infrastructure applications In our investigations, we did not take infrastructure applications (like deployment systems or traffic management systems) into consideration. Future work could investigate how to include these in the algorithm. This would improve the scope of the fault tree in the direction of a more holistic view. It might also help to model system fault tolerance, since it is sometimes realized through infrastructure applications (e.g. load balancers).

5.5 Summary

In this chapter we suggest qualitative fault trees as a way of modeling failure propagation in microservice architectures. We proposed a novel algorithm for transforming dependency graphs into qualitative fault trees. The algorithm respects two kinds of failures: internal failures from applications themselves and external failures propagating through the architecture via service dependency failures. Fault tolerance mechanisms were not considered. We executed the algorithm for one application of the case study's dependency graph with the resulting qualitative fault tree visible in 5.4.

6 Constructing quantitative fault trees

Based on the qualitative fault tree structure, we would like to quantify the failure probabilities of the basic events to calculate the probability of the TOP event. The actual TOP event probability itself may not be useful in assessing the dependability of a system. But it is possible to evaluate how changes to basic event probabilities affect the probability of the TOP event.

We base our investigation in this section on the fault tree from previous subsection 5.1.3 *Construction algorithm*, specifically shown in Figure 5.4.

To calculate the probability of the TOP event, we use the transformation rules as described earlier in subsection 2.1.5 *Fault tree analysis*, resulting in the following equation:

$$\begin{aligned}
 P(TOPEvent) = & \\
 & P(mysql) + P(authenticator) - P(mysql) * P(authenticator) + \\
 & P(memcached) - \\
 & (P(mysql) + P(authenticator) - P(mysql) * P(authenticator)) * P(memcached) + \\
 & P(soundcloud) - \\
 & (P(memcached) - (P(mysql) + P(authenticator) - P(mysql) * P(authenticator)) \\
 & \quad * P(memcached)) * P(soundcloud) + \\
 & P(threaded-comments) - \\
 & (P(soundcloud) - (P(memcached) - (P(mysql) + P(authenticator) - P(mysql) * P(authenticator)) \\
 & \quad * P(memcached)) * P(soundcloud))) * P(threaded-comments) \quad (6.1)
 \end{aligned}$$

To solve the equation probabilities for all basic events are needed. Rausand et al [6] (section 4.3.5) introduce three categories for approaches for handling probabilistic analyses of software failure. All of these assume a fault tree that includes software, hardware and other events.

1. **Assume the probability of software failure is zero** This approach assumes that software never fails. It therefore limits the probabilistic analysis to the other components. We use this approach for software components that the deployed method may not derive probabilities for, therefore we set these probabilities to 0.
2. **Assume the probability of software failure is one** This approach assumes that software always fails. It is therefore very pessimistic, since even though we may assume that software will eventually fail, usually it is more dependable than that. We do not use this approach in our works.
3. **Separate the analysis of the software** This approach proposes to use separate analysis methods for software and other components. The methods we propose here may be seen in the context of this category.

For assessing the basic event probabilities we investigated the following methods:

1. Source code metrics
 - a) Lines of code
 - b) Code churn
2. Operational availability
 - a) External heartbeat measurements
 - b) Production traffic measurements

We executed each of these methods in the context of our case study. Please note that a quantitative analysis of the methods was out of scope for this work. We executed the methods to show their feasibility. We do however evaluate each method qualitatively.

6.1 Via source code metrics

Source code metrics provide insights about the state of a software by analyzing source code and its meta data. In this section we describe two methods for deriving failure probabilities from source code metrics. For each method we explain how we executed it in our case study and discuss its results.

6.1.1 Lines of code

The *lines of code* (short: *LOC*) of an application are all text lines that contribute to the codebase of it. Earlier in section 2.2 *Software environment terminology* we assumed that each application has exactly one codebase and that we are able to map the application identifier to its codebase. In our case study this assumption is true for internal applications, but not for external applications. Thus we manually linked the external application identifiers to the appropriate codebases.

We manually acquired the codebases for each application in the fault tree. We then counted the lines of code for each codebase. Our measurement of LOC took all lines of text in the codebase into consideration. This included tests, deployment and build scripts, example code, documentation and comments. We did not take shared library code into consideration, except if they were included in the codebase (e.g. since they have been copied in).

The internal applications *soundcloud*, *threaded-comments* and *authenticator* were measured with their versions from 28 April 2014. The external applications *mysql* and *memcached* were measured with the version numbers that were deployed in the case study on 28 April 2014. The results can be seen in subsection 6.1.1.

To use the lines of code in the fault tree, we have to transform them to failure probabilities. We do this based on the assumption that per 1000 lines of code (short: *KLOC*), there are a number of bugs present in the code. McConnell et al [104] surveyed papers comparing *number of defects per KLOC*. The values range from 0.1 defects to 50 defects per KLOC. We optimistically chose the lower bound *0.1 defects per KLOC* as ratio to estimate the number of defects in a codebase. We now have to transform these to failure probabilities, in order to use them in the fault tree. We do so by assuming that the higher number of defects has a failure probability of 0.1. We may then normalize the other defect numbers accordingly. Table 6.1.1 shows the results of this method.

Given the resulting failure probabilities, we may now use these as basic event failure probabilities in earlier Equation 6.1 to calculate the TOP event probability. The resulting TOP event probability is **0,10704**.

Discussion

Both the functions for calculating defects per line as well as transforming defects to failure probabilities have been chosen by us as “good guesses”, since these were not in the focus of this work. We believe that more work is needed in choosing appropriate

Table 6.1: Lines of code calculation in the case study with accompanying failure probabilities.

Application	Lines of code	Estimated number of defects	Failure probability
soundcloud	635436	63,5436	0,007402703
threaded-comments	1163	0,1163	0,000013549
authenticator	6747	0,6747	0,000078601
mysql	8583837	858,3837	0,1
memcached	28354	2,8354	0,000330318

functions and values.

Given the *lines of code* numbers from table 6.1.1, we can see there is a large variance in the size of codebases in our case study. For example the *mysql* codebase is more than 7000 times larger than the *threaded-comments* codebase. The failure probabilities from this method are proportional to the lines of code. Therefore the calculated TOP event probability is highly influenced from the big codebases but does not respect smaller codebase well.

We calculated the *lines of code* by summing up all text lines in a codebase. An opportunity for future work would be to investigate, in how much these numbers change, if only the code that is immediately used for running processes is taken into consideration.

6.1.2 Relative code churn

Code churn is a measure based on summarizing the added and changed lines of code in a codebase for a period of time. In this subsection we will explain, how we used relative code churn to calculate the failure probability of an application.

In our case study, all internal applications' codebases were tracked in Git [33]. Therefore we were able to use the approach by Kraaijeveld [105] for measuring code churn: For a git branch, gather all parent commits for the defined time period. For each commit, summarize the number of added lines. Summarize these numbers over all commits. The resulting number is the absolute code churn.

As shown by Nagappan et al [106] relative code churn may be used as a predictor of defects in source code. Therefore, we normalize the absolute code churn by the lines of

code at the time of the start of the measured period (as explained in [106] as measure *M1*).

In our case study, relative code churn for own applications is relevant to the behavior of the deployed architecture, since deploys happen often (often within hours of the code changes being present in the codebase). We may therefore assume that the codebase's state matches the code that is deployed.

Table 6.2 shows the results from calculating relative code churn over three time periods of one week, as well as over a time period of four months.

Table 6.2: Relative code churn in the case study calculated over several time periods.

Application	Beginning LOC	Absolute churn	Relative churn
25 March 2014 - 31 March 2014			
authenticator	6981	42	0,006016
soundcloud	635434	2478	0,003900
threaded-comments	1163	1	0,000860
01 April 2014 - 06 April 2014			
authenticator	6839	32	0,004679
soundcloud	635571	326	0,000513
threaded-comments	1163	4	0,003439
18 April 2014 - 24 April 2014			
authenticator	6875	762	0,110836
soundcloud	635165	893	0,001406
threaded-comments	1163	0	0
01 January 2014 - 30 April 2014			
authenticator	6322	3805	0,601866
soundcloud	518904	33643	0,064835
threaded-comments	1163	40	0,029235

Assuming that relative code churn is a predictor for defect rates, we may also assume that it predicts the failure rates of the applications. Thus we use the relative code churn as basic event failure rates in earlier Equation 6.1 to calculate the TOP event probability. Table 6.3 shows the results for the different time periods.

