Protractor: Leveraging distributed tracing in service meshes for application profiling at scale.

ROBERT CAROSI
Protractor: Leveraging distributed tracing in service meshes for application profiling at scale.

ROBERT CAROSI
Abstract

Large scale Internet services are increasingly implemented as distributed systems in order to achieve fault tolerance, availability, and scalability. When requests traverse multiple services, end-to-end metrics no longer tell a clear picture. Distributed tracing emerged to break down end-to-end latency on a per service basis, but only answers where a problem occurs, not why. From user research we found that root-cause analysis of performance problems is often still done by manually correlating information from logs, stack traces, and monitoring tools. Profilers provide fine-grained information, but we found they are rarely used in production systems because of the required changes to existing applications, the substantial storage requirements they introduce, and because it is difficult to correlate profiling data with information from other sources.

The proliferation of modern low-overhead profilers opens up possibilities to do online always-on profiling in production environments. We propose Protractor as the missing link that exploits these possibilities to provide distributed profiling. It features a novel approach that leverages service meshes for application-level transparency, and uses anomaly detection to selectively store relevant profiling information. Profiling information is correlated with distributed traces to provide contextual information for root-cause analysis. Protractor has support for different profilers, and experimental work shows impact on end-to-end request latency is less than 3%. The utility of Protractor is further substantiated with a survey showing the majority of the participants would use it frequently.

Keywords: Observability, distributed tracing, service mesh, distributed profiling, microservices.
**Abstract**

Storskaliga Internettjänster implementeras allt oftare som distribuerade system för att uppnå feltolerans, tillgänglighet och skalbarhet. När en request spänner över flera tjänster ger inte längre end-to-end övervakning en tydlig bild av orsaken till felet. Distribuerad tracing utvecklades för att spåra end-to-end request latency per tjänst och för att ge en indikation vart problemet kan ligger med visar ofta inte orsaken. Genom user research fann vi att root-cause-analys av prestandaproblem ofta fortfarande görs genom att manuellt korrelera information från loggar, stack traces och övervakningsverktyg. Kod-profiling tillhandahåller detaljerad information, men vi fann att den sällan används i produktionssystem på grund av att de kräver ändringar i den befintliga koden, de stora lagringskraven som de introducerar och eftersom det är svårt att korrelera profilerings data med information från andra källor.

Acknowledgements

I would like to express my gratitude to my examiner Vladimir Vlassov for his constructive suggestions during the planning and development of this research work. My grateful thanks are also extended to my industrial supervisor Matthias Grüter for thinking along and his assistance in keeping my progress on schedule, Stefan Särne for his guidance and encouragement throughout the project and his help with interpreting results, Markus Gronlund and Erik Gahlin, and Staffan Larsen from Oracle for sharing their knowledge about the Java Flight Recorder, Zack Butcher and Douglas Reid from Google for their architectural suggestions, all of my colleagues with whom I had the pleasure of working, the survey participants for their cooperation and enthusiasm, and lastly, my family and friends for their support.

Stockholm, June 11th 2018
# Contents

1 Introduction  
  1.1 Context and motivation ........................................... 1  
  1.2 Problem statement and research question .......................... 2  
  1.3 Approach ......................................................................... 2  
  1.4 Research methodologies .................................................. 3  
  1.5 Main contributions .......................................................... 3  
  1.6 Delimitations ..................................................................... 4  
  1.7 Structure ........................................................................... 4  

2 Background, related work and state of the art  
  2.1 The evolution of cloud computing ......................................... 5  
    2.1.1 Tiered architectures ....................................................... 5  
    2.1.2 Isolation and virtualization ............................................. 6  
    2.1.3 Microservices architecture ............................................. 7  
    2.1.4 Container orchestration ............................................... 8  
    2.1.5 Cloud Native ............................................................. 10  
  2.2 State of the art analysis ...................................................... 10  
    2.2.1 Distributed tracing ....................................................... 10  
    2.2.2 Service meshes .......................................................... 15  
    2.2.3 Profiling ...................................................................... 17  
    2.2.4 Shortcomings of state of the art .................................... 20  

3 Problem analysis  
  3.1 Terminology ....................................................................... 21  
  3.2 User study ......................................................................... 22  
  3.3 Stakeholders ...................................................................... 23  
  3.4 Contributing factors ........................................................... 23  
  3.5 External investigation ........................................................ 24  
  3.6 Problem definition ............................................................. 24  

4 Design of Protractor  
  4.1 User stories ....................................................................... 26  
  4.2 Constraints ....................................................................... 26  
  4.3 Requirements .................................................................... 27  
  4.4 Design Goals ..................................................................... 28  
    4.4.1 Scalability .................................................................... 28
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A three tier architecture. From left to right, the first tier: presentation, the second tier: business logic, and the third tier: data.</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of virtual machines (left) and containers (right). Reproduced from [15]. Note that each virtual machine has its own operating system, whereas the operating is shared between containers.</td>
</tr>
<tr>
<td>2.3</td>
<td>Kubernetes architecture showing master node (left) and worker (right) nodes. Based on a figure from the Kubernetes documentation [55].</td>
</tr>
<tr>
<td>2.4</td>
<td>An example Deployment specification in Kubernetes. The deployment describes two replicas of an nginx Pod that exposes port 80.</td>
</tr>
<tr>
<td>2.5</td>
<td>A call graph visualizing multiple traces. Nodes are services, edges are requests. Edge weights represent the number of requests.</td>
</tr>
<tr>
<td>2.6</td>
<td>A Gantt chart visualizing the spans that make up an imaginary request trace.</td>
</tr>
<tr>
<td>2.7</td>
<td>A depiction of different sampling strategies. From left to right head-based sampling, tail-based sampling, and unitary sampling. Requests travel from top to bottom. Boxes represent services. The color of the box shows whether the span at that service is sampled, cached, or discarded. Reproduced from [79].</td>
</tr>
<tr>
<td>2.8</td>
<td>Different places to implement reliable service-to service communication. From left to right: application logic, fat client libraries, sidecar proxy. Adapted from [14].</td>
</tr>
<tr>
<td>2.9</td>
<td>A service mesh consisting of three services with sidecar proxies (data plane) and a controller (control plane). Note how proxies communicate on behalf of application to form a mesh. The mesh is configured by the controller.</td>
</tr>
<tr>
<td>4.1</td>
<td>A visual representation of correlating spans with profiling data. Operators can `drill-down’ on spans (left) to reveal profiling data about the underlying service (right).</td>
</tr>
</tbody>
</table>
5.1   1) An external caller sends a request to Application A. It is intercepted by Proxy A which will start Timer A. 2) Proxy A forwards the request to Application A. 3) Application A makes a request to Application B which is intercepted by Proxy A. 4) Proxy A connects to Application B. The request is intercepted by Proxy B. 5) Proxy B forwards the request to Application B. 6) Application B returns a response to Application A, which is intercepted by Proxy B. 7) Proxy B forwards the response to Application A. The response is intercepted by Proxy A. Proxy A stops Timer B. 8) Proxy A forwards the response to Application A. 9) Application A processes the response from Application B and sends its own response back to the caller. The response is intercepted by Proxy A. 10) Proxy A stops Timer A, and forwards the response to the caller. 11, 12) The metadata, including the timers (request durations) is asynchronously sent to mixer.

5.2 The profiler component and its interactions with the Java application and GCS. Making a GET request to the /dump endpoint will cause the profiler sidecar to connect to the JFR, and store the most recent profiling data in Google Cloud Storage.

5.3 Anomaly detection in Protractor. For illustration purposes we chose a window size of 5, and a threshold of 3. The boxes represent spans and the number is the processing time. Spans are processed from left to right. The adapter keeps a sliding window of request times delimited by dashed lines. At the top a span with request latency 43 is about to be processed. The mixer checks if the request time (43) exceeds the median of the window (12) multiplied by the threshold (3). This is the case and the span is classified as anomalous (red color) as it enters the window and pushes out the oldest entry (62).

5.4 Request flow for span creation, anomaly detection, and span augmentation. For each request the proxy generates a set of request attributes which are set to the mixer. Based on a predefined template the mixer bundles the attributes to form an instance. The instance is used as input to our custom adapter that creates a span, and performs anomaly detection. If the span is anomalous the mixer will make a request to the corresponding profiler, and receive an id that identifies the profiling data. The id is added to the span, after which the span is sent to the tracing backend (in our case Jaeger).

5.5 Protractor user flow for root-cause analysis.

5.6 Jaeger User Interface. The initial screen is the trace list (top). When a trace is clicked the view changes to show a Gantt chart visualizing the spans (bottom left). When a span is clicked the span attributes are shown including the link to the profiling data in case the span is anomalous (bottom right).

5.7 Java Mission Control showing the timeline view. The colored bars represent different types of events. The most important ones are annotated - Thread ID: list view at the top, garbage collection: vertical orange lines, processing time: horizontal blue lines, waiting for lock: horizontal red line, span duration: horizontal brown line.
5.8  An overview of Protractor’s architecture. ........................................ 45

6.1  Bar chart of average normalized SUS scores per question from 23 participants. Scores are from 0-5, higher is better. ......................... 48

6.2  Partial Pod specification for a Java application. .................................. 48

6.3  Specification highlighting the profiler sidecar after it is injected by the admission controller. ................................................................. 49

6.4  Specification highlighting the profiler sidecar after it is injected by the admission controller. ................................................................. 50

6.5  Specification highlighting the JAVA_TOOL_OPTIONS environment variable after it is injected by the admission controller. ............................... 51

6.6  The service topology used for the evaluation. ..................................... 54

6.7  Request latencies and HTTP error rates for different loads (3 mixer replicas). Note the error rate at 9000 requests per minute are significantly higher with Protractor deployed. Also, the 95th percentile request latency starts to diverge around 4500 requests per minute. .................. 56

6.8  Request latencies and HTTP error rates for different loads (5 mixer replicas). Note the error rates with and without Protractor are now closer together. Also, there is less difference in 95th request latency compared to the previous experiment. .................................................. 57

6.9  Request latency and HTTP error rates for different loads (5 mixer replicas, large nodes). Note how 95th percentile request latencies no longer diverge. This means the impact of Protractor on request latency is minimal. .............................................................................. 58

6.10 Average CPU utilization of the mixer as a function of request rate. ........ 60

6.11 Average memory utilization of the mixer as a function of request rate. ..... 61

A.1  Architecture v1. .................................................................................. 76

A.2  Architecture v2. .................................................................................. 77

A.3  Architecture v3. .................................................................................. 78

A.4  Architecture v4. .................................................................................. 79
List of Tables

2.1 Tracing terminology based on OpenCensus [71] . . . . . . . . . . . . . . 12
2.2 Different approaches to profiling, and their characteristics . . . . . . . 19
3.1 Terminology definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.1 JVM flags for profiled applications . . . . . . . . . . . . . . . . . . . . . 33
5.2 Zipkin b3 headers. Adapted from [6] . . . . . . . . . . . . . . . . . . . . 37
5.3 Indication of work: Source Lines Of Code per component . . . . . . . 43
6.1 Table of average normalized SUS scores per question. Higher is better . 47
6.2 Summary of the experimental setup . . . . . . . . . . . . . . . . . . . . . 55
6.3 A comparison of 95th percentile request times with and without Protractor under increasing amounts of load . . . . . . . . . . . . . . . . . 59
8.1 Summary of the extent to which the design goals have been met . . . . 69
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Pseudocode for custom trace event creation</td>
<td>38</td>
</tr>
<tr>
<td>6.1 Perf profiler sample code. Redacted error handling and irrelevant</td>
<td>52</td>
</tr>
<tr>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>6.2 Pod specification for a profiled elf binary. Note the shared process namespace, and the added security context to enable the perf profiler to work</td>
<td>53</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context and motivation

Large scale Internet services are increasingly being implemented as distributed systems in order to achieve fault tolerance, availability, and scalability [5]. This is exemplified by a shift from tiered architectures towards microservices [7]. Microservices architectures remove complexity from the code by making it single purpose, but increase the complexity of operations. Configuration management tools, and later container orchestration systems have been successful at taming this complexity, which has enabled much larger and more dynamic infrastructures.

Modern infrastructures consist of many heterogeneous services that communicate over a possibly unreliable network. As a result it is more difficult to discover where services spend most time, where exceptions happen, and how a system is performing as a whole.

Once adequate end-to-end metrics such as request latency no longer show a clear picture when requests span multiple services. Attempts to better understand distributed systems drive efforts to develop less intrusive and more effective mechanisms to increase system observability. Distributed tracing solutions such as Dapper [82] has emerged to break down end-to-end request latency on a per service basis to ascribe delays to particular services. However, we found that root-cause analysis of performance problems remains a difficult and mostly manual process of correlating information from monitoring tools [1], and log aggregators [69].

Profilers are another rich source of information that can be used for root-cause analysis. Profilers perform dynamic program analysis to measure, among other things, the usage of certain CPU instructions, memory consumption, and the frequency or duration of function calls. This information is useful to understand where a process spends its time, and what may have caused it to slow down. However, we found profilers are rarely used in production systems because of the changes required to existing applications, the substantial storage requirements they introduce, and because it is difficult to correlate profiling data with information from other sources.

The proliferation of modern low-overhead profilers opens up possibilities to do online always-on profiling in production environments. We propose Protractor as
the missing link that exploits these possibilities to provide application-level transparent distributed profiling. Instead of profiling services in a controlled test environment, this approach continuously profiles production services. Anomaly detection is used to selectively store relevant profiling data, and this data is then correlated with distributed traces. As a result operators can ‘dig-down’ on traces to obtain previously inaccessible contextual information on a per service level that facilitates root-cause analysis.

1.2 Problem statement and research question

This dissertation was carried out at a Digital Rights Management Company. They expressed the need to analyze performance problems in highly distributed architectures. Distributed tracers help attributing latency to individual services, however do not explain why a service is slow. Profiling data that is required for engineers to do root-cause analysis is not collected because of the changes required to existing applications, the substantial storage requirements it introduces, and because it is difficult to correlate profiling data with information from other sources. In chapter 3 the problem is reinterpreted and further analyzed. The resulting research question is:

“How can we create an integrated system to gather and correlate tracing and profiling information to provide operators with the necessary context for effective root-cause analysis?”

1.3 Approach

The goal of this dissertation is to perform distributed profiling in the context of large scale microservices architectures. We provide the conceptual contributions needed to do distributed profiling, and the technical contributions of a realistic implementation of a distributed profiling system and its evaluation.

The dissertation started with two weeks of research. During this time we studied the strengths and weaknesses of several existing solutions. Research was done in areas of distributed tracing, profiling, service meshes, and container orchestration systems as described in chapter 2. In cooperation with the company and other professionals from Oracle and Google we specified system requirements. The goal was to develop a general solution to the problem such that it may be used by the company, but also by others. The 2 weeks after the research phase were used to create a walking skeleton: a minimal prototype of the end-to-end system to identify design difficulties early on. The next 10 weeks were used to iterate on the design until it satisfied all the design goals. In the last 8 weeks we evaluated the system, and documented our findings.

Modern software development methodologies are used to continuously deliver an incrementally improved product throughout the process. This makes it possible to cope with changing requirements and other unexpected circumstances. The described approach led to a successful dissertation in which we have made several contributions, scientific and otherwise. The main contributions are listed next in section 1.5.
1.4 Research methodologies

This section describes the research methodologies, and how they were applied during the evaluation. Research methodologies can be divided into two types: quantitative and qualitative. Quantitative research is concerned with collecting and analyzing factual data. The goal is to quantify various aspects of the problem and of the solution. In contrast, qualitative research is used to make observations and gather non-numerical data. The goal is to focus on the human elements. It is concerned with the opinions, characteristics, and concepts rather than counts or measures. We use both approaches to form a comprehensive evaluation of this work.