Table 6.3: Relative code churn in the case study: TOP event failure probabilities after the fault tree for several time periods.

Application	Top event failure probability
25 March 2014 - 31 March 2014	0,010774
01 April 2014 - 06 April 2014	0,008610
18 April 2014 - 24 April 2014	0,112209
01 January 2014 - 30 April 2014	0,638560

Discussion

We calculated the code churn for internal applications only, but did not take the external applications *memcached* and *mysql* into consideration. We based this decision on the fact that the deploys of both applications in our case study were not upgraded with new versions in the time of our investigation. Therefore their code churn would not influence the deployed architecture.

We base our assumptions about mapping relative code churn to defect probabilities on Nagappan et al [106]. They investigated code churn on differences between software releases. We did our measurements in a continuous deployment environment without explicit releases. Instead of differences between releases we used arbitrary time periods to bound the code changes to include in the investigation. We did not investigate, how this change might alter the results. Future work should investigate the significance of relative code churn towards defect rates in a continuous deployment environment.

In our calculation of absolute code churn, we took into consideration all text in the codebase, regardless if it contributes to the running application. We did not evaluate how non-application text like tests, examples or documentation influence the results.

For calculating absolute churn, we did not take deleted code into consideration. This is due to the fact that in Git a modified line of code is expressed as an added and a deleted line, thus counting deleted lines would weight modified lines double and over-emphasize them against newly added lines. Furthermore we (similarly to [105]) assume that deleted code is a low-relevance churn operation.

We use relative code churn directly as failure probability. We may assume that the longer the time period for the measurement, the higher the code churn will be. This is visible in our results in Table 6.2, where the relative churn numbers for weeks are magnitudes lower than the churn numbers over months. Relative code churn may then

also exceed 1, when more lines have been added/changed than lines of codes existed at the begin of the time period. In that case, relative code churn could not be used anymore as failure probability directly, but would have to be normalized.

The code churn of a single application is highly influenced by the specifics of the development process used with regards to how changes to the codebase manifest themselves in the version control system. In our case study, all applications follow the same development process, loosely based on the model as outlined in [107]. There, different branches (code versions) are used for developing a feature. Once the development on a feature is completed, its changes are reviewed and then merged into a central branch (example names are *trunk*, *stable* or *master*). This branch is the one that is used while deploying. We also used that central branch for calculating code churn. Future work should investigate, to which extent other development processes influence the relative code churn and therefore the failure probabilities.

6.2 Via historical availability data

To calculate the TOP event probability in a fault tree, we need to assign failure probabilities to the basic events. In this section we describe two methods for deriving basic event failure probabilities from historical availability data.

6.2.1 Theory

Given that an application is deployed, we may measure the deploy's availability at runtime, which over a time period will provide us with the historical availability.

To acquire that data, we used two methods:

1. External heartbeat measurements
2. Production traffic measurements

In both measurements, we derived a time series for each deploy in the production environment. Each time step in the series holds the information if the deploy was either *up* or *down* in that time period.

Figure 6.1 shows an example of such a time series. As we can see, there were two time periods where the application was successively *down* (Minute 5 and minutes 9 to 10) as well as three time periods where the application was successively *up* (Minutes 1 to 4, minutes 6 to 8 and minutes 11 to 12). The duration of each *up* time period may also be

called the *time to failure*, thus the duration it took from getting to an initial *up* state to the next *down* state. Furthermore we may then calculate the average time it takes until failure, also called the *mean time to failure* or *MTTF*. Equation 6.2 shows how to calculate that from a time series. The execution for this with the example time series is visible in equation 6.3.

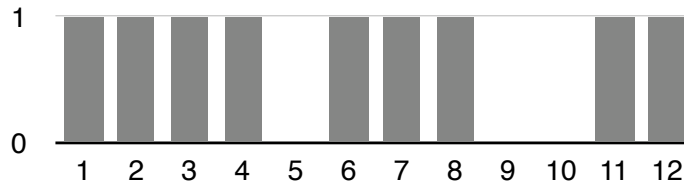


Figure 6.1: Historical availability: Example of time series for a time period of 12 minutes with 1 representing “up” and 0 representing “down” for the respective time step

$$MTTF = \frac{\text{Number of "up" steps}}{\text{Number of sequences with coherent "up" states}} \quad (6.2)$$

$$MTTF_{example} = \frac{9}{3} = 3 \quad (6.3)$$

We assume that during the time of our measurement the system has reached steady-state, meaning that if we would execute the measurements over the same length but at different points in time, we would get the same MTTF value.

The software systems we investigate are by definition repairable. For example after a failure occurred a software system may continue to work as expected afterwards due to self-repair mechanisms or, if the cause of failure is a crash, by being restarted and therefore repaired externally. We work under the assumption here that these system repairs are perfect, thus that after a repair the system is “as good as new”.

Given these assumptions, we may calculate the failure rate with equation 6.4:

$$\lambda = \frac{1}{MTTF} \quad (6.4)$$

As distribution for failure probability we assume an exponential distribution. Therefore the system is said to be “without memory”: the probability of failure at any point in time is the same, regardless of the time that passed already. We may calculate the failure

probability with equation 6.5:

$$Pr(t) = 1 - e^{-\lambda * t} \quad (6.5)$$

The variable t denotes the “mission time”. Given a specific mission time t , the probability $Pr(t)$ denotes the probability that the system has failed at or before that time. Intuitively we see that the longer the mission time is, the higher the probability that the system has failed will be.

Next we will describe the two methods we used for measuring the availability of systems. We executed both methods over the same period of seven days. We did not measure the applications *memcached* and *mysql* because no existing measuring mechanisms provided reasonable indicators for their availability.

6.2.2 External heartbeat measurements

To measure the availability of an application, we did regular requests to the deployed service instances of it. We will explain the method in detail next, then show our results from executing the method in our case study and end the section by discussing our findings.

Measurement method

To execute the measurements, we created our own measurement application (called *measurer*). It ran on one machine within the internal network, being able to do network requests to all deployed service instances. We measured three applications: *soundcloud*, *authenticator* and *threaded-comments*. Each application had one production deploy with many service instances. The instances were found via the Glimpse service discovery (as explained earlier in section 3.2.2 *Glimpse*). All services used the HTTP [8] protocol. For the measurement we chose one HTTP resource for each application. *Measurer* then requested that resource on all service instances of the deploy. Each HTTP request may either return an HTTP response with an HTTP status code, or may be a timeout if it has not returned after a certain period of time (in our case 30 seconds). For all resources we expected HTTP status code 200, if the call was successful. Thus, when a response with HTTP status code 200 was received, we defined that instance to be *up*. If the HTTP status code was equal to or bigger than 500, or a timeout occurred, we define the instance to be *down* for that period of time¹.

¹During the investigations we did not receive any other HTTP status codes like 201 or 301, thus we were not concerned with them here.

To alleviate load and limit impact of the measurements on the running system, we only requested 10 instances of each deploy at a time, but requested all instances of each deploy eventually within one measurement step. In Figure 6.2 are the total durations for querying all instances of an application in each measurement runs. For *authenticator* and *threaded-comments*, these always finished quickly. For *soundcloud* these show some spikes, resulting from some requests timing out at the 30 second mark we set as timeout interval.

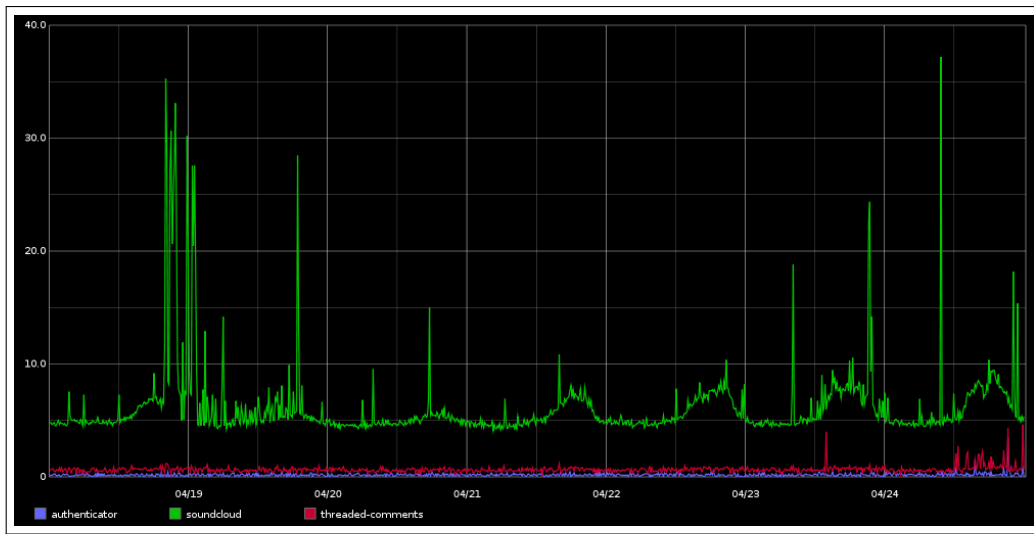


Figure 6.2: Heartbeat measurement: Duration of each measurement step for each application

One measurement happened every minute. The measurements were executed continuously for seven days between April 18 2014 and April 24 2014 inclusively. The results were stored in a *Graphite* server [69], which allowed for extracting the raw time series data for further analysis but also generated the plotted visual graphs shown in this section.