The research methodologies are applied to a number of quantitative and qualitative design goals listed in chapter 4. Our implementation described in chapter 5 is evaluated to assess whether the design goals are met. We do this by applying deductive reasoning: we formulate a rejectable hypothesis, and devise an experiment to test it.

1.5 Main contributions

In this dissertation we present Protractor, a distributed profiling system for microservices architectures. Protractor consists of 3 components. The first component is the profiler that profiles a running process. The second component is the mixer that monitors a service for anomalous behavior, and determines when profiling data should be annotated, catalogued, and stored. The final Protractor component is a trace visualizer with support for profiling information. The main contributions of this work are the following:

1. A non-intrusive distributed profiler for microservices. It is able to profile Java based applications and correlate profiling information from different services with distributed traces.

2. A trace visualizer with support for profiling information. The visualizer graphically displays profiling information. This gives operators valuable insights when used as a tool for debugging and root cause analysis. It helps to detect problems which would otherwise be hard to find.

3. An evaluation of the quantitative design goals. This includes experimental work to measure scalability and overhead. The evaluation is conducted in a realistic testbed where performance overhead is measured.

4. An evaluation of the qualitative design goals. This includes a System Usability Survey to evaluate usability, adding support for a different profiler to evaluate flexibility, and a description of changes in developer experience to evaluate application-level transparency.

The novelty of this work has been recognized by the open-source community. Parts of the research have been contributed as examples to a popular public repository.

1https://github.com/envoyproxy/envoy/pull/3513
1.6 Delimitations

Several delimitations are in place to keep the scope of the conducted research within the bounds of a dissertation. The delimitations are summarized below, each of which is accompanied by a rationale.

1. The system will include a reference implementation to profile Java applications. A plugin model is provided to add support for applications written in other languages. We focus on Java based applications because that enables the use of the company's services as a testbed, and because it is widely considered to be the most used programming language [86].

2. The reference implementation will use high request latency as a heuristic for anomaly detection. This means that failure modes that are not reflected by an increase in request-latency are not detected. More sophisticated anomaly detection is left as future work.

1.7 Structure

The structure of the dissertation is as follows: chapter 2 introduces relevant technologies and concepts and a survey of related work. Chapter 3 presents the main problem and outlines requirements for a solution. The design of Protractor is described in chapter 4, and the implementation is described in chapter 5. Evaluation of Protractor including the experimental work can be found in chapter 6. Chapter 7 discusses the implications of Protractor, and provides possibilities for future work. Finally, chapter 8 summarizes the dissertation and presents the final conclusions.
Chapter 2

Background, related work and state of the art

This chapter presents background information related to the design and development of Protractor. More specifically we start in section 2.1 with the evolution of cloud computing. Next we present three technologies relevant to the design and implementation of Protractor, respectively Tracing in section 2.2.1, Service Meshes in section 2.2.2 and Profiling in section 2.2.3.

2.1 The evolution of cloud computing

This section serves as a primer on cloud computing technologies. It presents an industrial, somewhat less scientific perspective on how architectures and paradigms shifted over the years as a result of virtualization technologies. As part of the background it introduces some of the infrastructure related concepts and terminologies that will be used throughout the rest of the dissertation.

2.1.1 Tiered architectures

Until recently most internet services followed a three tier architecture. A graphical depiction of this architecture is shown in fig. 2.1. The business logic tier encoded the domain knowledge, and was responsible for performing calculations, essentially moving data back and forth between the other two layers. The data tier was responsible for storing and fetching data from a database or file system. Finally, the presentation tier was the user-facing part of the application, commonly a web browser, which interfaces with the business logic tier to translate requests and responses to formats each can understand. Service components in this architecture typically were monolithic, meaning functionally distinguishable parts are all interwoven rather than forming separate architectural components. This architecture was popular for its simplicity to develop, deploy, and scale.
The infrastructure for running the applications was run on-premise. A staff of operators were responsible for managing machines in all aspects including software updates, backups, security, networking and so forth. This was done by manually logging in to individual machines, and executing commands. Operating infrastructure in this way was difficult and error prone. Configuration management tools such as Puppet [76] and Chef [19] improved this situation by allowing operators to configure clusters of machines centrally. This meant operators no longer needed to log in to every machine individually, and could automate repetitive tasks.

2.1.2 Isolation and virtualization

The ubiquity of internet connected mobile phones, together with the overall rapid growth of internet web apps, led to increased traffic for internet services. This increase put a strain on applications with a monolithic architecture. A common solution was to replicate multiple instances of the application behind a load balancer. The load balancer would act as a proxy, distributing load across the different instances.

Running multiple applications on the same machine was difficult because there was no way to isolate them. As a result one application could use all the resources, essentially starving all other applications on that machine. Two applications requiring different versions of the same shared library, or sharing the same port would make it impossible to run them on the same physical machine. In case one application crashed, it would risk taking down the entire machine, taking all other applications with it. For this reason every machine would run a single application, often leading to low resource utilization.

Virtual machines became popular as a remedy for many of the problems presented above. They provide a way to isolate applications by multiple virtual machines on the same physical machines. The underlying physical machine runs a hypervisor, a piece of software responsible for creating and running virtual machines. Each of the virtual machines runs its own operating system, and is allocated a part of the machine’s physical resources. Virtual machines cannot see other virtual machines running on the same physical machines. From the application’s point of view it was indiscernible from running directly on a physical machine, apart from decreased performance as a result of the virtualization layer. By isolating applications in virtual
machines, multiple applications could run on the same physical machine without interfering with each other. This solved both the problem of isolation, and by sharing resources allowed for better resource utilization.

Containers are an operating-system level virtualization technology that provided a more lightweight alternative to virtual machines. Containers simulate an isolated and closed environment running on a single host by using linux namespaces and cgroups. In contrast to virtual machines, containers share the operating system kernel as can be seen from fig. 2.2. This results in a reduced size compared to their virtual machine equivalents. Additionally, because containers share the host kernel, they can be started orders of magnitude faster than virtual machines. Container images popularized by Docker [25] became increasingly popular to package and deploy applications, accelerating the paradigm shift in software development towards a microservices architecture.

Figure 2.2: Comparison of virtual machines (left) and containers (right). Reproduced from [15]. Note that each virtual machine has its own operating system, whereas the operating is shared between containers.

2.1.3 Microservices architecture

In the following sections the terms service and microservice will be used interchangeably. We define a service as a collection of one or more instances. All instances that belong to a particular service are equivalent, and commonly placed behind a load balancer to distribute the traffic among each.

The microservices architecture is a relatively new approach to software architecture. This approach encourages to split up systems into small, independent services that communicate over the network. Each service can be individually deployed as an operating system process, or in a virtualized environment. This allows different services to evolve at varying rates, be developed by different teams, and be deployed independently from each other. The result was that organizations could scale by decoupling team deliveries and products. Another benefit of isolated services is technology heterogeneity: the flexibility to choose technology that is best suited for a particular problem, as opposed to a more general solution. A single service can be developed using a new language or framework, allowing for quickly testing out and adopting new technology.
Besides the influences on software development and architecture, microservices provide two important operational benefits. Microservices can be scaled and replicated independently due to their isolation, separation of concerns, and independence. Scaling can thus be limited to the services that need scaling, instead of scaling the system as a whole as would be the case in a monolithic architecture. Additionally, a service can run on different types of hardware, depending on its performance characteristics. The other operational benefit is resilience. To illustrate why this is the case, consider program crashes or misbehaviors are limited to a single service. Also, faulty services can be replaced, or traffic can be routed to other instances of that service.

While containers were not solely responsible for the popularity of microservices architectures, the higher cost of operations caused by microservices can be mitigated by automation, configuration management, and higher levels of infrastructure abstractions such as virtual machines, containers, and Docker.

2.1.4 Container orchestration

The ephemerality of service instances, together with the flexibility of the container format changed the way in which machines were treated. Instead of logging in to individual machines to deploy services, machines are grouped into an abstract pool of resources by resource managers. Resource managers include a scheduler that is responsible for finding suitable machines based on a scheduling policy. Examples of resource managers are Apache Mesos [46], and YARN [89]. In the rest of this section we will look at Kubernetes [55], a type of resource scheduler particularly suited for container-based applications. Kubernetes handles other aspects of managing containers, hence it is often referred to as a container orchestration system.

Figure 2.3: Kubernetes architecture showing master node (left) and worker (right) nodes. Based on a figure from the Kubernetes documentation [55].
Kubernetes is an open source container orchestration system developed by Google [55]. We will now define several terms that will be used in later sections. Figure 2.3 depicts the high level architecture. A pod is the atomic unit in Kubernetes and consists of one or more tightly coupled containers. A node is a machine. The cluster is the collection of all nodes. All nodes in the cluster run a kubelet daemon process that communicates with the Kubernetes API server, and is responsible for managing containers on that node. The API server is an interface to the Kubernetes data model. The API server is backed by etcd [29], a highly available distributed key value store. Kubernetes has a controller manager that hosts a number of controllers. Controllers communicate with the API server in order to reconcile resource objects from the current state towards the desired state. Resource object specifications are created by operators or developers and describe the desired state of a resource. An example of a Deployment resource for two instances of the nginx web server is shown in fig. 2.4. When a Deployment is submitted to the API server the Kubernetes scheduler will start the specified pods on a suitable node in the cluster.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.0.0
        ports:
        - containerPort: 80
```

Figure 2.4: An example Deployment specification in Kubernetes. The deployment describes two replicas of an nginx Pod that exposes port 80.

The main features of Kubernetes are scheduling, scaling, service discovery, and failure recovery. The latter means that when pods misbehave or stop running, Kubernetes will make sure to reschedule it elsewhere. These features solve difficult problems in distributed systems, unburdening operators. In recent years Kubernetes has been adopted by numerous large companies [63, 74, 91], and created an active open-source community with over 1500 [54] contributors. Various cloud providers offer hosted Kubernetes solutions making it easy to deploy container based services on
their infrastructure. Operators can leverage the capacity of cloud providers to scale services up and down as required by fluctuations in traffic.

2.1.5 Cloud Native

The increasing popularity of technologies such as Kubernetes and cloud providers have sparked a new Cloud Native approach for building applications. This approach leverages the advantages of the cloud computing delivery model to deliver speed at scale. The Cloud Native Computing Foundation or CNCF [23] emerged as part of the Linux Foundation [59] to “foster collaboration between developers, end users, and vendors around high-quality projects that orchestrate containers as part of a microservices architecture” [23]. Many of the technologies used in this project as described in chapter 5 are supported by CNCF giving them a degree of continuity and maturity.

With a shared view of the evolution of cloud computing, we will next do a state of the art analysis and describe three technologies in the context of cloud native applications. For each technology we will start with the problem it solves, different approaches, existing solutions, and describe the implementation used for our implementation in more detail. The technologies are seemingly disparate and unrelated, but we will see in chapter 4 how and why the technologies are used for the design of Protractor. We introduce tracing in section 2.2.1, service meshes in section 2.2.2, and lastly profiling in section 2.2.3.

2.2 State of the art analysis

2.2.1 Distributed tracing

Tracing is a technique used to make a system observable. It is complementary to application logging in that it describes program execution. A trace depicts the execution path through a system, possibly including timing information, and system properties at different points in time. Tracing a distributed system, termed distributed tracing became popular as (aggregated) logs were no longer sufficient to debug increasingly distributed and complex systems. In a microservices architecture request paths span a multitude of services. Distributed tracing is used to answer questions such as ‘Which services did a request touch?’, ‘Was this request served from cache or from disk?’, and ‘How long did this service take servicing the request?’. These questions are of crucial importance for performance debugging, detecting service failure, and to perform root-cause analysis. For the sake of brevity, from hereon the terms ‘tracing’ and ‘distributed tracing’ are used interchangeably, both referring to the distributed variant.

Approaches to tracing

Two approaches to tracing can be distinguished: black box tracing, and annotation based tracing. We will briefly describe each, and present related works using this approach.

Black box tracing treats the software system as opaque. Tracing is done by finding correlations between inputs and outputs. This approach is used by Orion [21] to
passively analyze network traffic and correlate network delays to discover service dependencies. Alguiera et. al [2] employed statistical methods and signal processing techniques to infer inter-call causality, and ascribe delays to particular causal paths. Facebook’s Mystery Machine [22] analyzes logs to generate a causal model of system behavior. It uses techniques from machine learning to create hypotheses about program behavior, and rejecting hypotheses that are contradicted by empirical observations.

Annotation based tracing annotates requests with a global trace identifier as they flow through a system. HTTP headers are commonly used to propagate these identifiers [73, 20]. Services along the path are expected to forward these headers such that requests with the same identifier can be grouped as being part of the same trace. Systems that use this approach include Pinpoint [20], X-trace [32], and Google’s Dapper [82]. All of these are conceptually similar in that they instrument a set of libraries to add and propagate global trace identifiers through the system. Magpie [8] uses event schemas written for each application to make the relationships between events explicit. This avoids the challenges of propagating a global trace identifier.

Both approaches have benefits and drawbacks. The black box approach requires no modification to services, messages, or middleware, making it a portable solution. This is important when tracing a system comprised of possibly closed-source proprietary services from different vendors. The biggest drawback is that it requires large amounts of data due to the reliance on statistical inference. The annotation based approach does not depend on any data, but requires programs to be instrumented.

We will now elaborate further on different aspects of annotation based tracing that are important to understand the rest of this work.

Standardization
As tracing is becoming an essential part of infrastructure, several standardization efforts are ongoing to allow different solutions to interoperate through a common set of interfaces and APIs. We will discuss three of such efforts, each focussing on a different aspects of tracing.

Open Tracing [70] is a set of open source, vendor neutral APIs to decouple tracing implementations from applications. Tracer implementations, although incompatible share many of the same semantics. OpenTracing defines the terminology, and the interfaces that all tracer implementations must implement. As a result application developers can switch tracer implementations by changing the configuration, without changing any code.

OpenCensus is 'a single distribution of libraries for metrics and distributed tracing with minimal overhead that allows you to export data to multiple backends [71].' It is similar to Open Tracing in that it makes an attempt to standardize the tracing API. The main differences are that 1) in addition to an interface OpenCensus includes tracer implementations for many popular languages, 2) OpenCensus uses a standard context propagation wire format [72] instead of vendor specific ones, and 3) includes a number of Exporters to support different storage backends such as Jaeger [49], and Stackdriver [39]. Based on OpenCensus in we introduce a set of terminology that will be used in the rest of this thesis.
Term | Definition
---|---
Span | A Span represents a service performing an operation. It encapsulates a name, a start- and end time, a Context, and a set of one or more Tags.
Trace | A Trace is a Directed Acyclic Graph (DAG) with Spans as nodes and requests as edges.
Tracer | The Tracer interface creates Spans and understands how to serialize and deserialize them across process boundaries.
Context | The mechanism by which identifiable information about a span such as identifiers and options are sent over Remote Procedure Call (RPC) scope boundaries.
Attribute | A key value pair to store information about the span.

Table 2.1: Tracing terminology based on OpenCensus [71].

The context as defined in Table 2.1 states that there must be a way to send identifiable information about spans across RPC scope boundaries. Because many internet services use RPC mechanisms built on top of HTTP [38, 44, 3], HTTP headers have been used to propagate context. In an effort to standardize the headers used for context propagation, W3C is developing a specification [43] describing each header and its purpose. When the specification is finalized compliant web servers are expected to forward the headers by default. This will remove the need to write application code or middleware to forward the headers, as is the case today.

Visualizing traces
Traces contain rich information that can be visualized for different purposes. We will discuss two of the most common data visualizations and their applications.

The first visualization is the call graph, graphically depicted in 2.5. The call graph is a directed acyclic graph consisting of nodes and edges. A node represents a service, and an edge between two nodes represents one service making a request to the other. Edge weights may represent arbitrary metrics e.g. average response time or the number of requests. This visualization can be used to display multiple traces at the expense of not showing forks, joins, and concurrency. Call graphs capture the dynamic topology of modern infrastructure. As services are added and removed, the call graph is updated to reflect these changes. A microservice architecture may comprise of hundreds of services, making it infeasible to track down dependencies by hand. The call graph can visualize service dependencies to find bottlenecks, or identifying a service that is never called.
Figure 2.5: A call graph visualizing multiple traces. Nodes are services, edges are requests. Edge weights represent the number of requests.

The second visualization is the Gantt chart. This visualization is commonly used to display a single trace. The chart consists of horizontal bars that represent the spans that together make up the trace. The horizontal axis represents relative time, and the vertical axis shows the different services. Figure 2.6 graphically depicts a Gantt chart of an imaginary request trace. Different services are involved in satisfying the request, represented by the horizontal bars. Forks and joins are preserved in this visualization, but they may be difficult to identify. In the figure Service C forks out into Service D, Service E, and Service F.

Figure 2.6: A Gantt chart visualizing the spans that make up an imaginary request trace.

**Sampling**

Sampling is a technique that can be used in tracing infrastructures to limit runtime and storage overhead. This is done by reducing the number of spans that are persisted to storage. Even when storage is done out-of-band (i.e., off the critical path of the distributed system) Google has shown sampling 0.01% of spans reduced throughput overhead from 1.5% to 0.06%, and latency from 16% to 0.20% [82]. When spans do not need to be stored, sampling can still be used to limit analysis-specific data structures [17].

To decide which spans to store, there are three fundamentally different options:
head-based coherent sampling, tail-based coherent sampling, and unitary sampling [79]. Coherent means that for a given trace, either all spans, or none of the spans are sampled. In the case of unitary sampling a trace may have only a subset of its spans sampled. If the causal relationship between spans must be preserved, coherent sampling options are required. Figure 2.7 graphically displays the three sampling options.

Figure 2.7: A depiction of different sampling strategies. From left to right head-based sampling, tail-based sampling, and unitary sampling. Requests travel from top to bottom. Boxes represent services. The color of the box shows whether the span at that service is sampled, cached, or discarded. Reproduced from [79].

**Head-based coherent sampling** makes a sampling decision at the start when a request first enter the system. This decision is sent along the request path and used by subsequent services to make a sampling decision. Head based coherent sampling is used by many existing tracing implementations [82, 49, 73] because it is simple and intuitive.

**Tail-based coherent sampling** makes a sampling decision at the end after a request has traversed the entire system. Delaying the sampling allows for more informed decisions taking various trace properties into account. Anomaly detection can be done based on response time, or latency and can be used to store only anomalous spans. Anomalous spans represent problems in the system and are more likely to contain information to debug the cause. The downside of tail-based coherent sampling is that spans need to be cached until the sampling decision is made. Because many requests can happen in parallel, the storage overhead is not always feasible.

**Unitary sampling** each service independently makes a sampling decision. This means that it is possible for a given trace that not all spans are sampled. For this reason it is not suitable to illustrate causality relationships between spans. Unitary sampling is useful when only the span metadata that is propagated along the request path is required by individual services to do online analyses such as attributing the consumed resources to the request.

The sampling decision is commonly made using one of two approaches. The simplest approach is to define a percentage of spans to sample. A sampling decision will in this case consist of randomly sampling the set percentage of samples passing through. This approach, known as random sampling works well in the general case, but can be problematic with uneven load distributions in the system. In particular services that have less load will generate less spans than services with more load. To
mitigate this, adaptive sampling can be used instead. Adaptive sampling will sample a fixed number of spans per time unit (e.g. 100 spans/sec) by dynamically adjusting the sampling rate.

2.2.2 Service meshes

Service meshes are a very young field. There is not much academic research about this topic hence we will use other sources such as documentation, books, blog posts, and professionals to provide the necessary context to understand how and where they fit in.

In a tiered architecture communication happens in the form of function calls between a limited number of static components. Communication logic is handled within the code of each layer. In contrast, in a microservices architecture services communicate over a network in the form of RPCs. There are many different components that are continuously created and destroyed resulting in more complex communication patterns. Consider two examples of functionality required for reliable service-to-service communication: a service discovery mechanism for services to find each other, and a retry mechanism in case a downstream service fails to respond. These functionalities must be implemented in every service, possibly in different programming languages. This requires developers to spend time on implementing communication logic, instead of focusing on core business logic. As organizations realized this inefficiency, the communication logic was abstracted away in fat client libraries such as Netflix’s Hystrix [65], Facebook’s Proxygen [30], Twitter’s Finagle [87], and Google’s Stubby [41]. The problem with fat clients is that organizations still need to invest time and resources to integrate the libraries with the rest of their environment. Also, the libraries are limited to a specific platform, severely limiting the tools, languages, and runtimes that can be used [14]. The complexity of the communication patterns, and the limitations of approaches such as fat client libraries motivated the need for a dedicated layer for service-to-service communication: the service mesh.

The service mesh packages functionality required for reliable service-to-service communication in a dedicated platform layer, completely transparent to the application [64]. Instead of all applications implementing communication logic, the dedicated software layer is responsible for reliable service-to-service communication, similar to how TCP/IP is responsible for providing a reliable byte stream between network endpoints [31]. The service mesh consists of two disparate parts: the data plane and the control plane.

Data plane

The data plane is formed by a collection of network proxies and services. Existing proxies include Linkerd [58], Envoy [28], NGINX [67], and HAproxy [67]. These network proxies intercept traffic that is sent to and from a service, with the possibility to operate on it. A common way to implement these proxies, as is done in the context of Kubernetes, is as an application sidecar. This means that the proxy is deployed in a separate container alongside the service. Resources and system namespaces such as the network are shared between the containers by e.g. placing both in the same
pod. Figure 2.8 shows where the service-to-service communication logic is located in different approaches. In case of the sidecar proxy, the application is unaware of the proxy’s existence, while operators are given a new way to introduce platform wide observability, security, and visibility [64]. When all applications forward traffic through the proxies they form a mesh. This mesh can be maintained as part of the underlying infrastructure instead of as part of the application. The maintenance and configuration of the service mesh happens centrally through the control plane.

![Figure 2.8: Different places to implement reliable service-to-service communication. From left to right: application logic, fat client libraries, sidecar proxy. Adapted from 14](image)

**Control plane**

The control plane turns a collection of stateless proxies into a distributed system. It provides a centralized mechanism to configure the proxies based on an operator supplied configuration. Istio [48], Nelson [90], SmartStack [4], and Conduit [11] are all control plane implementations. Control planes commonly provide an API enabling interoperation with different data plane implementations [88]. Control planes provide centralized functionality such as certificate management for mutual TLS, traffic management, access policies, and centralized telemetry. They are responsible for reconfiguring the data plane to reflect configuration changes. The configuration can provide routing rules (e.g. Service A can talk to Service B, but not to Service C), service based rate limiting (e.g. Service A can send at most 100 requests per second to Service B), and Service Discovery (e.g. Service A has 3 instances running on 1.2.3.4, 1.2.3.5, and 1.2.3.6). Configuration may also include a description of what metrics and telemetry data the proxies should generate for each request. This data is then sent to the control plane for possible online analysis and storage. Figure 2.9 graphically shows the relationship between the data plane, and the control plane.
Extensibility
The service mesh provides application transparent extension points for custom functionality. Data plane proxies often provide a way to extend functionality through scripts e.g. LUA [61]. Control plane functionality can be extended with plugins written in the language of the control plane implementation: i.e. Go [35] in the case of Istio, and Java or Scala for Nelson. More relevant to this thesis is the service mesh in the context of distributed tracing. The data plane provides a suitable extension point for trace creation. This change can be introduced completely transparent to the application, without requiring changes to application code. The control plane can be used as an intermediary before spans are stored, to perform online analysis on span data. In chapter 4 we will cover different approaches to distributed tracing using service meshes.

2.2.3 Profiling
Profiling is a technique to collect profiling data about program execution. This information is useful to understand the performance of a program, identify critical sections of code, and detect deadlocks and memory leaks. A profiler is a program that is responsible for collecting this information. Three main classes of profilers can be distinguished, each of which takes a different approach. The rest of this section will describe three different classes of profilers, how they work, and what existing solutions use this approach. The last part will describe distributed profiling, the main subject of this research, and illustrate how it relates to distributed tracing.

Instrumentation-based profilers
Instrumentation-based profilers inject additional instructions into programs. The instructions keep track of how many times each function is called, and how much time was spent in each function. This creates what is called a flat profile that is useful to identify which functions used most CPU cycles. A second profile, the hierarchical profile shows for each function which other functions it called and how many times.
Hierarchical profiles require the profiler to keep track of the call stack to preserve control flow information.

The additional instructions can be injected when compiling the program. This is called static instrumentation. The instructions are added to the binary either by a compiler that supports it, or by rewriting the functions in an existing binary and adding instrumentation code. Dynamic instrumentation rewrites programs as they are running. This approach requires runtime support to get notified when a method is about to be run, to add instrumentation instructions on the fly. Some static profilers such as Java Agent Profilers rewrite target code at a bytecode level which provides the possibility to instrument third-party code.