Results

The results of our measurements can be seen in Figure 6.3, Figure 6.4 and Figure 6.5 ². Each figure holds the measurements for one application, with the responses of all instances summarized for each time period, with a plotted graph per status code. If

²Due to the large time period represented in the plotted graphs, values are smoothed.

an instance returned HTTP status code 200, we interpret that as “*instance is up*”. If an instance returned a status code in the 5XX range or the request timed out, we interpret that as “*instance is down*”. Next we will comment on each figure.

Figure 6.3 soundcloud Around April 18 2014 3:00 there is an increase in 200 status codes that is due to a provisioning of more instances. Around April 23 2014 14:00 there are some spikes in 500/timeout, which peak at about 30 instances.

Figure 6.4 authenticator Around April 22 2014 11:30 there is an increase in up and down instances. This increase is due to the experimental deployment of a development branch on one instance. The development branch was tested with different configurations against live traffic, some of which configurations did not deliver correct functionality. At April 24 2014 12:00 onwards there are occasional spikes in 500/timeouts. These peak at 2 instances and at maximum last 3 minutes.

Figure 6.5 threaded-comments At April 23 10:00 there is a spike of 500/timeouts that lasts for 14 minutes and peaks at 23 instances.

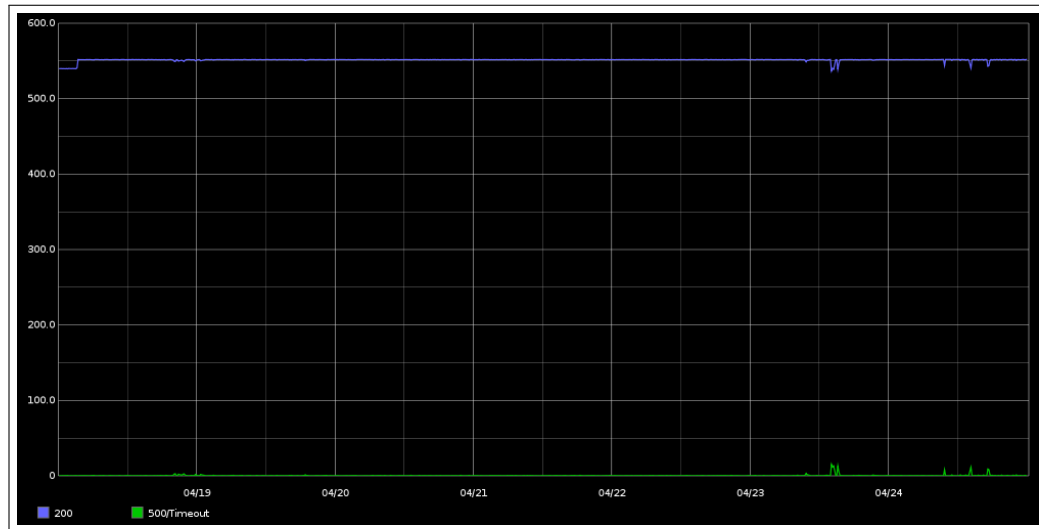


Figure 6.3: Heartbeat measurement: Summarized HTTP request status codes for each measurement step for application *soundcloud*

From the graphs it becomes apparent that when failures occurred, they never affected all instances of a deploy, but only a subset of them. The peak percentages of *down* instances relative to *up* instances are ~6% for *soundcloud*, ~20% for *authenticator* and ~57% for *threaded-comments*. Following this, we define that an application’s deploy is *down* in a certain time period, when more than 5% of all its service instances are *down*.

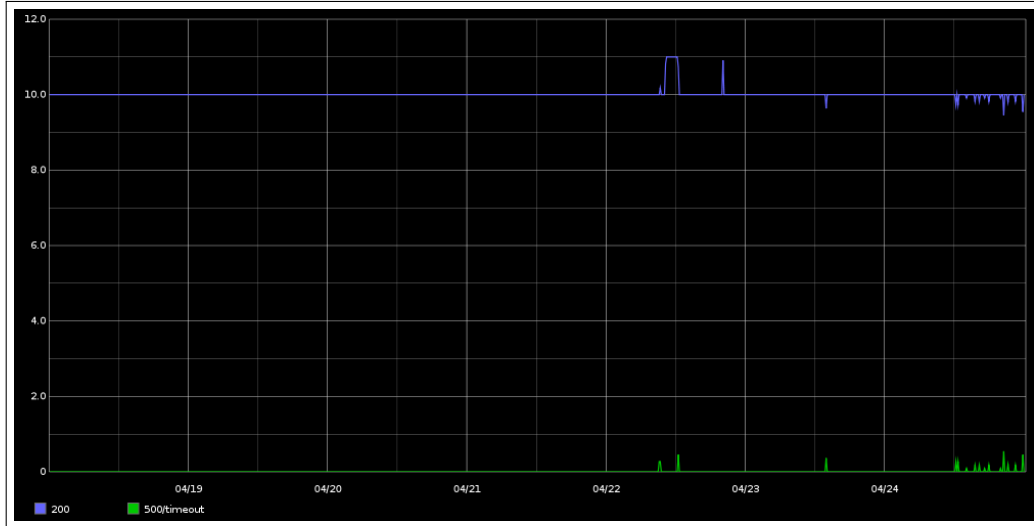


Figure 6.4: Heartbeat measurement: Summarized HTTP request status codes for each measurement step for application *authenticator*

Table 6.4 shows the results of our measurements as well as the calculated failure probabilities after the equations in subsection 6.2.1 *Theory*. As mission time we chose $t = 720 \text{ minutes (12hours)}$. Using these failure probabilities in the fault tree results in a TOP event probability of **0,96588**.

Table 6.4: Failure probabilities for application deploys measured via heartbeat measurements from April 18 2014 to April 24 2014 with *down* threshold 5%. The *mission time* for the failure probability calculation was 720 minutes (12 hours).

Application	#Datapoints	#Up	#UpSequences	MTTF	Failure probability
threaded-comments	10080	10066	2	5032,50	0,1333
authenticator	10080	10053	17	591,29	0,7040
soundcloud	10080	9996	28	356,96	0,8669

Discussion

Measurer requested each instance once per time period. We assume that the result of that single request represents the state of the instance for the whole time period. We do

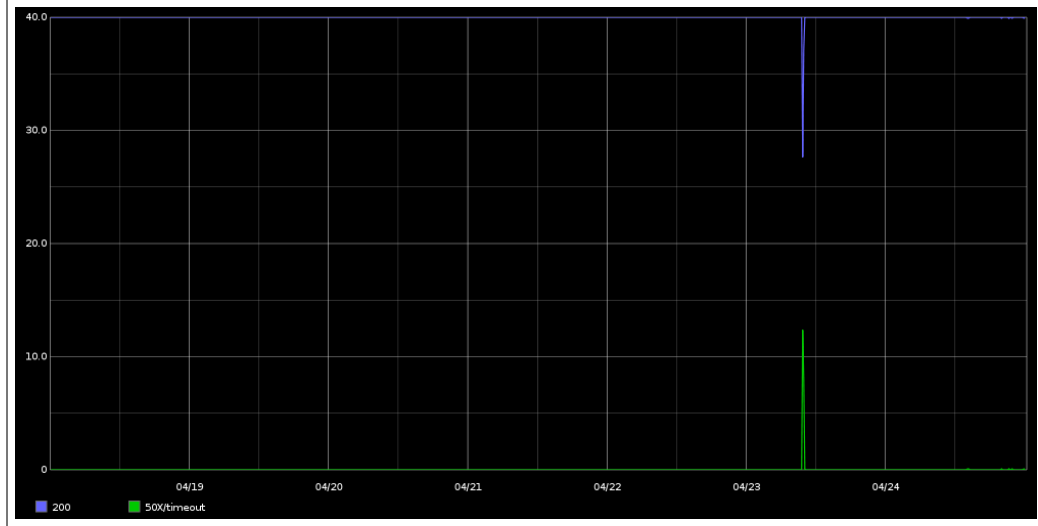


Figure 6.5: Heartbeat measurement: Summarized HTTP request status codes for each measurement step for application *threaded-comments*

not have knowledge about the state of the instance for the rest of the time. Even though this problem could be mitigated by decreasing the measurement interval (for example to one second), it remains a fundamental problem of the method.