Gprof \[42\] is a Linux based tool that uses a hybrid of statistical profiling and dynamic profiling to generate a flat profile. The instrumentation overhead has been found to be as high as 260% \[33\]. This is a result of a worst case scenario where a small function is called many times, making calls to the instrumentation library significant compared to the execution time of the function. CSProf \[33\] also takes a hybrid approach, but has an order of magnitude less overhead compared to gprof. Other implementations such as Gcov \[34\] preserve control flow information to generate a call graph profile. This allows richer visualizations such as call graphs as described in section \[2.2.4\].

Statistical profilers
Statistical profilers periodically sample program state by suspending program execution using operating system interrupts. The number of samples for a given section of the program can be used to estimate the relative time spent on that section. Statistically there will be more samples of code sections that are executed often, compared to less frequent ones. The advantages of statistical profilers is that they generally can be attached to running programs, without requiring re-compilation or re-linking. However, for optimized executables compiled without a symbol table, or debugging annotations, the results will be more difficult to interpret. Statistical profilers are less intrusive compared to other profilers, reducing side effects on the cache that may interfere with the results, and reducing the slowdown incurred by the target program. A downside is that statistical profilers offer a mere statistical approximation of execution times that is prone to error proportional to the number of samples.

Statistical profilers are well-suited to perform CPU-time profiling i.e. break down in which parts of the code a program spent time. Locations can be represented at the level of instructions, line numbers, functions, or source files. One of the most popular \[66\] statistical profilers for Linux is OProfile \[57\]. Certain profilers such as PCT \[9\] use debuggers to capture additional information such as function arguments that may help explain why time was spent in a particular section. M. Chilimbi et al. \[45\] have developed SWAT, a statistical profiler with an adaptive sampling strategy to detect memory leaks. This technique "samples executions of code segments at a rate inversely proportional to their execution frequency" to increase coverage of infrequently executed code where memory leaks often manifest \[45\].

Event-based profilers
Event-based profilers require support from the underlying runtime to provide hooks for various events. An event describes something that happened inside the runtime
such as an interrupt, a page fault, a system call, entering a method, creating an object, loading a class, and entering a thread. The type of events exposed depends on the runtime. The Linux kernel can itself be seen as a runtime that exposes different events as part of the unified API for kernel instrumentation frameworks [75]. In the context of programming languages it is the runtime, the Virtual Machine, or interpreter that is responsible for executing the code. Some language runtimes expose runtime event hook APIs include C#'s Profiling API [13], Python's hotshot [77], Ruby's ruby-prof [78], and Java's JVM TI [68].

Event-based profilers can use event hooks to register custom code that is executed whenever an event occurs. Whereas statistical profiling can only measure where in the code the CPU spends time, event-based profilers can provide much more context as a result of runtime support. For example, events may contain information about lock profiles, and garbage collection. Perf [60] is one example of an event-based profiler for the Linux operating system. It uses events exposed by the kernel to collect profiling data. The Java Flight Recorder (JFR) [50] is an event-based profiler that is integrated in the Java Virtual Machine. In chapter 5 we will discuss JFR in more detail, as is required to understand the implementation of Protractor. Table 2.2 summarizes the different approaches to profiling, and their characteristics.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Examples</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| Instrumenting    | GProf [42]   | Simple          | Obtrusive
                | CSProf [33]  |                 | High overhead
                | Gcov [34]    |                 | No runtime info
                |              |                 | Requires compiler support |
| Statistical      | OProfile [57]| Low interference| Approximative                  |
                | SWAT [45]    | Low overhead    |                               |
| Event-based      | Perf [60]    | Rich data       | Requires runtime support      |
                | JFR [50]     | Integrated      |                               |

Table 2.2: Different approaches to profiling, and their characteristics.

Distributed profiling
Distributed profiling aims to extend the notion of profiling across machine borders with the purpose of understanding the performance of a distributed system. Some work has been done in the area of distributed profiling. Magpie [8] combines traces across multiple machines to attribute resource usage to requests. It then uses a machine learning model to probabilistically model request behavior. This model can be used for anomaly detection. Whodunit [18] is a transactional profiler that attributes resource usage back to a request and the program source. This provides more context, but not sufficient for debugging purposes. Distributed debuggers such as Squash [85] help developers debug distributed applications. Squash attaches to application containers and allows stepping through the code, setting breakpoints, and offers other features similar to those of traditional debuggers. Debuggers must be manually attached to running applications, which means a problem must first be
detected, and then reproduced with the debugger active. In many cases this is not possible.

2.2.4 Shortcomings of state of the art

Based on the state of the art analysis we found that distributed tracing solutions such as Dapper, Orion, Pinpoint, and X-trace can be used to visualize the execution path through a distributed system, and attribute latency to the different services. We saw service meshes such as Istio, Nelson, SmartStack, and Conduit provide a dedicated infrastructure layer to abstract common functionality from applications. Lastly, we covered different profilers such as the Java Flight Recorder, perf, OProfile, and Gcov that collect diagnostic information about running processes. All of the aforementioned technologies are well-suited to perform their task, and this work is not an attempt to improve any of them specifically. Instead we seek to integrate tracing and profiling by correlating data from both sources. We combine existing solutions in a novel way to provide contextual information that will make it much easier to find and debug problems. We use the data plane of the service mesh to abstract away the tracing logic from applications, and the control plane as an extension point to add the logic required for correlation. In the next chapter: Chapter 3 we will more explicitly explore the problem.
Chapter 3

Problem analysis

We use definitions from IEEE 830[36] to structure the next three chapters. IEEE 830 defines a problem as the difference between what is perceived to be the case and what is desired, that we want to reduce; a solution is an action that reduces this difference. This chapter serves to explore what is perceived to be the case. Chapter 4 Design defines what is desired, and chapter 5 Implementation presents a solution that reduces this difference. This chapter explores the problem space to develop a shared understanding of different aspects of the problem. A user study is done to learn about the problem from an engineer who has experienced it first-hand. From the study we will derive a set of stakeholders and contributing factors that are important to consider. This chapter ends with a problem definition, and a research question.

3.1 Terminology

Before proceeding it is useful to define important terminology that will be used throughout the rest of this thesis in table 3.1.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>A system component that is responsible for performing some functionality.</td>
</tr>
<tr>
<td>Request</td>
<td>A message sent between two services.</td>
</tr>
<tr>
<td>Profiling data</td>
<td>Runtime information about a process, collected by a profiler. Information may include method traces, lock profiles, and garbage collection.</td>
</tr>
<tr>
<td>Anomaly</td>
<td>A request with high request latency.</td>
</tr>
<tr>
<td>Failure</td>
<td>When a service behaves in unintended ways, and fails to produce a correct result, or fails to deliver a correct result within in time.</td>
</tr>
<tr>
<td>Root-cause analysis</td>
<td>The process of finding the initiating cause that produced a failure.</td>
</tr>
</tbody>
</table>

Table 3.1: Terminology definitions.
3.2 User study

The first step at developing an understanding of the problem is done through a user study. From the problem statement it is clear that the problem is related to detecting and debugging failures in distributed systems. An engineer from the company was interviewed to identify problem areas, shortcomings of the current situation, and general pain points. The interview is transcribed and redacted for publication.

How are service failures detected?
"We have a monitoring system that collects metrics from running services. We mainly look at failure rate, and latency, but we also have something called black box pinging where we perform important requests from the outside, and measure the latency as the client would see it. If metrics are above a certain threshold for the 99th percentile, an alert triggers."

Why is it important to detect service failures?
"The main reason is to have a responsive user interface that makes for a pleasant user experience. For example: when a user attempts to make a payment and it takes too long, that customer may be lost. Another reason is that when latencies are high, work piles up. As a result services consume more resources. This escalates to the point where work builds up and starts a snowball increasing the workload. Finding issues early is a way to recover early saving cost of increasing the server fleet."

Are there any problems with this approach?
"There are two major problems. The first is that we only measure end-to-end latency, and we have no way to break this down on a per service basis. Tracing would solve this problem. Secondly, we only find some of the failures, and when we find them we cannot dive in to get more context about what was going on inside the process at the time. Ideally, it would be easy to find and investigate profiling data related to the failure."

When a service failure is detected, what steps are taken for root-cause analysis?
"Depending on the severity of the failure, the alert will either send an email or cause an on-call engineer to be paged to investigate and resolve the failure. The process differs depending on the engineer, and on the type of problem but an example could look like this. We start looking at a service that has poor statistics, looking at graphs, breaking them down per machine, and looking at statistics of dependent services. We try to answer questions such as: Is it one host or multiple hosts? Is there something in common between the hosts? Is latency increasing? Are there timeouts? Have there been any changes recently? Who has deployed what, when? Then we use domain knowledge to hypothesize possible causes. We would add additional metrics inside the process such as queue sizes, we dump stack traces, and try to correlate logs from different sources. If a service is blocking on i/o, or resizing something, or waiting for a lock, that's not something we can guess. We occasionally use the perf tool [60] to measure cpu counters, and see where cpu cycles are used. That allows us to identify if a service spends too much time in this area, and we can try to figure out why. When we identified the cause, we try to fix it in the code so that it does not happen again."

What information do you use to do root-cause analysis?
"We currently use graphs to visualize metrics collected from services, perf to see where cpu
cycles are spent, VM logs for inspecting heap size changes, heap dumps to do memory analysis, allocation profiling to see where objects are allocated and identify hotspots in the code.”

How would you describe the experience of doing root-cause analysis?
“It depends on the issue. Sometimes it’s fun and it feels like doing detective work. Sometimes it’s tedious because information is difficult to find. Sometimes it’s tedious because a lot of the techniques are manual and crude. Sometimes it’s stressful because when we try something we need to wait until it propagates, and try not to break other things in the process.”

What makes root-cause analysis difficult?
“It is difficult because it can be anything, because you (often) have to look into things you are not familiar with. You typically look for anomalies, often you find “red herrings”. The anomalies typically do not have a clear cause. Sometimes it is hard to verify an hypothesis because: 1) it is hard to test it, 2) it is scary to test it, 3) it is hard to be sure of absence of the problem due to the change (especially if it is intermittent), and 4) it can be due to external changes (increased user load, link shared on twitter, campaign launched, etc).”

3.3 Stakeholders
From the user study we identify three main stakeholders that are affected by the problem. A short description of each stakeholder, and how they are affected follows.

1. **Engineers** that operate and debug services. Engineers experience slower product iterations because time is spent on resolving failures, lack of confidence because failures are not always detected, and frustration from a lack of contextual information when doing root-cause analysis.

2. **The company**. The company experiences unexpected downtime as a result of failures, which leads to increased engineering cost, and a damaged brand image.

3. **End users**. End users are not able to utilize the company’s services as a result of unexpected downtime.

3.4 Contributing factors
Three main factors make root-cause analysis difficult. These factors are important to consider when designing a solution.

1. Requests span multiple services, making it difficult to determine which service was responsible for a failure.

2. Limited visibility in microservices architectures makes it difficult to detect failures as they occur.

3. Runtime profiling data that is useful for debugging is not collected.
3.5 External investigation

To ensure the problem was not limited to the company that hosted this dissertation but satisfied a need shared by different parties we conducted thorough external investigation. The goal of this investigation was to confirm the assumptions about the problem from outside sources.

Visiting KubeCon + CloudNativeCon
As part of the problem analysis we visited KubeCon + CloudNativeCon Europe May from 2nd until May 4th in Denmark. The conference is organized by the CNCF and is the largest of its kind with over 4000 attendees. Industry and academic speakers presented the latest developments in the field of cloud infrastructure. A popular topic at the conference was observability. Many of the talks described the need for, and challenges of observability confirming the outcome of our user study. Talking to the industry leaders from companies such as Google, Microsoft, Amazon, and RedHat was helpful to validate assumptions and ideas we had about this project. Furthermore, we had the chance to meet up with Istio core maintainers and get their suggestions on how to best leverage service meshes for distributed profiling.

Meeting JFR maintainers
Another part of the problem analysis was meeting up with the core maintainers of the Java Flight Recorder from Oracle. They were curious to hear how we planned to use the JFR for distributed profiling, and helped shape our view of this technology. They provided valuable insights on how to best use it to achieve our goal, and confirmed an industry trend towards distributed profiling.

Release of Google Stackdriver Profiler
During this research Google released Stackdriver Profiler. A commercial product that is part of the Google Cloud suite. The goal of this product is to "continuously analyze the performance of CPU or memory-intensive functions executed across an application and present the consumption of the relevant function in an interactive flame graph that helps developers understand which paths consume the most resources and the different ways in which their code is actually called." Currently supported languages include Java, Go, and Javascript. The release suggests Google noticed the same need for increased observability through profiling, and confirms the problem is existent.

3.6 Problem definition

From the user study, the stakeholder analysis, the contributing factors, and the external investigation we summarize the problem in a problem definition:

Problem definition
In a microservices architecture requests span multiple services. It is difficult to detect problems, and when they are detected it is difficult to know which service was responsible. Profiling data that is required for engineers to do effective root-cause analysis often not col-
lected because it requires changing existing applications, it introduces substantial storage requirements, and because it is difficult to correlate profiling data with information from other sources. Engineers currently use a combination of manually inspecting logs and stack traces and logging into machines to perform root-cause analysis. This approach is time-consuming and frustrating.

From the problem definition we can derive a research question. The research question intends to capture all the problems and contributing factors for all stakeholders in one sentence:

**Research question**

“How can we create an integrated system to gather and correlate tracing and profiling information to provide operators with the necessary context for effective root-cause analysis?”

We define a set of sub-questions to break down the research question into smaller, more manageable parts. Answering all of the sub questions will answer the main research question.

**Sub-questions**

1. How to decide which profiling information is relevant to store?
2. How to correlate distributed traces with profiling information?
3. How to make it easy to find relevant profiling events in the profiling data?

The sub-questions serve as guides throughout the research, and are answered in chapter 8.
Chapter 4

Design of Protractor

In the previous chapter we presented the research question that underlies this thesis. We hypothesize Protractor: a system that leverages distributed tracing in service meshes for application profiling at scale, as an answer to the research question. In this chapter we describe the design of Protractor. Section 4.1 presents a number of user stories that describe desired system functionality from a user perspective. We then specify constraints in section 4.2 and system requirements in section 4.3. The last section of this chapter: section 4.4 specifies the design goals of the system, and the design choices that will contribute to reaching these goals. The structure of this chapter is loosely based on IEEE 830 [36]: Generating Software Requirements Specification with Use Cases.

4.1 User stories

The following list of user stories are based on interviews with employees at the company. The result is a list of system functionality driven by user needs. For consistency, the user stories are all in the format As a <user>, I want <action>, so that I <reason>.

1. As an engineer, I want the system to give me access to relevant profiling information, so that I can quickly locate problems.
2. As an engineer, I don’t want to change services to integrate with the system, so that I can focus on doing my work.
3. As a company, I want the system to scale well with increased load, so that I can keep using it as the company grows.
4. As a company, I want the system to have minimal performance impact on the infrastructure, so that services are not slowed down.

4.2 Constraints

There are four major constraints on the design provenient from external factors. The first two constraints are academic and are imposed by KTH. The second two con-
strains are technical, and are imposed by the company.

**The design must be ethical, and sustainable in nature.**
The work done in this thesis will make it easier for engineers to do root-cause analysis. This will benefit the company where the engineers are employed. Hence, the ethics of the work will ultimately depend on the ethics of the company where it is applied. The work contributes to sustainability by making engineers more effective, saving natural resources otherwise wasted on slow failure recovery.

**The design must be implemented within 22 weeks, starting January 22nd 2018.**
This temporal constraint will necessarily limit the scope of the design to a proof of concept. Chapter 7 will discuss future work and includes what needs to be done to make the solution suitable for production use.

**The design must support services running in Kubernetes.**
This technical constraint is reasonable if we consider that Kubernetes is the leading orchestration platform at the time of writing, according to a survey conducted in March 2017 by [23]. 77% of participants showed a preference for Kubernetes [24] over other container orchestration systems.

**The design must support services written in Java.**
We focus on Java applications because Java is widely considered to be the most used programming language [86], and to use the company’s services as a testbed.

### 4.3 Requirements

We can use the user stories together with the constraints to create a list of system requirements. The requirements are ordered by priority.

1. collect and store spans for every service.
2. perform anomaly detection based on service processing time.
3. profile and store profiling data for anomalous services.
4. correlate spans with profiling data.
5. provide a user interface to visualize traces and download profiling data for offline analysis.
6. augment profiling data with request identifiers.

Some of these requirements are satisfied by existing systems. Distributed tracing solutions capture spans for every service. Logging and monitoring data can be used for anomaly detection. Profilers exist that capture runtime profiling data. However, to the knowledge of the author, at the time of writing there does not exist a system that satisfies all the requirements in order to provide an integrated solution. Protractor’s main goal is to provide contextual, actionable profiling data to help with root-cause analysis. The difficulty lies in correlating span data with profiling information. This
would allow operators to ‘drill-down’ on span data as is shown in fig. 4.1. Where span data answers the question *if* a request was slow, profiling data sheds light on *why* a request was slow by showing what was happening inside the service at the time.

![Diagram of correlating spans with profiling data](image)

Figure 4.1: A visual representation of correlating spans with profiling data. Operators can ‘drill-down’ on spans (left) to reveal profiling data about the underlying service (right).

### 4.4 Design Goals

The *design goals* of Protractor were driven by user needs and the shortcomings of prior systems. Design goals are desirable properties of the system that will be evaluated in chapter 6: Evaluation. Each design goal is accompanied by an argument about its importance, and by a number of *design choices* that will contribute to reaching the goal. Protractor shares many of the same design goals as existing tracing systems such as Dapper [82]. The design goals and design choices are presented below in no particular order.

#### 4.4.1 Scalability

Protractor’s impact on request latency must remain constant when request rates increase.

**Design choices:**

- *Distributed storage*: Cassandra [56] is used as a database to store spans. It is a highly available distributed database providing linear scalability and proven fault-tolerance [56]. In addition Google Cloud Storage (GCS) [37] is used for object storage of profiling data.

- *Stateless components*: All of Protractor’s components excluding storage are stateless microservices. This approach makes it easy to scale up as required by traffic patterns by selectively replicating and load-balancing components under stress.
4.4.2 Low-overhead

The system must have minimal performance overhead on running services, else operators would be reluctant to use the system at all.

Design choices:

- **Out-of-band span processing**: Span collection and anomaly detection happens off the critical path. This means that in theory services with the system enabled should only incur the performance penalty resulting from the application profiler.

- **Sampling**: Instead of tracing all requests, Protractor uses sampling to only trace and store a small fraction. This reduces overhead and saves storage space. Dapper has shown sampling is effective reducing overhead - from 16.3% at 1/1 sampling rate to 0.20% at 1/1024 sampling rate [82] - while keeping traces representative.

4.4.3 Application-level transparency

Protractor should operate in a manner that is transparent to the applications. This makes it easy to deploy in an existing environment, without requiring code changes to each application.

Design choices:

- **Sidecars**: The profiler component, and the tracing component are deployed as sidecars. This is a design pattern that packages functionality in independent container that are deployed alongside the application container. The application container can communicate with the sidecar over the localhost network.

- **Sidecar injection**: Sidecars can be automatically injected before applications are deployed. As a result developers do not need to be aware of the existence of sidecars, and deploying applications is no different from their perspective.

4.4.4 Usability

Usability is a combination of easy to learn and easy to use. It is an important design goal for the adoption and success of any system. Protractor can and should be used by engineers with different levels of technical expertise. As such it is important for the system to have a shallow learning curve, and to feel intuitive.

Design choices:

- **Extend existing workflows**: Protractor integrates and extends with existing technologies that engineers may be familiar with. The user experience is designed to be intuitive and functionality is discoverable, without interfering with existing workflows.
• **Contextual information**: Span data and profiling data are augmented, correlated, and combined, to provide relevant contextual information. The ultimate goal is to make engineers more productive.

### 4.4.5 Flexibility

Protractor should be flexible to allow the replacement and customization of components.

**Design choices:**

- **Pluggable architecture** The system has a pluggable architecture that allows components to be replaced and extended. For example new profilers can be supported by implementing a simple interface, Cassandra can be replaced with ElasticSearch.

- **Open technologies** Protractor is based solely on open technologies and standards, with the exception of GFS, which can be replaced by any other object store that implements the S3 interface such as Ceph[16]. The wire format is based on the OpenTracing[70] standard, and technologies such as Kubernetes[55], Istio[58], and Jaeger[49] are open source. JFR will be open-sourced in the near future.
Chapter 5

Implementation of Protractor

5.1 Overview

The implementation of Protractor was developed through several short iterations. In each iteration the goal was to develop a working version, improving upon the previous one, until all design goals were satisfied. Because many of the technologies involved are new, it was often not clear from the (possibly missing) documentation how the technology would work, and how it should be used. For this reason the easiest way to find out was to develop many prototypes, each one leading to new insights and possible improvements for the next. For sake of completion the iterations of the architecture have been attached as Appendix A with accompanying architecture diagrams. In the rest of this chapter we will cover the final architecture of Protractor.

Protractor consists of 5 individual components. Components were designed to have high cohesion, and low coupling. This approach, borrowed from object oriented design, helps create maintainable systems. High cohesion means that parts of a component are grouped because they all contribute to a single well-defined task of the component. Low coupling means components are independent and communicate using a public interface. In the rest of this chapter we will describe each component, along with its task, and public interface. The components are interdependent and as a result their descriptions may contain forward references to other components described later. Components may be difficult to understand in isolation, but the reader is encouraged to read through until section 5.5, which will present the complete picture illustrating how all components come together.

5.2 Components of the final architecture

By incorporating the lessons learned from each iteration we were able to settle for an architecture that satisfied all the design goals. The architecture consists of five components which are described in the upcoming sections.
5.2.1 Proxy

The first component is a proxy. The proxy is part of the data plane of the service mesh. It is deployed alongside services as a sidecar, and intercepts network traffic. For each request the proxy generates metadata such as source ip, destination ip, request size, and request duration and sends it to the mixer component for span creation. The mixer is described in detail in section 5.2.3, but for now assume it is an extension of the control plane in the service mesh. In a previous architecture, see appendix A we generated spans in the proxy and sent them directly to storage. The current approach has two major advantages. Firstly, the mixer provides an out-of-band extension point for anomaly detection and trace augmentation. Doing this in the proxies is limited to using in-band LUA scripts, infeasibly slowing down the critical path, or editing the C++ codebase which would be hard to maintain. The second advantage is that by using mixer as a layer of indirection we can decouple span creation from tracing backends. This approach makes it possible to switch tracing backends by reconfiguring mixer.

Metadata generation

The proxy is based on Envoy[28], a high performance distributed proxy written in C++. Envoy has been used in production by large scale internet companies such as Lyft [62] for several years. Two reasons motivated this choice. 1) Application-level Transparency. The proxy can be deployed as a sidecar in the same Pod alongside every Java application. Firewall rules applied with IPtables [47] are used to configure the proxy to intercept all traffic sent to and from the Pod. This makes the proxy transparent to the application in line with our Application-level Transparency design goal. 2) Out-of-band request metadata generation. The request metadata is asynchronously sent to the mixer component for span creation. In theory this should not slow down the critical path, contributing to our low performance overhead design goal.

Example

We now illustrate proxy functionality and metadata generation in more detail using an example with two Pods: Pod A, and Pod B. Both pods run a Java Application, and the proxy. A request is made from an external caller to Application A (blue arrows). Application A makes a request to Application B (green arrows), and returns the response to the caller. Request metadata is generated by the proxies and sent to the mixer (red arrows). Figure 5.1 visually depicts this flow and breaks it down step by step.
Figure 5.1: 1) An external caller sends a request to Application A. It is intercepted by Proxy A which will start Timer A. 2) Proxy A forwards the request to Application A. 3) Application A makes a request to Application B which is intercepted by Proxy A. 4) Proxy A connects to Application B. The request is intercepted by Proxy B. 5) Proxy B forwards the request to Application B. 6) Application B returns a response to Application A, which is intercepted by Proxy B. 7) Proxy B forwards the response to Application A. The response is intercepted by Proxy A. Proxy A stops Timer B. 8) Proxy A forwards the response to Application A. 9) Application A processes the response from Application B and sends its own response back to the caller. The response is intercepted by Proxy A. 10) Proxy A stops Timer A, and forwards the response to the caller. 11, 12) The metadata, including the timers (request durations) is asynchronously sent to mixer.

5.2.2 Profiler

The main task of the profiler is to collect and store profiling data about a running Java service. We use the Java Flight Recorder (JFR): a Java profiler that is integrated into the Java Virtual Machine (JVM) and causes almost no performance overhead \[53\]. Additionally the JFR can be controlled by another process over the network using Java Management Extensions (JMX). Java applications must set a number of JVM flags shown in section 5.2.2 to enable this functionality.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-XX:+UnlockCommercialFeatures</td>
<td>Unlock commercial features on JVM.</td>
</tr>
<tr>
<td>-Dcom.sun.management.jmxremote.port=9000</td>
<td>Set JMX port to 9000 (Random by default).</td>
</tr>
<tr>
<td>-Dcom.sun.management.jmxremote.authenticate=false</td>
<td>Disable password authentication.</td>
</tr>
<tr>
<td>-Dcom.sun.management.jmxremote.ssl=false</td>
<td>Disable Secure Socket Layer link encryption.</td>
</tr>
</tbody>
</table>

Table 5.1: JVM flags for profiled applications.

JFR collects events about a running JVM. Each event describes something of importance happening inside the JVM. The events are stored in a circular in-memory buffer overwriting the oldest events when the buffer is full. The buffer can be persisted by dumping it to storage.
From our benchmarks, the JFR generates in the order of 10MB of data per minute under heavy load. This translates to roughly 15GB of profiling data per day for a single process. The volume of data makes storing everything infeasible. Instead, we chose to store only profiling data related to anomalous spans to enable retrospective investigations of the causes. Section 5.2.3 will explain this in more detail.

The public interface of the profiler is three HTTP endpoints. The endpoints are `/start` to start profiling, `/stop` to stop profiling, and `/dump` to write the profiling data from the in-memory buffer to a file in external object storage. We use Google Cloud Storage (GCS) and each file is assigned a unique identifier which is returned as a response. The identifier can be used by other components to retrieve the profiling data.

The profiling and the storage of profiling data happen out-of-band to not slow down the critical path. The profiler is deployed as a sidecar to make profiling application transparent. This means applications are not aware of being profiled, and don’t need to be concerned with it. A diagram of the profiler component is shown in fig. 5.2.

![Figure 5.2](image)

Figure 5.2: The profiler component and its interactions with the Java application and GCS. Making a GET request to the `/dump` endpoint will cause the profiler sidecar to connect to the JFR, and store the most recent profiling data in Google Cloud Storage.

### 5.2.3 Mixer

The mixer is the central component that uses the other components to enable distributed profiling. It extends the Istio [48] control plane with functionality to do span creation, anomaly detection, and span augmentation. The extension is done using Istio adapters. To understand how adapters work, we need to briefly discuss the design of mixer in the context of the service mesh.

The mixer is part of the control plane of the service mesh. It serves as a generic intermediation layer between services and infrastructure backends. Infrastructure backends provide functionality to build services, including telemetry reporting. If services would directly integrate with backend systems, that would create tight coupling and pin down backend specific semantics. Instead, the mixer takes responsibil-
ity for interfacing with the backend systems, providing a layer of indirection and an extension point. Different backends are supported through binary plugins written in Go [35]. These plugins are called adapters and their main function is to abstract away the backend’s implementation details from services. As a result the dataplane and the control plane are agnostic to backend implementations, providing the flexibility to change them by simply updating the mixer configuration.

The data plane consists of Envoy proxies running alongside applications. For every request the proxies generate request metadata and send it to the mixer. In istio, this metadata is called attributes. The mixer defines a set of templates that specify how to bundle multiple attributes together to produce instances. Instances form the input data for adapters to operate on. At runtime operators can define handlers that describe which adapters should operate on which instances. The last required part of configuration to enable adapters is a rule that specifies for a route which adapter and which instances to activate.

Next we will describe the three main functionalities of mixer: span creation, anomaly detection, and span augmentation.

Span creation
By default traces are created in the proxies and sent directly to the tracing backend. For our implementation we have chosen to do span creation in the mixer instead. This gives us the flexibility to write a custom adapter that does anomaly detection and span augmentation, as described below. Mixer receives attributes sent from the proxies and generates a tracespan instance. This instance contains all the fields required to create a span that conforms to the OpenCensus [71] span format. Our custom adapter will operate on this instance and create a span object.

Anomaly detection
Anomaly detection is used to limit the storage of profiling data to anomalous requests. Storing all profiling data would require large storage capacity, without providing additional benefits. Instead we only store profiling data for services when requests they are serving appear anomalous. This is used as an indicator that something is wrong and profiling data may be needed at a later point to investigate the cause.

A failure mode describes a possible way a service can fail. Going forward it is useful to define the failure modes we want to detect from anomalies. The goal for Protractor was for anomaly detection to be simple, yet detect a wide range of failures. We observe that network issues, insufficient resources, database problems, and other failures generally cause a service to become slower. Hence, high request latency can be used as a heuristic for a failing service. More sophisticated anomaly detection is out of scope and left as future work.

The anomaly detection is implemented in the adapter. The adapter keeps a sliding window of past request latencies for each service. For every processed span the adapter will add the request latency to the window, pushing out the oldest entry when it is full. A span is considered anomalous when its request latency exceeds a configurable threshold multiplied by the median of the request latencies in the window. Figure 5.3 visually depicts a span being detected as anomalous.
Figure 5.3: Anomaly detection in Protractor. For illustration purposes we chose a window size of 5, and a threshold of 3. The boxes represent spans and the number is the processing time. Spans are processed from left to right. The adapter keeps a sliding window of request times delimited by dashed lines. At the top a span with request latency 43 is about to be processed. The mixer checks if the request time (43) exceeds the median of the window (12) multiplied by the threshold (3). This is the case and the span is classified as anomalous (red color) as it enters the window and pushes out the oldest entry (62).

We found a window size of 500, and the threshold of 3 to work well for our purposes. Because anomaly detection was not the focus of this work, we did not put much effort into finding the optimal values. In general we can say that increasing the window size causes older request latencies to be taken into account for anomaly detection. On the other hand increasing the threshold makes the anomaly detection less sensitive, and will result in fewer anomalies.

Span augmentation
When an anomaly is detected the source address of the service that generated the span is looked up. Next, the public interface of the profiler running alongside that service is used to dump the most recent profiling data to remote storage. The profiler returns an identifier that can be used to download the profiling data. The span is augmented with this identifier. Additionally, a custom attribute is added to the span: `error: true`. This will help filtering out anomalous spans in the user interface as will become apparent from section 5.2.5. Lastly the span is sent to the tracing backend which is responsible for saving it to external storage. For Protractor we use Jaeger [49] as the tracing backend, and Cassandra for storage, but this can be replaced with other backends supported by OpenCensus [71]. Figure 5.4 shows communication flows and illustrates how the adapter fits in.
5.2.4 Web server

Two modifications to web servers are required to be compatible with Protractor. For this research we modified Spark [83]: a popular Java web server, but the changes are simple enough to implement in another web server.

Tracing header forwarding

The first modification is required to enable distributed tracing. The tracing infrastructure needs a way to determine what collection of spans are part of the same trace. To do this requests carry a set of request headers that all web servers on the request path are expected to propagate from inbound to outbound requests. The headers are currently those defined by the Zipkin header propagation document [6]. There is an ongoing effort to standardize the headers into a W3C specification [43] that forces compliant web servers to propagate these by default. This would remove the need to manually modify the web server. Section 5.2.4 shows the headers that are used in our implementation, and their function.

<table>
<thead>
<tr>
<th>HTTP Header</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-b3-traceid</td>
<td>Indicates the overall ID of the trace. Every span in a trace shares this ID.</td>
</tr>
<tr>
<td>x-b3-spanid</td>
<td>Indicates the position of the parent operation in the trace tree. When the span is the root of the trace tree, the parentspanid is absent.</td>
</tr>
<tr>
<td>x-b3-parentspanid</td>
<td>Indicates the position of the parent operation in the trace tree. When the span is the root of the trace tree, the parentspanid is absent.</td>
</tr>
<tr>
<td>x-b3-sampled</td>
<td>An accept sampling decision is encoded as x-b3-Sampled: 1 and a reject as x-b3-Sampled: 0.</td>
</tr>
</tbody>
</table>

Table 5.2: Zipkin b3 headers. Adapted from [6].
Correlating traces with profiling data

With the spans, and the profiling data stored, the next step is to correlate the information from both sources. However, the profiling data contains all events leading up to the anomaly. The web server may contain many different threads that concurrently process requests. To find the events that are relevant to the anomalous request, it would be useful to know the thread identifier that processed the request, and the timestamp when it was served. This information can be added by submitting a custom event to the JFR and makes it easier to narrow down profiling data and find the relevant events.

JDK 9 introduced a new Events API [52] for the JFR. This API allows programs to commit custom events containing arbitrary data to the JFR. This means these events will be recorded, and stored along with other events in the profiling data.

In listing 5.2.4 we show the pseudocode required to create the custom JFR events in the web server. Error handling is omitted to save space. For event creation we leveraged before and after request hooks provided by the Spark Java web server [84]. Any other web server can be modified to perform this functionality. A HashMap from traceID to TraceEvent is created to persist state between the two different hooks. Before processing a request (line 1) the web server will create a custom event object using the JFR API (line 4). The traceID, and the spanID headers are read from the incoming HTTP request, and added to the event as instance fields (lines 5-10) and the TraceEvent is put in the map (line 11). After the request is processed (line 14) the event is committed to JFR, and removed from the map (line 18-19). The commit method will add the thread ID, and timing information automatically. This information will enable temporal visualizations as described next in section 5.2.5.