For each application we defined one HTTP resource location to query against. Therefore *Measurer* may also only know about the availability of that single resource. Given that an application might show different failure characteristics for different resource locations, the results from our measurement may not capture the complete availability of the instances, as they are experienced by the clients of said instances. Querying more resource locations per instance might mitigate this problem, but its practical applicability is bound to the number of resource locations to query.

During operation the measurements themselves do create load on the instances, thus measuring might alter the results of the measurement. Given that we assume to work with systems that should be able to handle hundreds of requests per second per instance, the effect of measurements should not have a significant impact.

In our investigations we do not guard against network failure. Since we did not experience simultaneous failures of all instances at the same time, we may assume that no total network outage happened during the time of our tests. Still, partial network failures might have altered the results of our measurements.

One of the implicit assumptions we have with this measurement is that our system for

executing the measurements experiences no or less failures than the system we measure. This includes *Measurer* as well as *Graphite* (which we used for storing the measurement information). We did not test, as to how much this assumption is true.

Once we had the results from the measurements, we had to decide for each time period if the deploy was *up* or *down*. We based this on a ratio of *up* and *down* instances: if more than 5% of the instances were *down* we assumed the deploy to be *down*. We set this ratio threshold as a “good guess” after manually assessing the collected data. Future work should investigate better ways of assessing the ratio threshold or investigate other ways of making the *up/down* assessment for deploys.

6.2.3 Production traffic measurements

To collect historical availability data of an application deploy, we measured the production traffic for the deploy's service instances. In this section we will explain the method, how we executed it in our case study and discuss our findings.

Measurement method

When an application is deployed in the production environment it will serve the production traffic created by the users. Collecting information on that traffic allows us to derive historical availability data.

In our case study we were interested in measuring the three applications *soundcloud*, *authenticator* and *threaded-comments*. All of these have deploys in the production environment. They all use the HTTP protocol [8] to handle requests. For these requests we were able to gather metrics about the returned HTTP status codes from each deploy's service instances. To determine if an application is *up* or *down* for a specific time period we looked at the summed number of requests that returned an HTTP response status code smaller than or equal to 500. We then set a threshold. If the measured sum was above this threshold we considered the deploy to be *down*.

For measuring the production traffic we used the existing telemetry systems (as outlined earlier in subsection 3.2.3 *Telemetry*). Finding the correct telemetry system and metrics within these was a non-trivial task that we executed manually, since it required deep knowledge of the system and measurement architecture. We gathered data for seven days between April 18 2014 and April 24 2014 inclusively.

Next we will explain the exact measurement method for each application deploy and show the respective results:

soundcloud All traffic to service instances of *soundcloud* goes through *HAProxy* [108] load balancing servers. The access logs of *HAProxy* are parsed, aggregated, and saved to *Graphite*. The resolution of the data saved in *Graphite* is 56 minutes. That data is an average of sum per minute over all instances of *HAProxy*. The results can be seen in Figure 6.6. The *down* threshold was set to 75.

authenticator Each *authenticator* instance aggregates the HTTP response status codes all requests internally and aggregates them with *Prometheus*. The resolution is

1 minute. The results can be seen in Figure 6.7. The *down* threshold was set to 20.

threaded-comments Each *threaded-comments* instance sends the HTTP status code of each response to *statsd*. The resolution is 1 minute. The results can be seen in Figure 6.8. The *down* threshold was set to 20.

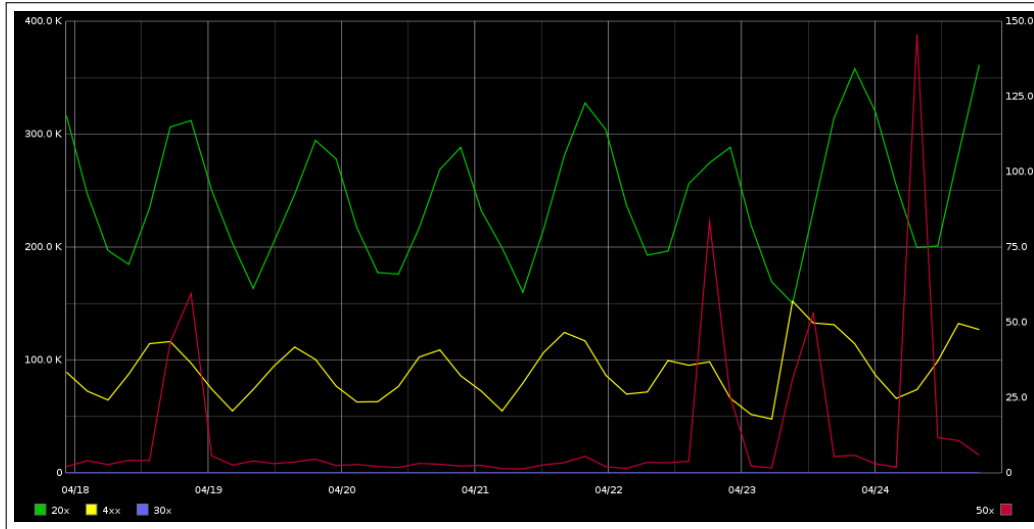


Figure 6.6: Existing measurements: Summarized HTTP response status codes on *HAProxy* for application *soundcloud*. Please note that there are two y-axes

Results

Table 6.5 shows the results of our measurements as well as the calculated failure probabilities after the equations in subsection 6.2.1 *Theory*. As mission time we chose $t = 720 \text{ minutes (12hours)}$. Using these failure probabilities in the fault tree results in a TOP event probability of **0,96585**.

Discussion In our case study we found two types of how production traffic was measured: *Threaded-comments* and *authenticator* are measured internally, within the instance processes. *Soundcloud* is measured externally, on the load balancer between service consumers and service instances. Both measure the actual service consumer traffic. We believe the external measurements to be more accurate: If a service

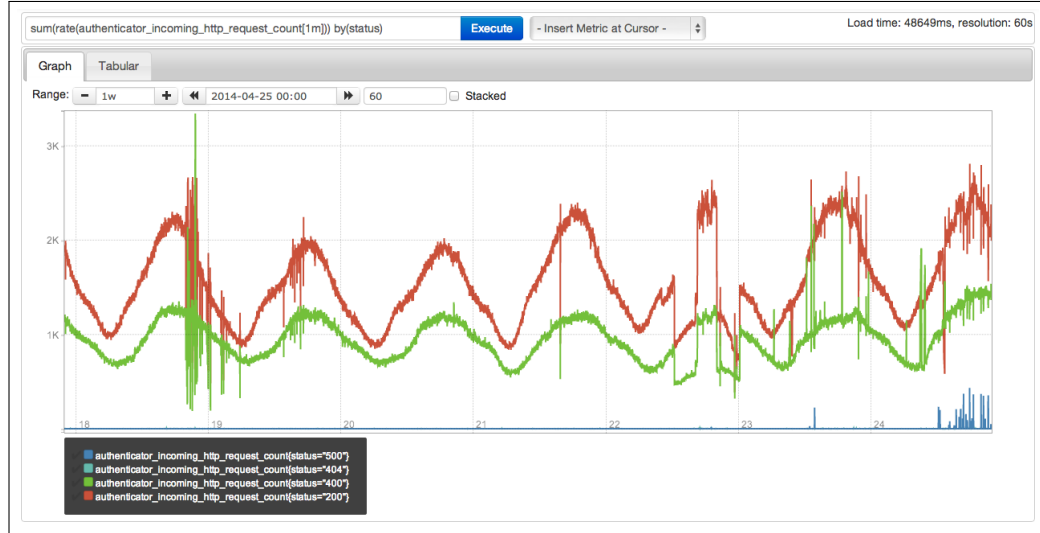


Figure 6.7: Existing measurements: Summarized HTTP response status codes for application *authenticator*

instance crashes the internal measurement will equally crash and therefore not count failed requests anymore. External measurements can count instance crashes without problems. The assumption with external measurements is then that the complete production traffic is measured on that measuring point. For example in our case study the external measurements happened on the load balancers. We verified that the service instances were not serving production traffic routed through other means. If that would be the case, the other means would have to be incorporated into the external measurements as well.