```java
ConcurrentHashMap<String, TraceEvent> traceEvents;

before ((req, res) -> {
    TraceEvent event = new TraceEvent();
    String traceId = req.headers("x-b3-traceid");
    String spanId = req.headers("x-b3-spanid");

    event.begin();
    event.traceID = traceId;
    event.spanID = spanId;
    traceEvents.put(traceId, event);
});

after ((req, res) -> {
    String traceId = req.headers("x-b3-traceid");

    TraceEvent event = traceEvents.get(traceId);
    event.commit();
    traceEvents.remove(traceId);
});
```

Listing 5.1: Pseudocode for custom trace event creation.
5.2.5 User Interface

The last component of Protractor is the user interface. This is the public interface for operators to investigate performance problems. The user interface makes it easy to identify anomalies, and perform root-cause analysis. Different ready-made components were used and extended to provide a fluid user experience. The user flow is shown in fig. 5.5 and consists of a part related to **tracing data**, and a part related to **profiling data**, totalling 6 distinct steps. Each step is now described. Screenshots in fig. 5.6 are added to show what the different screens of the user interface look like.

![Figure 5.5: Protractor user flow for root-cause analysis.](image)

Visualizing trace data

Traces are displayed using Jaeger’s built in web interface. The interface is written using modern web technologies and it is easy to plug in custom functionality. On the next page we present fig. 5.6 showing the different screens.
Figure 5.6: Jaeger User Interface. The initial screen is the trace list (top). When a trace is clicked the view changes to show a Gantt chart visualizing the spans (bottom left). When a span is clicked the span attributes are shown including the link to the profiling data in case the span is anomalous (bottom right).

1. On the home screen (fig. 5.6:top) we can search for traces and apply filters to display only those matching particular criteria. As described mixer will add a custom error attribute to anomalous spans. The attribute can be used for filtering out only the traces that contain one or more anomalous spans, which are likely those of interest. The number of errors (anomalous spans) is shown in the list view on the right.
2. When clicking on one of the traces from the list the screen changes to visualize a Gantt chart of the different spans that make up the trace (fig. 5.6, bottom left). Each span is visualized as a horizontal bar with the length proportional to the time a service took to handle that part of the request.

3. Anomalous spans display a red exclamation mark (!) in front of the colored bar. When a span is clicked the span information such as the span duration, and any custom tags added by the mixer are displayed (fig. 5.6, bottom right). Custom tags give more context about the request and include source, destination, median (median request time of mixer’s sliding window for this service), and a link to the profiling data in case the span was anomalous. The Gantt chart is useful to determine which of the spans caused the request to be slow. It shows which services were touched by the request, and how long each service took to process the request.

Visualizing profiling data

4. Now that it is clear which spans are responsible for slowing down the request, an operator can decide to drill down deeper on a process level. This is done by downloading the profiling data originating from the corresponding service. The Jaeger user interface is extended to generate a clickable link to download the profiling data from GCS, based on the custom profiling_data attribute added to the span by the mixer. If the operator has previously logged in to any Google service, and is authorized to access the storage bucket, no authentication is required. Otherwise the operator is prompted to log in with a valid Google account with sufficient permissions to read from GCS.

5. The profiling data is downloaded to the local machine with a .jfr extension. Several tools are available to visualize JFR profiling data, but for this implementation we chose to use Java Mission Control (JMC) [51]. JMC is developed by Oracle as part of the JDK, and provides rich visualization of JFR events.

6. An extensive description on doing root-cause analysis with JMC is outside the scope of this work, but we will highlight a screen that is of particular interest. Figure 5.7 shows all JFR events leading up to the anomaly on a timeline. The custom trace event created by the web server is also shown on this screen along with other events. To find the exact moment of the anomaly we use the search function to look up the span ID of the span we’re investigating. Doing so causes JMC to pinpoint the span in context with other events such as garbage collection (GC) pauses and lock synchronization. In the example shown in fig. 5.7 we can see that the span (brown bar) was slowed down by a 50ms GC pause (orange bar), followed by 100ms of processing (blue bar), after which the thread was blocked for one second waiting for a lock (red bar). From here operators can drill down even deeper and look for the thread that was holding the lock, or analyze object allocations to determine the reason for a GC pause.
42  CHAPTER 5. IMPLEMENTATION OF PROTRACTOR

Figure 5.7: Java Mission Control showing the timeline view. The colored bars represent different types of events. The most important ones are annotated - Thread ID: list view at the top, garbage collection: vertical orange lines, processing time: horizontal blue lines, waiting for lock: horizontal red line, span duration: horizontal brown line.

5.3 Assumptions

The final architecture makes a number of assumptions that should be taken into consideration:

1. Services are deployed on a cluster running Kubernetes and Istio - required for tracing infrastructure and sidecars.
3. Java applications use the Oracle JVM with enabled commercial features - required to use JFR.
4. Java application expose JMX over the local network - required for the profiler.
5. Communication happens using HTTP/1.1 or HTTP/2.0 - required for Istio.
6. Proxies are run with –includeIPRanges=cluster ip range CIDR e.g. 10.44.0.0/16 parameter - to allow egress traffic to GFS.
7. A Cassandra cluster is available - as a storage backend for the tracing infrastructure.
8. An object store e.g. GFS is available - as a storage backend for the profiling data.

9. Istio’s mixer component can be recompiled to include the custom adapter - required for span creation, anomaly detection, and span augmentation.

### 5.4 Indication of work

As an indication of work we have included section 5.4 to show the Source Lines Of Code (SLOC) for each component. However, SLOC is a poor indicator of work done for this thesis. The difficulty comes from architecting the solution, combining new and existing technologies in novel ways that have not been done before. The bulk of time was spent researching different systems, and assessing their feasibility as part of Protractor. As previously mentioned the final design is the result of five iterations, each involving different components, and displaying different performance characteristics. The final architecture may look straightforward, but it hides the hurdles that had to be overcome for its inception. Additionally, many of the technologies such as Istio and Jaeger, and OpenCensus are very new, and contained many unstable features and unexpected behavior. Much work was put into collaboration and communicating with developers and maintainers to discuss and propose functionality and use cases, helping influence the roadmap of these projects. We also contributed some of our findings to the open-source community.

<table>
<thead>
<tr>
<th>Component</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiler</td>
<td>213</td>
</tr>
<tr>
<td>Mixer</td>
<td>513</td>
</tr>
<tr>
<td>Web server</td>
<td>313</td>
</tr>
<tr>
<td>User interface</td>
<td>11</td>
</tr>
<tr>
<td>Resource definitions</td>
<td>343</td>
</tr>
</tbody>
</table>

Table 5.3: Indication of work: Source Lines Of Code per component.

### 5.5 Putting it all together

In this chapter we have presented the implementation of Protractor. Now that all components have been described individually it is useful too zoom out once more and look at the big picture. Figure 5.8 presents an architecture diagram of a fictitious deployment to illustrate all components and how they communicate. Using this example we will illustrate the steps involved from a client making a request to the operator doing root-cause analysis.

Three services are deployed in a Kubernetes cluster: Service A, Service B, and Service C. Each service runs in a separate Pod, along with two sidecars: the proxy, and the profiler. Three other pods are deployed as part of Protractor: the jaeger user interface, the jaeger collector, and the mixer. GFS and Cassandra run outside the cluster as storage backends.

[1]https://github.com/envoyproxy/envoy/pull/3513
The client makes a request to Service A. The request is routed through ingress, to Service B, and from Service B to Service C. At each service the proxy will generate request attributes that are sent to the mixer. The mixer will create a span for the respective service, and perform anomaly detection. If the span is anomalous its error attribute is set to true, and the /dump endpoint of the corresponding profiler is called. The profiler will store the latest profiling data to GFS, and return an identifier to look up this data later. The mixer will then augment the span to contain the identifier, and send it to the collector which saves it to Cassandra.

The operator can use the web interface exposed by Jaeger to visualize traces. Slow traces are highlighted and can be inspected to find anomalous spans. The span tags contain a link to download the profiling data to a local machine, and provide operators the necessary information investigate the issue with JMC.
CHAPTER 5. IMPLEMENTATION OF PROTRACTOR 45

Figure 5.8: An overview of Protractor’s architecture.
Chapter 6

Evaluation of the design goals

In this chapter we present an evaluation of the design goals specified in chapter 4. The purpose of this chapter is to evaluate to which extent the design goals are met. The design goals fall into two categories: quantitative and qualitative. This chapter follows the same structure: respectively section 6.1 and section 6.2.

6.1 Qualitative evaluation

Qualitative research focuses particularly on the human elements of technology. This section evaluates three qualitative design goals of Protractor. Firstly, usability is evaluated through a usability survey which was conducted after a showcase of using Protractor to debug thread contention. Next, we evaluate application-level transparency by describing the changes in developer experience that are required to make use of Protractor’s features. The last evaluation is targeted at flexibility. We show the process of adding support for profiling executables in the Executable and Linkable Format, abbreviated ELF format.

6.1.1 Usability: Debugging showcase

We evaluate usability by means of a usability survey. We showcased Protractor at the company and showed how it can be used to debug thread contention. The goal of the showcase was to demonstrate Protractor’s functionality and quantify how this was perceived by the audience with regards to Usability. At the end of the showcase we conducted a usability study to quantify the extent to which the Usability design goal was met. The study consists of a standardized System Usability Scale (SUS) survey. This survey was created by John Brooke in 1986 to evaluate the usability of products and services [10]. The survey has become an industry standard with references in over 1300 scientific publications. The major reason for choosing this test is that it can be used on sample sizes as small as 5 with reliable results [80]. The survey with 10 usability questions can be found in appendix B and was administered to the 23 infrastructure engineers at the company who participated in the demonstration.

The result of survey was 66. It is important to note that the scale is from 0 to 100, but scores should not be interpreted as percentages. We could try to compare the
score to the scores for other products. As an example, [81] reports the average SUS from over 500 studies in consumer products to be 68. However, making this comparison is not very informative because some products are inherently more difficult to use than others, no matter how well they are designed. Instead we can use the score as a baseline for further iterations of the product to measure how its usability changes over time.

Table 6.1 shows the average score for each question and is visualized in fig. 6.1. All scores have been normalized such that a higher score is better. We can see that questions 3, 5, and 10 scored worse. Questions 3 and 10 are related to learnability, and suggest that the participants were not familiar with interpreting profiling data. At the same time we see the high number of participants that report they would use the system frequently. This contrast suggests that the system looks difficult to learn, but would be worth the effort. Lastly, question 5 concerns the integration of the various functions within the system. The low score suggests we could do a better job clustering and exposing features in a way that is more intuitive to the user. Overall, the showcase was received with great enthusiasm. The user survey had a 100% response rate, and we received positive feedback from participants, who mainly voiced their eagerness try out the system once it is ready for use.

In conclusion, we may state that the usability score of Protractor is close to the average of products in other areas. Because of the inherently difficult nature of the solution this is not a bad outcome. From the lowest scoring questions, we conclude there is room for improvement with regards to learnability.

<table>
<thead>
<tr>
<th>#</th>
<th>Statement</th>
<th>Average score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I think that I would like to use this system frequently.</td>
<td>3.53</td>
</tr>
<tr>
<td>2</td>
<td>I found the system unnecessarily complex.</td>
<td>3.42</td>
</tr>
<tr>
<td>3</td>
<td>I thought the system was easy to use.</td>
<td>2.99</td>
</tr>
<tr>
<td>4</td>
<td>I think that I would need the support of a specialized person to be able to use this system.</td>
<td>3.53</td>
</tr>
<tr>
<td>5</td>
<td>I found the various functions in this system were well integrated.</td>
<td>2.66</td>
</tr>
<tr>
<td>6</td>
<td>I thought there was too much inconsistency in this system.</td>
<td>4.02</td>
</tr>
<tr>
<td>7</td>
<td>I would imagine that most people would learn to use this system very quickly.</td>
<td>3.04</td>
</tr>
<tr>
<td>8</td>
<td>I found the system very cumbersome to use.</td>
<td>3.37</td>
</tr>
<tr>
<td>9</td>
<td>I would feel very confident using the system.</td>
<td>3.59</td>
</tr>
<tr>
<td>10</td>
<td>I would need to learn a lot of things before I could get going with this system.</td>
<td>2.99</td>
</tr>
</tbody>
</table>

Table 6.1: Table of average normalized SUS scores per question. Higher is better.
6.1.2 Application-level Transparency: Developer experience

The goal of this section is to evaluate the Application-level Transparency design goal. We will do this by describing the necessary changes in developer experience in order for a Java application to exploit Protractor.

The changes are described by walking through the process of deploying a sample application. At each step we will briefly discuss the changes, why they are necessary, and how they could potentially be hidden from developers in the future. The starting specification for the Pod describing the Java application is shown in fig. 6.2.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-java-application
      image: my-repository/my-java-application:1.3
...
```

Figure 6.2: Partial Pod specification for a Java application.

All of the required changes boil down to modifying the Pod specification for the application. Because this is a fairly common use case, Kubernetes introduced the concept of Admission Controllers, which as of Kubernetes v1.9 is in beta. Admission controllers intercept requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized. The MutatingAdmissionWebhook is a type of admission controller of particular interest because it can modify the object before it is persisted. We will propose a way to abstract away
changes in developer experience by using this tool to automatically make the modifications. The three modifications are presented below:

1. **Injecting the proxy sidecar**
   The first change is to inject the proxy sidecar into the Pod, which is required for Protractor’s tracing infrastructure to work. Istio conveniently ships with an admission controller that injects the proxy in all Pods before they are created. Figure 6.3 shows the result of the Pod specification after it is modified by the Istio admission controller.

   ```yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: my-pod
   spec:
     containers:
     - name: my-java-application
       image: my-repository/my-java-application:1.3
     - name: istio-proxy
       image: docker.io/istio/proxy:0.6.0
     ...
   ``

   Figure 6.3: Specification highlighting the profiler sidecar after it is injected by the admission controller.

2. **Injecting the profiler sidecar**
   The second change is to inject the profiler sidecar which will connect to the JVM running the Java application. This can be done in a similar way to how the Istio admission controller injects the proxy. What needs to be done is to write an admission controller that will check if the Pod contains a Java based container, and if so inject the profiler sidecar alongside it. Figure 6.4 shows what the Pod specification would look like after this modification.
3. Injecting the JVM flags

The last change is to add JVM parameters introduced in section 5.2.2 to the Java application to expose JMX on the localhost network. This change is necessary for the profiler sidecar to connect to the JVM, and control the Java Flight Recorder.

The JVM flags can be injected using an admission controller. The controller would check if a Pod contains a Java based container, and if so set the required JVM flags in the `JAVA_TOOL_OPTIONS` environment variable. This environment variable is picked up by the JVM and will result in the same behavior as specifying these options as command line flags to the Java executable.
apiVersion: v1
class: Pod

metadata:
  name: my-pod

spec:
  containers:
  - name: my-java-application
    image: my-repository/my-java-application:1.3
    env:
      - name: JAVA_TOOL_OPTIONS
        value: 
          -XX:+UnlockCommercialFeatures | 
          -Dcom.sun.management.jmxremote.port=9000 
          -Dcom.sun.management.jmxremote.authenticate=false 
          -Dcom.sun.management.jmxremote.ssl=false
      - name: profiler
        image: protractor/profiler:1.0
      - name: istio-proxy
        image: docker.io/istio/proxy:0.6.0

Figure 6.5: Specification highlighting the JAVA_TOOL_OPTIONS environment variable after it is injected by the admission controller.

Bottom line
While the first change is automatically applied by the Istio admission controller, the other two changes still need to be done manually by the developer. Writing the admission controllers was outside of the scope of this work and is a suitable area of future work. As a result the goal of Application-level Transparency is not completely satisfied because of the two manual changes that are required. This means that for now developers still need to be aware that Protractor exists in order to exploit it.

6.1.3 Flexibility: Profiling ELF binaries with perf

The design goal of flexibility was defined as "The system should be flexible to allow the replacement and customization of components". To evaluate this goal we will develop a profiler for ELF binaries. ELF stands for Executable and Linkable Format and is the standard format for x86 Linux executables. To do this we will use the perf tool that was introduced in chapter 2.

The idea is to create a new profiler sidecar that uses the perf tool to continuously profile an ELF binary executing in the Pod. We will first look at the code in the sidecar profiler, and then describe the changes to the Pod specification that are required for the profiler to work.

The ELF profiler needs to have the same interface and the same semantics as the JFR profiler. It consists of a web server that exposes the /dump HTTP endpoint. To Protractor the implementations are irrelevant, as long as the semantics of making a GET request to the endpoint results in the profiling data being written to a file in external storage, and returns an ID to identify the file. The file containing profiling
data can be downloaded just like the other file through the Jaeger interface, and can be analyzed locally with tools such as `perf report`. Listing 6.1 shows the profiler code, which we will now describe in more detail.

```python
#!/usr/bin/python

import subprocess
import os
import sys
import uuid
import signal
from flask import Flask

app = Flask(__name__)

profilee_pid = find_profilee_pid()

perf_cmd = ['sudo', '/usr/bin/perf', 'record', '-g', '-e', 'cycles', '--switch-output', '--overwrite', '-a']

perf_cmd = perf_cmd + ['-p', profilee_pid]

proc = subprocess.Popen(perf_cmd)

profiler_pid = proc.pid

@app.route('/profiler/dump')
def dump():
    cmd = ['sudo', 'kill', '-USR2']
    dump_cmd = cmd + [str(profiler_pid)]
    subprocess.Popen(dump_cmd)

    ident = str(uuid.uuid4())
    store_perf_data(ident)

    return ident

def find_profilee_pid():
    ...

def upload_perf_data(file_id):
    ...
```

Listing 6.1: Perf profiler sample code. Redacted error handling and irrelevant functions.

The ELF profiler is written in Python using the Flask web framework. It is for demonstration purposes only, and the code has been redacted to hide unnecessary details. When the profiler starts up it will use `perf` to attach to the profilee. Perf can be launched with the `-overwrite` flag to write profiled events to a circular memory mapped buffer. Sending the perf process a `SIGUSR2` signal will dump the buffer to a file. This behavior is very similar to how JFR operates. Calling the `dump` endpoint
will send a \texttt{USR2} signal to the \texttt{perf} process and write the resulting file to remote storage with a randomly generated UUID as the filename. This filename is returned to the caller (mixer) which will use it to annotate the corresponding span.

In order for the profiler to work as a sidecar in a Pod several changes to the Pod specification are required. The entire specification is shown in listing 6.2 and is described in more detail below.