In our case study the time period resolution of the measurements varied with 56 minutes for *soundcloud* and 1 minute for *threaded-comments* and *authenticator*. This led to an exaggeration of the downtime that we measured for *soundcloud*, which we scaled up to match the resolution of the other measurements. We believe that a high resolution (preferably in the seconds not minutes range) as well as the same resolution for all measurements would improve the quality of the results.

Once we had the results from the measurements, we had to decide for each time period if the deploy was *up* or *down*. We based this on a threshold value of failed responses based on the delivered HTTP status code. We set this threshold as a “good guess” after manually assessing the collected data. Future work should investigate better ways of assessing the threshold or investigate other ways of making the *up/down*.

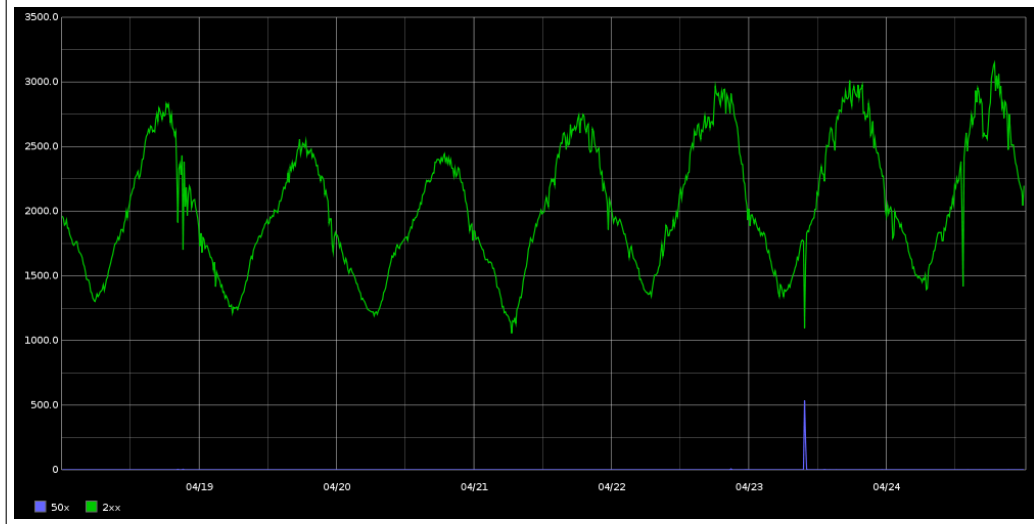


Figure 6.8: Existing measurements: Summarized HTTP response status codes for application *threaded-comments*

6.2.4 Discussion

Both approaches measure the perspective of the service consumers and how these experience the service. We believe this to be a good perspective for measuring historical availability since failures are the entity that impedes on a service’s availability and failures may only be experienced (and therefore measured) from the service consumer’s perspective.

We believe that the *production traffic* approach will deliver more accurate data. We base this opinion on two arguments: The *heartbeat* measurement may only measure a limited number of HTTP resources, whereas the *production traffic* approach will measure all relevant resources, given that these are used by the actual service consumers. Secondly the *heartbeat* approach has the problem of choosing appropriate querying intervals. In the *production traffic* approach, this problem does not exist since the querying intervals are implicitly chosen by the service consumer traffic. If we assume that a service instance serves several hundred requests per second, the effective querying interval in the *production traffic* approach is in the range of milliseconds.

For both approaches we split time into a time series with equal time periods. In both approaches we then had the problem of deciding for each time period, if a deploy as *up* or *down* for that period. In both cases we based this decision on a threshold which we set as a “good guess”. Depending on the measured data and the chosen threshold

Table 6.5: Failure probabilities for applications measured from production traffic from April 18 2014 to April 24 2014. The *mission time* for the failure probability calculation was 720 minutes (12 hours). Note that we scaled up the datapoints for *soundcloud* to make up for the low granularity provided from the measurement.

Application	Threshold	#Datapoints	#Ups	#UpSequences	MTTF	Failure probability
soundcloud	75	10080	9968	3	3322,66	0,1948
authenticator	20	10080	10037	27	358,46	0,8658
threaded-comments	20	10080	10067	2	5033,5	0,1333

value, the resulting *up/down* periods may look significantly different. We believe this to be a crucial problem of the methods. Future work should investigate this aspect more.

Next we will compare the resulting MTTFs of both methods in Table 6.4 and Table 6.5. The MTTFs of *threaded-comments* are nearly the same (*heartbeat*: 5032,50 and *production traffic* 5033,5). The MTTF of the *authenticator heartbeat* measurement (591,29) is ~65% higher than the *production traffic* measurement (358,46). The MTTF of the *soundcloud heartbeat measurement* (356,96) is ~10% of the *production traffic* measurement (3322,66). The lowest MTTFs of both methods are nearly the same (358,46 and 356,96).

TOP event probability calculation

Given that we have the historical availability time series for each of the application deploys we need to calculate the failure probability for of them.

First we calculate the MTTF and with that the failure rate. Basis for this calculation is the assumption that the system has reached steady-state. Thus given that our observation period was one week, we assume that all other weeks yield similar MTTF values than the one we observed. This is a weak assumption, since we know from experience that the availability of deploys may differ significantly between weeks. One aspect to investigate for mitigating this effect is observing over longer periods of time.

Based on the failure rate, we may calculate the failure probability. For this we have to assume a failure distribution. We chose the exponential distribution (Equation 6.5) since it allows for analytical calculation. Future work should investigate to which extent it represents failures of application deploys well and if other distributions might be

better suited. The exponential distribution equation needs a mission time for calculating the failure probability. We chose 720 minutes as a mission time as a “good guess”. Intuitively, the higher the mission time, the higher the resulting failure probabilities will be.

Given the failure probabilities for the basic events, we may calculate the TOP event probability of the fault tree. When comparing the two results (*heartbeat approach*: 0,96588 and *production traffic approach*: 0,96585) we can see that they are very similar, despite the fact that the MTTFs of the individual applications do differ significantly. We attribute the similarity to the fact that the basic event with the highest failure probability has the most significant impact on the calculated TOP event probability. Given that we have seen that the lowest MTTFs of both case study executions is nearly the same, the highest failure probabilities are equally similar, thus leading to similar TOP event probabilities.

One of the assumption when using fault tree basic event probabilities is that they are stochastically independent. When measuring historical availability, this assumption is not necessarily fulfilled, given that applications do depend on each other. We did not investigate this aspect during our measurements. We see opportunity in future work differentiating between “inherent” failure and “propagated” failure of an application.

6.3 Discussion

In this section we will discuss the findings from this chapter, compare the methods we investigated and hint to potential for future work.

In this chapter we have investigated options for creating a quantified fault tree. We based these investigations on the qualitative fault tree constructed in earlier chapter 5 *Constructing qualitative fault trees*. Given such a qualitative fault tree, we may assign failure probabilities to the basic events of the fault tree in order to calculate the TOP event probability. For the fault tree we used in our case study execution, we have shown the equation at the beginning of this chapter in Equation 6.1.

For obtaining basic event failure probabilities, we investigated four methods: two methods based on *source code metrics* (section 6.1 *Via source code metrics*) and two methods based on *historical availability* of application deploys (section 6.2 *Via historical availability data*). The focus of this work was to investigate the feasibility of the approaches in the context of our case study. We found all approaches to be executable. We did not gather enough information to do a meaningful quantitative analysis, but rather see our work as proof that it is possible to do such analyses.

Comparing the four approaches regarding the numerical values of the TOP event probability is not a fruitful task, since all methods at some point in their execution rely on a threshold or normalization chosen by the user. Still it is possible to qualitatively compare them, which we will do next.

Mapping application identifiers

For the methods based on *source code metrics* it was easy to map application identifiers to the respective codebase and source code management repository, since as assumption we earlier established in section 2.2 *Software environment terminology* that all applications have exactly one codebase with canonical repository location. For *historical availability* this process was a bigger problem. In practice, *historical availability* may only be measured from concrete service instances. The measured values may then be aggregated to represent the deploy, which in turn may be taken as the representation of the application. We were interested in the perspective of applications and dependencies between them since we took that perspective in earlier investigations chapter 4 *Constructing dependency graphs* and chapter 5 *Constructing qualitative fault trees* as well. An opportunity for future work is to use an approach based on service instances. Building the dependency graph and qualitative fault tree might then contribute from our experiences of measuring network traffic as introduced in subsection 4.2.4 *From network connections*.

External applications

Lines of code was the only method with which we calculated probabilities for external applications. We assumed *relative code churn* to not be expressive for external applications, since during our investigations no new versions of the external applications were deployed in our case study. With *historical availability* measurement, we found no representative existing *production traffic measurement* and therefore also did not investigate *external heartbeat measurements* further. Even though we did not execute these methods for external applications in our case study, technically these should be well feasible.

Source code metrics

When we executed the *lines of code* method in our case study, it revealed big differences in the size of codebases. Given that we linearly normalized the lines of code to failure

probabilities, this led to the over-emphasis of large codebases and the neglectable impact of smaller ones. Future work may investigate, how different mapping functions from *lines of code* to failure probabilities might impact the results.

The calculated *relative code churn* numbers from our case study fluctuated between 0 and $\sim 0,6$ depending on the chosen time periods. Also a property of *relative code churn* is that it cumulatively increases with increasing time periods. Fluctuation and cumulation make comparing relative code churn numbers and resulting failure probabilities hard to compare with other methods. Given the assumption that it is a good predictor of failure probabilities, we believe it may be used as an indicator for potential dependability risks. For example application developers could be notified when their service dependency applications show an increase in failure probability based on *relative code churn*.

In our work we only investigated two approaches based on *source code metrics*. Many more exist, which may be investigated in future work. The work by D'Ambros et al [109] surveying defect prediction approaches may provide a good starting point.

Historical availability

During our investigation in the case study we found both methods for measuring *historical availability* feasible. We believe that *production traffic measurements* are to be preferred over *heartbeat measurements*, since they test more resources and do not have the problem of choosing an appropriate test interval. We furthermore believe that they are well integrateable in existing measurement systems, thus lowering the complexity for implementation.

One potential problem we noticed, but did not investigate thoroughly, is the difference between “internal” and “propagated” failures of a service instance. Given that we see that a service instance is unavailable, we do not know if the failure is from within the application itself (“internal”) or cause by another application failing (“propagated”). An opportunity to investigate this in future work is to correlate failures over time, over several systems as well from the users’ perspective. We tried to do this in our case study, but during our investigation periods no problems surfaced that allowed for meaningful correlation. In our investigations we therefore assumed all noticed failures to be of “internal” nature.

Another interesting opportunity for future work is the decomposition of services into separate resources or functionalities. Each may then be measured individually for its availability. This would allow for more fine-grained fault trees and failure correlations, which would allow more detailed insights into the architecture.

Reliability for repairable systems

In the way we used fault trees here, the TOP event probability denotes the unreliability of the investigated system, without taking repairs into consideration. We believe software always to be repairable (most fundamentally by restarting it). Future work might investigate how to reflect this property in fault tree investigation.

Duplicated basic events

In the fault tree we used (Figure 5.4) there are no duplicate basic events. Our method for calculating the TOP event probability requires all basic event to be stochastically independent. This can be a problem, since there is no guarantee that in the methods we introduced. In practice, once two separate applications (*A* and *B*) depend on the same other application (*C*), then that application *C* will be represented twice in the resulting fault tree. Examples can be seen in Figure 5.5, where the basic event “5” exists many times.

Investigate change

Our investigations with *code churn* are the only ones that allow for looking at changes of the resulting TOP event probability over time. We see potential in further investigating this with the other methods, especially how changes to an application do impact the TOP event probability. Looking at changes over time would also allow for comparing the results of the methods better, since it would be possible to correlate changes.

6.4 Summary

In this chapter we investigated approaches to constructing a quantitative fault tree. Given a qualitative fault tree the TOP event probability may be calculated if basic event probabilities exist. We investigated two types of methods for gathering basic event probabilities for applications: source code metrics and historical availability. The two source code metrics *lines of code* and *relative code churn* are known to allow prediction of defects. We proposed to extend these metrics for calculating failure probabilities. Similarly, *historical availability measurements* allow the calculation basic event probabilities. We proved the feasibility of these methods by executing them in

our case study and gathered promising results, but more work is needed to quantifiably evaluate these.

7 Discussion

One of the starting points of this work was using fault trees as a way to model the dependability of microservice architectures. As a step towards constructing fault trees for a deployed microservice architecture, we investigated dependency graphs on the granularity level of applications and their service dependencies. During that process we learned that the dependency graphs themselves already are valuable tools in modeling the structure of a deployed architecture, since they allow engineers to answer the questions “Which applications does my application depend on?” and “Which applications depend on my application?”. They even provide that visibility transitively over multiple degrees of service dependencies. In comparison, the qualitative fault trees we constructed were not better suited for visually assessing an architecture, due to their significantly larger size.

However, the structure of qualitative fault trees allowed us to construct quantitative fault trees for calculating TOP event probabilities. In this work we showed that it is possible to construct them, however the resulting TOP event probabilities were only a first step towards a wide range of opportunities for assessing dependability attributes. We especially see two fields of interest for future work: Quantitative fault trees may help to quantify the impact of changes to an architecture and therefore inform architectural and engineering decisions. They may also be valuable tools for aggregating realtime measurements when operating a microservice architecture.

One of the enabling factors in our investigations was the existence of infrastructure systems, which lead to a “programmable infrastructure”. Due to their distributed nature, microservice architectures suffer from a distribution of meta information about the system. Gathering information about a deployed and running microservice architecture therefore requires dedicated infrastructure systems that collect, instrument, store and provide this meta information. These infrastructure systems are needed to allow automated investigations like the ones presented in this work.

This thesis is a first step towards researching the benefits of modeling dependability through dependencies in microservice architectures. To reduce the complexity of the work in this thesis, we excluded fault tolerance from investigations. We believe that

Discussion

including fault tolerance into the modeling approaches is a logical next step. This might aid representing reality better in the models and improve their expressiveness.

8 Summary

In this work we showed approaches for modeling dependability in microservice architectures and applied these in a case study with the “Software as a Service” company “SoundCloud”. When building a software system in the microservice architectural style, engineers face the problem of assessing its dependability attributes. In a microservice architecture many individual applications work together as one software system. These applications depend on each other, which leads to the possibility of failures propagating through the system.

In this thesis we assessed to what extend dependency graphs and fault trees could be used to model dependability of microservice architectures. As a basis we proposed our own terminology model for describing microservice architectures. Based on that model, we constructed dependency graphs, transformed them automatically to qualitative fault trees and investigated quantifying the fault trees with failure probabilities.

In our investigations, we found dependency graphs to be a valuable tool for visualizing applications and their dependencies in microservice architectures. Dependency graphs turned out to be a better visualization of service dependencies than qualitative fault trees, due to their significantly smaller size. For constructing dependency graphs from a deployed microservice architecture we investigated four methods. Our qualitative evaluation did not reveal a clear preference for a specific method, but we see three of them (*semi-automatic creation from manual annotations, from deployment configuration, from network traffic*) as viable methods.

We believe quantitative fault trees may be used for assessing the impact of architectural change on the dependability of applications in microservice architectures. In this thesis we have proven the possibility to construct them for a deployed microservice architecture. We showed four possible ways for gathering failure probabilities for applications. We found the methods of *relative code churn* and *historical availability from production traffic* to be promising.

All methods we described were executed in the context of the case study with “SoundCloud” and therefore have been tested with real data in a productively deployed

Summary

microservice architecture.

We believe that there is a tendency to utilize the microservice architectural style, since it allows for building big and complex software systems by dividing and encapsulating business concerns into individual services. Furthermore it has proven in the industry to be a viable way of constructing "Software as a Service" systems. Therefore we believe the microservice architectural style to grow in importance in the future. With that the problem of assessing dependability of such an architecture is raised. In this work we specifically addressed the problem of low visibility for dependencies between applications. We believe that this improves the discussion of dependability in microservice architectures, but many more subjects of that discussion remain open for further investigation. We look forward to the future work in industry and academia on these topics.