```yaml
apiVersion: v1
custom: Pod
custom:
metadata:
  name: profiled-elf
spec:
  containers:
  - shareProcessNamespace: true
custom: containers:
    - name: my-go-web server
custom: containers:
      image: my-repository/go-web server:1.1
    - name: istio-proxy
      image: docker.io/istio/proxy:0.6.0
    - name: perf-profiler
      image: protractor/elf-profiler:0.1
securityContext:
  privileged: true
custom: capabilities:
  add:
    - SYS_PTRACE
```

Listing 6.2: Pod specification for a profiled elf binary. Note the shared process namespace, and the added security context to enable the perf profiler to work.

The \texttt{perf} tool can connect to other processes by process ID (PID). However, because the profiler sidecar runs in a separate container in the same Pod we must add \texttt{shareProcessNamespace: true} (listing 6.2 line 9) in the Pod specification. This is currently an alpha feature in Kubernetes v1.10.0, and requires starting the apiserver with the \texttt{-feature-gates=PodShareProcessNamespace=true} flag. The result is that the process namespace is shared between Pods, allowing one container to see processes in other containers.

\texttt{Perf} can be run with the \texttt{-p <pid>} flag to pass a process ID to attach to. To attach \texttt{perf} uses the \texttt{PTRACE} system call, which is not allowed in Pods by default. To allow this we need to add a security context for the profiler container, and add \texttt{privileged: true} (listing 6.2 line 16) \texttt{SYS_PTRACE} (listing 6.2 line 19).

In this section we have seen how profilers for new languages and executable formats can be added to Protractor. The result is that Protractor can now profile both JVM based applications and ELF binaries in the same Kubernetes deployment. The profiler is under 100 lines of code, demonstrating utility of the profiler interface, and the flexibility of Protractor’s design. We conclude that the \textit{flexibility} design goal is satisfied.
6.2 Quantitative evaluation

The goal of this section is to quantify how well Protractor meets the two quantitative design goals. We devise and carry out an experiment to analyze how Protractor behaves under increasing amounts load.

The second quantitative evaluation seeks to quantify the scalability aspect of Protractor. This is done by taking a larger deployment and measuring various performance metrics. We hypothesize Protractor to consume a constant number of resources per request, hence scaling is expected to be linear. This experiment will prove or disprove that.

Throughout the evaluation we will make use of a sample topology. The topology was modeled after a production system to provide a real-world example. The code inside the services is replaced with simulation code that simply forwards requests. This change simplifies deploying the test setup, and although this change will affect cache behavior, and resource utilization, we assume it will not affect the conclusions drawn from the experiments.

Service topology
The service topology is modeled after a real-world example deployment consisting of 10 services. The code running in each service is identical and starts a web server to accept HTTP traffic on a designated port. Services have been mocked to accept incoming requests and make a number of outgoing requests to other services based on an environment variable. This environment variable defines a comma separated list of service names that a particular service should call. Using environment variables moves the topology definition from the code to the configuration, and allows the same code to run on all services. To generate the topology we wrote a Python program that converts a topology defined as an adjacency list into a set of resource definitions that can be deployed to Kubernetes.

![Service topology diagram](image)

Figure 6.6: The service topology used for the evaluation.

Experimental setup
The goal was to run the experiment on infrastructure that simulates a subset of a re-
alistic production environment. To achieve this the topology described in the previous section is deployed on a Kubernetes cluster running Kubernetes v1.9.7 in Google Kubernetes Engine. The nodes are located in `europe-west4a` in Eemshaven, Netherlands. On top of Kubernetes we deployed Istio v0.6.0, and Jaeger v1.4.1. Each service is replicated to form 2 instances to simulate horizontal scaling. Trace sampling is disabled to simulate a worst-case scenario. The infrastructure description and configuration is summarized in section 6.2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud provider</td>
<td>Google</td>
</tr>
<tr>
<td>Google compute zone</td>
<td><code>europe-west4a</code></td>
</tr>
<tr>
<td>Kubernetes version</td>
<td><code>v1.9.7</code></td>
</tr>
<tr>
<td>Istio version</td>
<td><code>v0.6.0</code></td>
</tr>
<tr>
<td>Jaeger version</td>
<td><code>v1.4.1</code></td>
</tr>
<tr>
<td>Trace sampling rate</td>
<td><code>100%</code></td>
</tr>
<tr>
<td>Number of services</td>
<td><code>10</code></td>
</tr>
<tr>
<td>Replicas per service</td>
<td><code>2</code></td>
</tr>
</tbody>
</table>

Table 6.2: Summary of the experimental setup.

**Experiment description**

The goal of the experiments is to study the performance characteristics of Protractor. To do this we use a load testing tool to generate increasing amounts of load in the form of HTTP requests and send them to the outside facing service $A$ until the system starts to fail. We define failure as the performance failures failure mode. This failure mode is defined as "the server is delivering the correct values, but they arrive at the wrong time, either early or late." An experiment is carried out twice: once with Protractor deployed, and once without Protractor deployed which serves as a baseline. While the system is under stress we collect a number of metrics from the nodes in the cluster. We can then compare the metrics from the two experiments to quantify the impact of Protractor compared to the baseline.

An experiment consists of multiple separate phases of increasing load. A phase is characterized by a number of requests per minute. Each phase is run for 10 minutes to give the system time to reach a steady state, and reduce the impact of random events. Between phases the environment is reset in order to minimize interference between different runs. We now describe the conducted experiments and present the results to evaluate Protractor’s quantitative design goals.

### 6.2.1 Scalability: Latency analysis

The latency analysis serves to illustrate how request latency is impacted by Protractor. Because the system was designed to do all of its work off the critical path we hypothesize there will be minimal impact on request latency. To verify this assumption we look at request latency from the experiment described above.

To measure request latency we need to define a metric. Latency tends to be multimodal, partly due to hiccups in the services. These hiccups can be caused by GC pauses, context switches, indexing, cache buffer flushes, etc. For this reason mean
and median are misleading: they hide outliers because the majority of data is normal. Max is also misleading because it is easily distorted by a single outlier. Instead we use the 95th percentile. The 50th percentile (median) is added to the plots for reference.

**Experiment 1: initial configuration**
For the first experiment we used a cluster of 12 nodes, each node with 2 vCPUs, and 7.5 GB of memory. The mixer is replicated to form 3 replicas. Figure 6.7 presents the latency analysis for the experiment. The graph shows request latency on the left vertical axis (log scale) as a function of requests per minute on the horizontal axis. The 95th percentile is plotted as circles, and the 50th percentile is plotted as crosses. HTTP errors are added as percentages on the right vertical axis. They are displayed as vertical bars.

![Figure 6.7: Request latencies and HTTP error rates for different loads (3 mixer replicas). Note the error rate at 9000 requests per minute are significantly higher with Protractor deployed. Also, the 95th percentile request latency starts to diverge around 4500 requests per minute.](image)

We note that request latency percentiles start fairly close together, until they start to diverge at 4500 requests per minute. This means that the deployment of Protractor is slowing down requests. This is also demonstrated by the fact that the HTTP error rate at 9000 requests per minute is close to 20% with Protractor, while under 1% without Protractor.

Our hypothesis is that the slowdown is because mixer cannot keep up with check calls from the proxies. Check calls are performed on a fraction of the requests forwarded by the proxies to see if they are allowed according to access control policies. Proxies will block until the check call returns, which means that if mixer’s response time to these calls slows down, the requests will slow down too. This is not unique for Protractor, but a result of using Istio.

To see how Protractor reduces mixer’s performance we consider the following. The anomaly detection for each request is executed in its own goroutine: a lightweight
thread. The mixer’s check calls are also executed in goroutines, which means that check calls now need to contend with anomaly detection goroutines from the same thread pool. Additionally, nodes only have 2 vCPUs. Although we made an effort to handle all requests on separate threads in a concurrent way, we learn that concurrency is not equal to parallelism. In this case even though many threads may be ready to execute, only 2 can execute at any point in time. We verify this by looking at the number of created goroutines and note that these are monotonically increasing for the entire duration of a phase. This means that they are created more rapidly than they are consumed which confirms this suspicion.

Experiment 2: horizontal scaling
We attempt to resolve the bottleneck by horizontally scaling mixer to 5 instances. Figure 6.8 shows the results.

![Figure 6.8: Request latencies and HTTP error rates for different loads (5 mixer replicas). Note the error rates with and without Protractor are now closer together. Also, there is less difference in 95th request latency compared to the previous experiment.](image)

We note the error bars at 9000 requests per minute are now closer together, and request latencies only start to diverge at 6000 requests per minute, up from 4500 in the previous experiment. Additionally we can see that the difference of 95th percentile request latency between Protractor and without Protractor is less than before. This is another data point in favor of our hypothesis. It suggests that the mixer indeed was a bottleneck. However, it could still be the bottleneck in this new configuration. To verify whether this is the case we consulted CPU utilization graphs of the different pods. We noted that mixer instances were no longer bound by CPU. Instead, the CPU utilization of services $E$ and $J$ reached to 100%. These services are on the request path respectively 2 and 3 times (see fig. 6.6) thus receive more traffic than other services. The diverging of 95th percentile request times can be attributed to the increased CPU utilization of the proxy when Protractor is enabled. With Protractor the proxies send additional attributes to the mixer for each request. Because proxies are deployed in the same Pod as the services they share the same CPU. This means that
sending additional attributes uses CPU cycles that could otherwise be used to reply to requests. We suspect the bottleneck is now Envoy CPU utilization.

**Experiment 3: vertical scaling**

To validate our hypotheses we scaled the nodes vertically from 2vCPUs and 7.5 GB of memory to 8 vCPUs and 30GB of memory. The rationale is that by increasing the resources on the nodes, we alleviate the CPU bottleneck which should reduce the difference in 95th percentile request latencies. The results of experiment 2 are presented below in fig. 6.9.

![Figure 6.9: Request latency and HTTP error rates for different loads (5 mixer replicas, large nodes). Note how 95th percentile request latencies no longer diverge. This means the impact of Protractor on request latency is minimal.](image)

From the graph it looks like the performance impact of Protractor is minimal. The 95th percentile request latency with and without protractor are very close together until 45000 requests per minute. At this point there are HTTP errors which suggests the system is overloaded. From looking at the CPU utilization of the nodes we learn that the bottleneck are now the services rather than Protractor’s infrastructure. To quantify the performance impact of Protractor we can plot the values in a table and look at the difference in 95th percentile request latency. We present this table below in Table 6.3.
Requests per minute & 95th percentile without Protractor (ms) & 95th percentile with Protractor (ms) & ∆95th percentile
--- & --- & --- & ---
600 & 81.10 & 82.30 & 1.48%
1500 & 76.92 & 77.45 & 0.70%
3000 & 78.10 & 74.78 & 4.25%
4500 & 76.66 & 76.69 & 0.04%
6000 & 73.17 & 74.79 & 2.22%
7500 & 73.49 & 74.45 & 1.30%
9000 & 75.73 & 75.01 & -0.96%
10500 & 76.87 & 77.34 & 0.61%
12000 & 80.12 & 78.59 & -1.91%
18000 & 89.17 & 91.85 & 3.00%
30000 & 118.73 & 121.74 & 2.53%
45000 & 2778.47 & 3130.73 & 12.68%

Table 6.3: A comparison of 95th percentile request times with and without Protractor under increasing amounts of load.

In the rightmost column we see the difference in 95th percentile request latency for different request rates. The percentages represent request latency slowdown from deploying Protractor. We see that the values are very close to 0, and sometimes take on negative values. A negative value would mean that the system becomes faster from deploying Protractor. This is nonsensical, hence the standard deviation must be larger than the performance impact. At the highest request rate the slowdown is 12.68%. However, as we noted in the previous experiment this is caused by resource starvation of Protractor’s infrastructure voiding this data point. Across all request rates the average slowdown is 0.43%. We add to this number the slowdown incurred from the profiler, which in the case of the Java Flight Recorder is less than 2% for most applications [53]. We conclude that the total impact is less than 3% as long as Protractor is allocated sufficient resources.

A surprising result is that the request latency decreases as we increase the request rate. This may seem counterintuitive so we tried to understand why this was the case. The most likely cause of this is that the JVM will optimize code that is frequently executed. The instructions in this code are interpreted at first, then compiled Just In Time (JIT), and finally optimized. The period between interpreted code and optimized code is called warmup time. Hence, when we increase the request rate, a relatively larger portion of requests are handled by optimized code, making them faster. This is true up to a certain point, when the resource starvation in the individual service outweighs the optimization benefit.

### 6.2.2 Low Overhead: Resource utilization & network footprint

In this section we aim to quantify the overhead of Protractor in terms of resources: CPU, memory, and network. The data was gathered from experiment 3.

**CPU utilization of the mixer**

The first resource to quantify is CPU usage. Many Protractor components are based
The mixer forms an exception because we modified it to perform anomaly detection and span creation. We would like to measure the performance impact of these changes and ensure that the overhead scales linearly with the request rate. To do this we collected average mixer CPU utilization for each phase during experiment 3. The result is shown in fig. 6.10.

![Figure 6.10: Average CPU utilization of the mixer as a function of request rate.](image)

Both series appear linear at first sight. CPU utilization with Protractor is steeper, which is the result of the changes we made. More concretely, the CPU utilization without Protractor fits a first grade polynomial $ax + b$ with $a = 0.0130$, and $b = 0.2449$ with a mean squared error of 0.004. The utilization with Protractor fits a first grade polynomial with $a = 0.0179$, and $b = 0.2340$ with a mean squared error of 0.003. These results confirm that the CPU utilization scales almost perfectly linear with request rate. We compare the slopes of the 2 polynomials and learn that by employing Protractor the mixer consumes about 38% more CPU per request.

**Average memory utilization of the mixer**
To ensure our changes to mixer are not leaking memory we can look at mixer’s average memory utilization during experiment 3. We expect mixer to consume slightly more memory as a result of doing anomaly detection. The data is presented below in fig. 6.11.
CHAPTER 6. EVALUATION OF THE DESIGN GOALS 61

Figure 6.11: Average memory utilization of the mixer as a function of request rate.

The graph shows employing Protractor causes mixer to consume slightly more memory. Across all request rates this increase is on average 1.85%. More importantly the difference stays constant over varying rates which signifies no memory is leaking and the scalability of mixer is not affected.

**Network footprint of the tracing infrastructure**

The network footprint of Protractor’s tracing infrastructure scales linearly with the number of requests. This holds under the assumption that the likelihood of a request being anomalous is independent of the number of requests. For each request a number of attributes are sent from the Envoy sidecar to the mixer. Some of the attributes such as the request path are variable in size, which makes it difficult to determine the size of the attributes payload per request. In our experiment the size of the attributes payload was on average 298 bytes as an upper bound we will use 500 bytes. Mixer receives the attributes, performs anomaly detection, and possibly makes a round trip to a profiler sidecar to dump profiling data with an upper bound of 1000 bytes. Mixer then compiles a span of 216 bytes, and sends the span to the collector. The span is also variable in size so we will again use 500 bytes as an upper bound. The collector processes the span and saves it to external storage, using another 500 bytes. Lastly we add 72 bytes which is the upper bound size of a TCP header. In practice spans are batched together and sent as a single TCP packet, but we add the header size for each span as a worst case. The network footprint from the tracing infrastructure is summarized below:

\[ y = n \times s \times (3 \times 500 + (1000 \times p) + 72) \]

where \( y \) is the network footprint per request, \( n \) is the number of requests, \( s \) is the trace sampling factor, \( p \) is the likelihood of a single request being anomalous. The lower bound when \( p = 0 \) is 1572 bytes, and the upper bound when \( p = 1 \) is 2572 bytes per request. Under 3 kilobytes of network bandwidth per request is unlikely
to have measurable impact when most data centers are equipped with gigabit network links (a difference of 5 orders of magnitude).

**Network footprint of the profiling infrastructure**
The network footprint of Protractor’s profiling infrastructure also scales linearly with the number of requests. This holds under same assumption that the likelihood of a request being anomalous is independent of the number of requests. For each anomalous request a configurable amount of profiling data is written to external storage.

\[ z = n \times s \times p \times d \]

where \( z \) is the network footprint per request, \( n \) is the number of requests, \( s \) is the trace sampling factor, \( p \) is the likelihood of a single request being anomalous, and \( d \) the maximum size of profiling data. In our experiment we set the \( d \) to 5MB which corresponds to 2-4 minutes of profiling data depending on the load. The value of \( d \) can be lowered to reduce network footprint at the expense of the utility of profiling data. The sampling rate can be lowered to reduce network footprint at the expense of availability of profiling data. More specifically, there is a chance that an anomalous request is not traced due to sampling which means Protractor will not detect it.
Chapter 7

Discussion & future work