References

- [1] Adrian Cockcroft. “Migrating to Microservices”. In: *QCon London 2014*. 2014.
- [2] Jeremy Cloud. “Decomposing Twitter: Adventures in Service-Oriented Architecture”. In: *QCon New York 2013* (2013). URL: <http://www.infoq.com/presentations/twitter-soa>.
- [3] Todd Hoff. *Amazon Architecture* (Accessed on 07/05/2014). 2007. URL: <http://highscalability.com/blog/2007/9/18/amazon-architecture.html>.
- [4] Jean-Claude Laprie. “Dependable Computing : Concepts , Limits , Challenges”. In: *IEEE International Symposium on Fault-Tolerant Computing* 25 (1995), pp. 42–54.
- [5] Algirdas Avi, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: 1.1 (2004), pp. 11–33.
- [6] Marvin Rausand and Arnljot Hø yland. *System Reliability Theory: Models, Statistical Methods, and Applications*. 2nd Edition. Wiley-Interscience, 2003, p. 664.
- [7] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. “Fundamental Concepts of Dependability”. In: (2000), p. 6.
- [8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, June 1999. URL: <http://www.ietf.org/rfc/rfc2616.txt>.

References

- [9] John Knight. *Fundamentals of Dependable Computing for Software Engineers*. Chapman & Hall/ CRC Press, 2012, p. 433. ISBN: 9781439862551.
- [10] Flavin Cristian. "Understanding Fault-Tolerant Distributed Systems". In: *Communications of the ACM* 34.2 (1991), pp. 56 –78.
- [11] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. ISBN: 978-1441938619.
- [12] D. H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press; 2nd edition edition, 2003, p. 488. ISBN: 0873895983.
- [13] Clifton A Ericson. "Fault Tree Analysis - A History". In: *System Safety Conference, Orlando, Florida*. 1999, pp. 1–9.
- [14] W. E. Vesely and F. F. Goldberg. *Fault Tree Handbook Nureg-0492*. Washington, 1981.
- [15] Nasa Office, Mission Assurance, and Nasa Headquarters. *Fault Tree Handbook with Aerospace Applications*. Washington: NASA Office of Safety and Mission Assurance, 2002, p. 205.
- [16] *FuzzEd* (Accessed on 28/05/2014). URL: <http://fuzzed.org>.
- [17] Nikolaos Limnios. *Fault Trees*. 1st Edition. Wiley-ISTE, 2007, p. 150. ISBN: 978-1905209309.
- [18] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. 1st Edition. Prentice Hall, 2004, p. 408. ISBN: 978-0131465756.
- [19] Peter F Brown, Rebekah Metz, and Booz Allen Hamilton. "Reference Model for Service Oriented 1.0". In: *OASIS Standard* October (2006), pp. 1–31.
- [20] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000.
- [21] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996, p. 242. ISBN: 0131829572.

References

- [22] Martin Fowler and James Lewis. *Microservices* (Accessed on 25/03/2014). 2014. URL: <http://martinfowler.com/articles/microservices.html>.
- [23] Gianpaolo Carraro and Chong Fred. *Software as a Service (SaaS): An Enterprise Perspective* (Accessed on 20/05/2014). Tech. rep. 2006. URL: <http://msdn.microsoft.com/en-us/library/aa905332.aspx>.
- [24] Alain Abran, J W Moore, P Bourque, R Dupuis, and L L Tripp. "Software Engineering Body of Knowledge". In: *IEEE Computer Society, Angela Burgess* (2004).
- [25] Ross Snyder. "Continuous Deployment at Etsy: A Tale of Two Approaches". In: *SXSW The Lean Startup 2013*. 2013.
- [26] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010, p. 512. ISBN: 978-0321601919.
- [27] Craig D. Weissman and Steve Bobrowski. "The design of the force.com multitenant internet application development platform". In: *Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09* (2009), p. 889. DOI: 10.1145/1559845.1559942.
- [28] Ralph Mietzner, Tobias Unger, Robert Titze, and Frank Leymann. "Combining Different Multi-Tenancy Patterns in Service-Oriented Applications". In: (2009). DOI: 10.1109/EDOC.2009.13.
- [29] Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, and Paul Fremantle. "Multi-tenant SOA Middleware for Cloud Computing". In: *2010 IEEE 3rd International Conference on Cloud Computing* (July 2010), pp. 458–465. DOI: 10.1109/CLOUD.2010.50.
- [30] Martin Fowler. *ServiceOrientedAmbiguity* (Accessed on 07/05/14). 2005. URL: <http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html>.
- [31] Chris Partridge and Ian Bailey. "An Analysis of Services". In: *Ministry of Defence 05248454* (2010), p. 38.

References

- [32] Adam Wiggins. *The Twelve-Factor App* (Accessed on 05/01/14). 2012. URL: <http://12factor.net/>.
- [33] Scott Chacon. *Pro Git*. 1st Edition. Apress, 2009, p. 231. ISBN: 978-1430218333.
- [34] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008, p. 432. ISBN: 978-0596510336.
- [35] Abraham Silberschatz and Peter B Galvin. *Operating System Concepts*. 9th Edition. Vol. 4. Wiley, 2013, p. 944. ISBN: 978-1118129388.
- [36] *Web Services @ W3C* (Accessed on 09/05/14). URL: <http://www.w3.org/2002/ws/>.
- [37] Behrouz A Forouzan. *TCP/IP Protocol Suite*. 2nd Edition. McGraw-Hill, 2002, p. 976. ISBN: 0072460601.
- [38] Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. 1st Edition. Addison-Wesley Professional, 2011, p. 352. ISBN: 978-0321544209.
- [39] Sasu Tarkoma. *Publish / Subscribe Systems: Design and Principles*. 1st Edition. Wiley, 2012, p. 360. ISBN: 9781118354285.
- [40] Andrew D. Birrell and Bruce Jay Nelson. "Implementing remote procedure calls". In: *ACM Transactions on Computer Systems* 2.1 (Feb. 1984), pp. 39–59. ISSN: 07342071. DOI: 10.1145/2080.357392.
- [41] Merriam-Webster Online Dictionary. *Telemetry* (Accessed on 05/07/2014). URL: <http://www.merriam-webster.com/dictionary/telemetry>.
- [42] *SoundCloud - Hear the world's sounds* (Accessed on 09/05/14). URL: <https://soundcloud.com/>.
- [43] *About SoundCloud on SoundCloud - Hear the world's sounds* (Accessed on 09/05/14). URL: <https://soundcloud.com/pages/contact>.

References

- [44] Romain Dillet. *SoundCloud Now Reaches 250M Listeners In Its Quest To Become The Audio Platform Of The Web* (Accessed on 09/05/14). 2013. URL: <http://techcrunch.com/2013/10/29/soundcloud-now-reaches-250-million-listeners-in-its-quest-to-become-the-audio-platform-of-the-web/>.
- [45] Alexa. *soundcloud.com Site Overview* (Accessed on 09/05/14). 2014. URL: <http://www.alexametric.com/siteinfo/soundcloud.com>.
- [46] Alexander Grosse. "How SoundCloud scales". In: *Berlin Expert Days* (2013).
- [47] *SoundCloud Developers* (Accessed on 30/05/14). URL: <https://developers.soundcloud.com/>.
- [48] David Thomas and David Heinemeier Hansson. *Agile Web Development With Rails*. 4th Edition. Pragmatic Bookshelf, 2011, p. 451. ISBN: 9781934356548.
- [49] Phil Calçado. "From a monolithic Ruby on Rails app to the JVM". In: *JDC2013 Cairo*. 2013.
- [50] Sean Treadway. *Backstage Blog - Evolution of SoundCloud's Architecture - SoundCloud Developers* (Accessed on 09/05/14). 2012. URL: <https://developers.soundcloud.com/blog/evolution-of-soundclouds-architecture>.
- [51] Henrik Kniberg. *Scrum and XP from the Trenches*. C4Media Inc, 2007. ISBN: 978-1430322641.
- [52] Melvin E. Conway. "How Do Committees Invent?" In: *Datamation magazine* (1968).
- [53] *GitHub* (Accessed on 09/05/2014). URL: <https://github.com/>.
- [54] Kyle Neath. *Introducing Organizations* (Accessed on 14/05/2014). 2010. URL: <https://github.com/blog/674-introducing-organizations>.
- [55] Alexander Simmerl and Matt Proud. "Bazooka - Continuous Deployment at SoundCloud". In: *goto Conference Zürich*. 2013.

References

- [56] Anubhav Hanjura. *Heroku Cloud Application Development*. 1st Edition. O'Reilly, 2014. ISBN: 9781783550975.
- [57] Eugene Ciurana. *Developing with Google App Engine*. 1st Edition. Apress, 2011, p. 164. ISBN: 9781430218319.
- [58] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. "Mesos : A Platform for Fine-Grained Resource Sharing in the Data Center". In: *NSDI 2011* (2011).
- [59] Flynn - The product that ops provides to developers (Accessed on 27/05/14). URL: <https://flynn.io/>.
- [60] OpDemand. *Deis | Your Paas. Your Rules*. (Accessed on 22/06/14). 2014. URL: <http://deis.io/>.
- [61] Doozer (Accessed on 20/06/2014). URL: <https://github.com/ha/doozerd>.
- [62] Richard M. Stallman and Roland McGrath. *GNU Make: A Program for Directed Compilation*. Free Software Foundation, 2002, p. 196.
- [63] Navin Sabharwal and Manak Wadhwa. *Automation through Chef Opscode: A Hand-on Approach to Infrastructure Automation, Devops Automation, and Reporting through Chef*. 1st Edition. Apress, 2014, p. 300. ISBN: 978-1430262954.
- [64] Opscode. *opscode/chef - Github* (Accessed on 25/05/2014). URL: <https://github.com/opscode/chef/blob/master/README.md>.
- [65] Jason Wilder. *Open-Source Service Discovery - Jason Wilder's Blog* (Accessed on 05/06/14). 2004. URL: <http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/>.
- [66] J. Postel. *Domain Name System Structure and Delegation*. RFC 1591 (Informational). Internet Engineering Task Force, Mar. 1994. URL: <http://www.ietf.org/rfc/rfc1591.txt>.
- [67] S. Cheshire and M. Krochmal. *DNS-Based Service Discovery*. RFC 6763 (Proposed Standard). Internet Engineering Task Force, Feb. 2013. URL: <http://www.ietf.org/rfc/rfc6763.txt>.

References

- [68] Cricket Liu and Paul Albitz. *DNS and BIND*. 5th Edition. O'Reilly Media, 2006, p. 642. ISBN: 978-0596100575.
- [69] Chris Davis. *Graphite* (Accessed on 13/05/2014). URL: <http://graphite.readthedocs.org>.
- [70] *StatsD* (Accessed on 07/05/2014). URL: <https://github.com/etsy/statsd>.
- [71] *Prometheus* (Accessed on 07/05/2014). URL: <http://github.com/prometheus/prometheus>.
- [72] Thomas Reps and Susan Horwitz. "The Use of Program Dependence Graphs in Software Engineering". In: *ICSE '92 Proceedings of the 14th international conference on Software engineering* (1992), pp. 392–411.
- [73] Gary Chartrand. *Introductory Graph Theory*. Dover Publications, 1984, p. 320. ISBN: 978-0486247755.
- [74] Ecma International. *The JSON Data Interchange Format - Standard ECMA-404*. Geneva, 2013.
- [75] *Bundler: The best way to manage a Ruby application's gems* (Accessed on 15/05/14). URL: <http://bundler.io/>.
- [76] *The Go Programming Language* (Accessed on 15/05/14). URL: <http://golang.org/>.
- [77] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (INTERNET STANDARD). Updated by RFC 6874. Internet Engineering Task Force, Jan. 2005. URL: <http://www.ietf.org/rfc/rfc3986.txt>.
- [78] *Graphviz | Graphviz - Graph Visualization Software* (Accessed on 14/05/14). URL: <http://www.graphviz.org/>.
- [79] Peter Bourgon. *Backstage Blog - Go at SoundCloud - SoundCloud Developers* (Accessed on 09/05/14). 2012. URL: <https://developers.soundcloud.com/blog/go-at-soundcloud>.
- [80] *Linguist* (Accessed on 15/05/2014). URL: <https://github.com/github/linguist>.

References

- [81] *What is a gem? - RubyGems Guides* (Accessed on 15/05/14). URL: <http://guides.rubygems.org/what-is-a-gem/>.
- [82] *haml | RubyGems.org | your community gem host* (Accessed on 15/05/14). URL: <https://rubygems.org/gems/haml>.
- [83] *rack | RubyGems.org | your community gem host* (Accessed on 15/05/14). URL: <https://rubygems.org/gems/rack>.
- [84] *jberkel-mysql-ruby | RubyGems.org | your community gem host* (Accessed on 15/05/14). URL: <https://rubygems.org/gems/jberkel-mysql-ruby>.
- [85] Paul DuBois. *MySQL*. 5th Edition. Addison-Wesley Professional, 2013, p. 1176. ISBN: 978-0321833877.
- [86] *amqp | RubyGems.org | your community gem host* (Accessed on 15/05/14). URL: <https://rubygems.org/gems/amqp>.
- [87] *RabbitMQ - Messaging that just works* (Accessed on 15/05/14). URL: <http://www.rabbitmq.com/>.
- [88] *memcached - a distributed memory object caching system* (Accessed on 15/05/14). URL: <http://www.memcached.org/>.
- [89] Aslan Brooke. *Use Canary Deployments to Test in Production* (Accessed on 19/05/14). 2013. URL: <http://www.infoq.com/news/2013/03/canary-release-improve-quality>.
- [90] Willy Tarreau. *HAProxy Configuration Manual - Version 1.4.25*. 2014.
- [91] *Twitter Developers* (Accessed on 31/05/14). URL: <https://dev.twitter.com/>.
- [92] James Murty. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly, 2008, p. 604. ISBN: 978-0596515812.
- [93] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming, Volume 1: The Sockets Networking API*. 3rd Edition. Addison-Wesley Professional, 2003, p. 1024. ISBN: 978-0131411555.
- [94] Paul Krzyzanowski. *Introduction to Sockets Programming* (Accessed on 21/05/14). 2014. URL: <http://www.cs.rutgers.edu/~pxk/rutgers/notes/sockets/>.

References

- [95] LXC - *Linux Containers* (Accessed on 22/05/14). URL: <https://linuxcontainers.org/>.
- [96] SUSE Documentation Team. *Virtualization with Linux Containers (LXC)*. 2014.
- [97] Paul Menage, Paul Jackson, and Christoph Lameter. *CGROUPS* (Accessed on 22/05/14). 2006. URL: <https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>.
- [98] P.V. Mockapetris. *Domain names - concepts and facilities*. RFC 1034 (INTERNET STANDARD). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. Internet Engineering Task Force, Nov. 1987. URL: <http://www.ietf.org/rfc/rfc1034.txt>.
- [99] Scott Hogg. *Network Packet Monitoring Matrix Switches* (Accessed on 23/05/14). 2010. URL: <http://www.networkworld.com/community/blog/network-packet-monitoring-matrix-switches>.
- [100] Benjamin H Sigelman, Luiz Andr, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. 2010.
- [101] Johan Oskarsson. "Zipkin - A distributed tracing framework". In: *Strange Loop* (2013).
- [102] Nancy G Leveson and Peter R Harvey. "Software Fault Tree Analysis". In: *J. Syst. Softw.* 3.2 (1983), pp. 173–181. ISSN: 0164-1212. DOI: 10.1016/0164-1212(83)90030-4.
- [103] Steven A Lapp and Gary J Powers. "Computer-aided Synthesis of Fault-trees". In: *April* (1977), pp. 2–13.
- [104] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. 2nd Edition. Microsoft Press, 2004, p. 960. ISBN: 978-0735619678.
- [105] Jos Kraaijeveld. *Mining Git repositories and understanding Code Churn* (Accessed on 02/05/14). 2013. URL: <http://kaidence.org/posts/mining-git-repositories-and-understanding-code-churn.html>.

References

- [106] N. Nagappan and T. Ball. “Use of relative code churn measures to predict system defect density”. In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (2005), pp. 284–292. DOI: 10.1109/ICSE.2005.1553571.
- [107] Vincent Driessen. *A successful Git branching model* (Accessed on 02/05/14). 2010. URL: <http://nvie.com/posts/a-successful-git-branching-model/>.
- [108] *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer* (Accessed on 05/05/14). 2014. URL: <http://haproxy.1wt.eu/>.
- [109] Marco D’Ambros, Michele Lanza, and Romain Robbes. *Evaluating defect prediction approaches: a benchmark and an extensive comparison*. Vol. 17. 4-5. Aug. 2011, pp. 531–577. ISBN: 1066401191739. DOI: 10.1007/s10664-011-9173-9.