This dissertation is an attempt to further the field of observability. In section 7.1 we take a moment to critically discuss the limitations of this research, and present other interesting aspects. Next, section 7.2 present possible areas of future work.

7.1 Discussion

Firstly, sampling was put forward as an approach to reduce the number of collected traces. However, when a request is not sampled it will also not be detected for anomaly, nor will profiling information be collected. As a result, not all anomalous requests are detected. At scale this means that common anomalies are statistically likely to be detected, whereas rare anomalies can go unnoticed. If it is deemed important to detect every anomaly the sampling rate can be increased at the expense of higher storage requirements.

Next, though the evaluation demonstrated the design is flexible enough to add profilers for other languages. This does require the profiler to support 1) continuous profiling, and 2) attaching to a running process. In case a deployment consists of services written in different languages, some of which are not supported, Protractor’s utility gracefully degrades to show profiling information for the supported services.

Another limitation was that the anomaly detection was based on a perceived increase in request latency. However, failure modes exists that do not necessarily lead to increased request latency. For example a failure may lead to parts of the code not being executed which will lead to a decreased request latency. Such failures would currently go unnoticed.

Lastly, during the quantitative evaluation it was difficult to find a deployment configuration that stressed the limits of Protractor’s infrastructure. Consequently, we cannot say with complete certainty if Protractor would function adequately in a large scale production environment. We noticed CPU utilization of services was saturated at relatively low request rates, whereas Protractor’s resource utilization remained low. Because we only tested a single topology we may have biased the results if this topology happened to be particularly well-suited for Protractor. Scalability appeared linear from our experiments, but more investigation is required to evaluate Protractor at high request rates. Also, the overhead was measured as a func-
tion of request rate. A sensitivity analysis with regards to factors such as cluster size, and topology would confirm the validity of the conclusions independently from a specific experimental setup.

It should be noted that although we used the Istio service mesh for the implementation of Protractor, this is not a strict requirement if one is willing to sacrifice application-level metadata transparency. Namely, application code can be modified to generate request metadata, which can be sent to a standalone service doing span creation, span augmentation, and anomaly detection. This approach is simpler and better suited if deploying a service mesh is not an option.

Lastly, the Istio project is evolving quickly and has not yet reached a stable release. From talking to the maintainers we learned that the way to extend mixer is likely to change in the future. Whereas now it is necessary to edit the Istio source code and recompile it entirely, in the future it will be possible to deploy adapters as separate processes that communicate with the mixer through remote procedure calls. Running adapters as separate processes will remove the need to modify Istio source code, and makes it considerably easier to modify and deploy adapters enabling new interesting use-cases. For example, this would make it possible to support different anomaly detection policies, and reconfigure them without restarting the control plane.

### 7.2 Future work

We expect to see more research in this area as the need for observability continues to increase. Possible future work to extend this research is presented below.

**Admission controllers**

In order to make Protractor truly application transparent the profiler sidecar should be injected automatically for all Pods that have a container that needs to be profiled. In case of Java containers there are a number of JVM flags that need to be set; this should also happen transparently to the developer to enable true application-level transparency. A mutating admission controller can be used to make these changes to Pod specifications.

**Tail-based tracing**

In the current implementation we use a head-based tracing approach meaning sampling decisions are made when requests first enter the cluster. Exploring tail-based sampling is an area of future work. Tail-based sampling could delay the sampling decision which allows for more informed decisions. These decisions could take various trace properties into account such as response time, or latency. In this case sampling could complement anomaly detection done by mixer, increasing the likelihood that anomalous spans are sampled.

**Sophisticated anomaly detection**

Anomaly detection in mixer is currently done by detecting outliers based on the median request times in a sliding window. Future research could evaluate more sophisticated techniques to do anomaly detection. A machine learning approach could
use k-nearest neighbor, Bayesian networks, hidden markov models, or support vector machines. Envoy also has a filter chain to perform outlier detection which provides another alternative.

**Sticky load-balancing**

If mixer is replicated to handle much traffic, the communication between the Envoy of a service and mixer will be load balanced. Currently supported load balancing strategies would result in the Envoy connecting to different mixer instances. This would lead different instances to all initiate a sliding window for this service for anomaly detection. However, each mixer instance only receives a part of the requests, possibly impacting the accuracy of the anomaly detection. It would be useful to evaluate if this is still a problem at scale, and if sticky load-balancing strategies would successfully mitigate this issue.

**Interpreting profiling data**

The inherent difficulty of interpreting profiling data proved to be a challenge when developing Protractor. While developers see the utility of accessing profiling data, they expressed a perceived difficulty to use such data in practice. It would be interesting to see if profiling data could be parsed to automatically detect common problems, and present these in a more user-friendly manner. This would reduce the learning curve, and likely increase the adoption rate.

**Storing an entire trace**

When mixer detects an anomaly it currently dumps the profiling data of the service that generated the span. Another strategy could be to dump the profiling data for all spans in the trace. This corresponds to all services that were involved with handling the request since it entered the system. This approach is useful if the failure is related to other services, and cannot be debugged in isolation.

**Deduplicating profiling data**

The Java Flight Recorder source code, and the profiling format will be open sourced in the near future. This will open a range of new possibilities. An example is to deduplicate two overlapping profiling files by parsing the events and deleting duplicates. Another interesting possibility is to create a web interface for the profiling data, and integrate this with tracing tools such as Jaeger. This would greatly benefit user experience as operators no longer need to download the profiling files to their local machines, and are no longer required to download Java Mission Control.

**Open-source**

In the future Protractor can be open-sourced to engage with other companies that have similar needs. By creating a community and collaborating with developers from all over the world it is possible to reduce maintenance costs, improve the software quality, and benefit everyone who is involved.

**Evaluation in a production environment**

Evaluations are carried out on synthetic workloads and benchmarks. It would be interesting to see how Protractor stands up in a production setting with real traffic
and a multitude of heterogeneous services. The major difficulty is that all services are required to run the sidecar proxies, and all services on the request path are expected to forward tracing headers. This makes it difficult to partially roll out the solution to a subset of services, instead requiring modifications to all services in a deployment.
Chapter 8

Conclusions

In this thesis we have presented Protractor, a system leveraging distributed tracing in service meshes for application profiling at scale. It improves the observability of deployments by making contextual profiling information available to operators for root-cause analysis. This need was discovered through interviews with engineers, within and outside of the company, surveying related work, and several observations from the KubeCon + CloudNativeCon Europe 2018 conference. Compared to existing state-of-the-art systems that provide tracing or profiling in isolation, Protractor correlates and integrates the information from both sources to provide better contextual information. We provided the conceptual contributions needed to do distributed profiling, and the technical contributions of a realistic implementation of a distributed profiling system and its evaluation. The evaluation demonstrated Protractor is able to do distributed profiling with minimal application-level changes and negligible performance overhead. From the usability survey we learned that users may have difficulty learning how to use the system, but do see it as a worthwhile investment. Protrator’s utility is further evidenced by the positive reactions from engineers within the company. Lastly, the flexibility of the design made it easy to add support for a different profiler. This opens up the possibility to meet changing needs and requirements in the future. In the next sections we will conclude this research by answering the research questions, and by reiterating how well design goals were met.

8.1 Concluding research questions

In chapter 5 we defined the main research question, and broke it down into three smaller sub questions. We will now first answer each of the sub questions, and see how doing this consequently answers the main question.

How to decide which profiling information is relevant to store?
This question emerged from the observation that storing all profiling data from all services is not feasible. Instead, only profiling data that is likely to be relevant in the future is stored by performing anomaly detection based on predefined failure mode. A failure is defined as a service that serves requests much slower than usual. This
How to correlate distributed traces with profiling information?
Profiling and distributed tracing are currently being used as separate solutions to increase the observability of a system. We observed that by correlating data from these two sources we can obtain a notion of distributed profiling, which provides more relevant and contextual insights than the systems could in separation. To perform the correlation we augmented anomalous spans with a link to the profiling data originating from the corresponding service. This data contains events leading up to the anomaly and can be analyzed by operators to debug problems and perform root-cause analysis.

How to make it easy to find relevant profiling events in the profiling data?
Profiling data contains events about different threads, possibly executing concurrently. This question pertains finding a way to filter down the events to those that are relevant to a particular request. In Protractor we addressed this issue by modifying web servers to submit custom events to the profiler containing the trace- and span identifiers corresponding to the request. As a result operators are now able to search for the custom event in the profiling data in order to find other related events.

How to provide an integrated system to gather and correlate tracing and profiling information to provide operators with the necessary context for effective root-cause analysis?
Answering the previous three sub-questions provide the building blocks to answer the main research question. The main question was answered in this work by implementing Protractor: an integrated system to gather and correlate tracing and profiling information to provide operators with the necessary context for effective root-cause analysis. The system satisfies all of the constraints, namely: the design is ethical as it helps engineers debug problems, the design was implemented by the end of April which is within 22 weeks, the design supports services running in Kubernetes, and the design supports services written in Java.

8.2 Concluding requirements
The system requirements specified in chapter 4 are all satisfied. They are repeated below along with how they are satisfied by Protractor:

1. collect and store spans for every service: Protractor uses proxies running alongside all services to send meta information about requests to the mixer, which will use these to generate spans.

2. perform anomaly detection based on service processing time: Mixer performs anomaly detection based on the median of a sliding window of previous request latencies.

3. profile and store profiling data for anomalous services: Protractor uses the profiler sidecar running alongside all services to profile applications.

4. correlate spans with profiling data: Mixer augments spans with profiling data.
5. *provide a user interface to visualize traces and download profiling data for offline analysis*: Protractor extends the Jaeger user interface to add the possibility to download profiling data to a local machine.

6. *augment profiling data with request identifiers*: Protractor modified web servers to generate custom profiling events containing the trace- and span identifiers.

### 8.3 Concluding design goals

The design goals specified in section 4.4 have each been individually evaluated in chapter 6. Based on the results of this evaluation we will show how well Protractor satisfied the different design goals. This is summarized in table 8.1.

<table>
<thead>
<tr>
<th>Design goal</th>
<th>Satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>Partially</td>
</tr>
<tr>
<td>Application-level transparency</td>
<td>Partially</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Yes</td>
</tr>
<tr>
<td>Scalability</td>
<td>Yes</td>
</tr>
<tr>
<td>Low-overhead</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 8.1: Summary of the extent to which the design goals have been met.

The first goal of *Usability* is partially satisfied. The outcome of the System Usability Survey was 66 which was average to products in other areas, but low scores on questions regarding learnability and coherence suggested possible improvements in these areas can be made. The second goal of *Application-level Transparency* is partially satisfied. The evaluation showed how two out of three modifications still need to be done manually. In the future these changes may be applied automatically with a custom admission controller. The third goal of *Flexibility* is satisfied. The evaluation demonstrated how to add support for the *perf* profiler in under 100 lines of code, without changing any other parts of Protractor’s infrastructure. The fourth goal of *Scalability* was satisfied. Three experiments showed Protractor causes less than 3% slowdown of request latency, as long as the infrastructure is allocated sufficient resources. The last goal of *Low-overhead* was satisfied. CPU utilization of the mixer was shown to be 38% higher, but linear with respect to request rates. Mixer’s memory consumption with Protractor was slightly affected, but remained stable over increasing request rates. Network overhead from generating and storing profiling data is proportional to profiling data size. It can be reduced by decreasing the trace sampling rate at the expense of availability of profiling data, or by decreasing profiling data size at the expense of utility.

To the best of our knowledge, Protractor’s approach to distributed profiling is the first of its kind. By integrating tracing and profiling it provides previously inaccessible contextual information that facilitates root-cause analysis. Within the company the project was received with much enthusiasm, and by sharing our findings we make a humble attempt to advance scientific progress towards more observable distributed systems.
Bibliography


Appendix A

Architecture Evolution

v1 Figure A.1. The first iteration used a local Envoy deployment with docker compose [26] to launch three different services. Profiler was deployed in a separate process inside the application containers, breaking process isolation and application transparency. We extended the proxy to call a LUA script for each incoming request. The script was called before and after the request sending the request ID to the profiler which would start and stop a timer. If the duration was longer than usual, the request is anomalous and another call to the profiler was made to trigger a dump. The dump was done using the jcmd command line tool. At this point we had no way to store the profiling data yet. Spans were generated by the proxies. The LUA script executed in-band and made at most 4 http calls which slowed down request times by an order of magnitude, which made this approach unsuitable.

v2 Figure A.2. In the next iteration, instead of extending the proxy to do anomaly detection, we deployed a separate process that would watch the Jaeger collector for newly created spans. If a span was found to be anomalous, it would trigger a dump on the service. This was much faster and out-of-band, but hard to scale up. Multiple copies of the monitor process would need to synchronize which spans to process. We also used JMX instead of jcmd, removing the need for a separate binary in the profiler.

v3 Figure A.3. In the third iteration we switched to Kubernetes, added GFS, and Cassandra as storage backends. A collector plugin was used for anomaly detection instead of a separate process. This scaled better than the previous approach, because there was no need for synchronization between these processes. However, the collector process was still centralized, and incurred significant slowdown.

v4 Figure A.4. This time we tried to extend the Jaeger agent to do anomaly detection in order to distribute the anomaly detection work across all services. Halfway through the implementation tracing in Istio changed to an approach where spans were generated in the mixer. This would present a stateless, replicable component to do anomaly detection. The new direction voided this architecture, and motivated the final architecture.
APPENDIX A. ARCHITECTURE EVOLUTION

Figure A.1: Architecture v1.
Figure A.2: Architecture v2.
Figure A.3: Architecture v3.
Figure A.4: Architecture v4.
# Appendix B

## Usability Survey

### Protractor Usability Survey

Please check the box that reflects your immediate response to each statement. Don't think too long about each statement. Make sure you respond to every statement. If you don't know how to respond, simply check box 3.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Strongly Disagree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. I think that I would use this system frequently.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. I found the system unnecessarily complex.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. I thought the system looked easy to use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. I would need the support of a specialized person to be able to use this system.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. I found the various functions in this system were well integrated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. I thought there was too much inconsistency in this system.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. I think that most people would learn to use this system very quickly.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. I think the system is very cumbersome to use.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. I would feel very confident using the system.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. I would need to learn a lot of things before I could get going with this system.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comments: