Bauhaus-Universität Weimar
Faculty of Media
Degree program Computer Science for Digital Media

Toward Automated Resolution of Microservice Configuration

Master's Thesis

Nicolai Ruckel
born August 30, 1990 in Kassel

Matriculation number 100291

First referee: Prof. Dr. Norbert Siegmund
Second referee: Junior Professor Dr. Florian Echtler

Submission date: September 25, 2018

Statutory Declaration

I hereby affirm that the master's thesis at hand is my own written work and that I have used no other sources and aids other than those indicated. All passages, which are quoted from publications or paraphrased from these sources, are indicated as such, i.e., cited, attributed.

This thesis was not submitted in the same or in a substantially similar version, not even partially, to another examination board and was not published elsewhere.

Nicolai Ruckel

Weimar, October 8, 2018

Abstract

A current trend in modern software development is the architectural style of microservices, which splits up the functionalities of old monolithic applications. These microservices are highly scalable and there exists a variety of different technologies to support developers set up the required environment. However, the number of tools comes with a cost. Each has to be configured and often the configuration of one tool affects other tools. Sometimes, the same option can be configured in different software layers. Contradictions in these configurations can lead to erroneous behaviour or severe instability and security problems.

In this thesis we developed a graph-based approach to model the configuration options of technologies as nodes and their dependencies as edges to detect possible misconfigurations and recommend solutions. We call those graphs configuration networks. The main difference to other approaches is that the conflict detection is agnostic about the tools represented in the graph and we do not require developers to learn an additional software or programming language. While we do not have a production-ready application, our prototype has already shown promising results.

Contents

Acronyms

API	Application programming interface
AWS	Amazon Web Services
CD	continuous delivery
CI	continuous integration
CsCE	Cross-stack Configuration Error
HTTP	Hypertext Transfer Protocol
JDK	Java Development Kit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LXC	Linux Containers
OCI	Open Containers Initiative
OS	Operating system
OSS	open-source software
REST	Representational state transfer
SCI	Shared Configuration Item
SSH	Secure Shell
URL	Uniform Resource Locator
VCS	Version Control System
VM	Virtual Machine
XML	Extensible Markup Language
YAML	YAML Ain’t Markup Language

1 Introduction

Today, more and more software systems are developed in small, agile, and self-organized teams. Such an agile development process enables iterative development or rapid prototyping, in which a software system is constantly enriched with new functionality. Software is delivered early, and changes, such as bug fixes and new features, get vastly integrated to create multiple working builds per day to get user feedback as soon and as often as possible. Such a continuous software development requires the use of automation tools. Of course, those tools are not restricted to agile development. Organizations with other development processes can make heavy use of them to improve the work process, too. Therefore, developers have to learn and work with a number of different concepts and tools.

One example is the *continuous integration* (CI), where the integration process is transformed from being complex and time-consuming to being a non-noteworthy event where errors can be found quickly [1]. Typically, developers use a combination of a *Version Control System* (VCS), software tests, and a CI server. First, a developer checks out a working copy of the mainline branch of the VCS and makes some code changes. Then, the code is tested on the developers machine using a test suite, which typically consists of several unit tests [2]. If written properly, unit tests avoid breaking changes and acceptance testing ensures that the functional requirements of the software are fulfilled. When no errors occurred, the working copy of the developer has to be updated with the current version of the mainline since other developers could have made changes, too. Then the changes get merged into the mainline branch and get tested on the CI server. This process is usually automated, for example with Git Hooks [3]. The details of this process depends heavily on the combination of the used VCS, programming language, and choice of automation tools. Errors, occurring at any time during this process, can be fixed immediately. With this process, multiple stable builds are created every day, each with new integrated features.

CI is often used in combination with *continuous delivery* (CD). In CD, new features or bug fixes are released into production as soon as they are implemented and tested [2]. Since everything is well tested with CI, these releases can be safely rolled out to the users without worrying about stability. Manual tasks, such as copying the binaries to a web server or

configuring the environment to make the software usable, get automated, which leads to stable and reproducible builds. CD allows shorter release cycles and therefore faster user feedback. If new bugs that were not caught by the test suite, are introduced, they can be found quickly and become easier to fix because they are probably in code that was recently written. Since the software is delivered very often, it can be efficiently automated and the delivery process gets very stable. This also enables easy rollbacks and immediate releases if necessary.

All of this automation enables the *DevOps* culture. Its goal is to reduce the separation of operations and development and therefore decrease the time until a new implemented feature or a bug fix is introduced in the production build. This removed separation also leads to a shared responsibility of developers and operations teams over the whole development cycle and lifetime of the software as well as the infrastructure, such as cloud environment configuration. Since developers have to care about operational problems and operations teams can give fast feedback, they can improve the deployment process together with automation tools [4] to model an *Infrastructure as Code*, where interactive configurations are replaced by automation scripts. The organizational culture of DevOps also demands teams to be autonomous which allows them to make those decisions on their own. This increased collaboration between development and operations (hence the name) leads to more productivity [5].

In parallel, in recent years, software systems became more and more complex with the integration of third-party dependencies and the need for different configuration options for different target systems. To manage those dependencies, build tools are used, which are often extensible with plugins for additional functionality, such as code quality checks [6]. Some build tools such as Gradle or Maven even provide wrapper shell scripts that pull the current version of the build tool building of the software application on machines without installing the build tool. This simplifies the initial configuration of CI servers, which depends on the used build tool. Some build tools use markup languages, such as XML, others use general-purpose programming languages, such as Groovy, or have their own syntax, such as Make.

To conclude, various tools and programming languages are used, each with their own dependencies and configuration options. They can even depend on each other, which means that changes to the toolchain can require non-trivial alterations of various configuration files and code artifacts. This leads to a situation where all these tools, which originally had the purpose of simplifying the development and delivery process, form a complex problem

of their own.

All of the mentioned challenges arising from configurability apply to the development of web applications. A current trend in this domain is to adopt a modern architecture style called *microservices*. Microservices replace the old code bases of monolithic applications which rely on legacy technologies and are hard to upgrade because of tight coupling of their internal components. With microservices, an application is split up into many smaller services that call each other. Since scaling is done by continuously adjusting the number of instances of a microservice, one cannot rely on fixed addresses for inter-microservice communication. Thus, mechanisms for *service discovery* are needed. This also includes monitoring of the services to isolate errors and *Application programming interface* (API) documentation to enable efficient collaboration between different teams. All of these aspects are related to the infrastructure at which the microservices are running on, again requiring a substantial amount of configuration.

While there are many advantages of the different kinds of microservices over monolithic approaches, such as better scalability and higher agility, the required environment is highly complex. Therefore, a number of different technologies to manage the combination of microservices and general software development infrastructures emerged in the last years. A lot of those tools have more than one purpose and some of these purposes are optional. Also, these tools do not exist in a vacuum but often depend on each other in their configuration. This can lead to a *Cross-stack Configuration Error* (CsCE) [7], which is an error that occurs in a software when a developer changes the configuration in another software or tool. CsCEs can result in unexpected behaviour or crashes. With an increasing number of used tools, the number of additional required changes can rise, too. In a microservice environment, these errors can cascade to other services as well, breaking the whole infrastructure.

One introductory example would be that a microservice defines its port in the source code:

```
// port of the service as member of the main class
private int port = 8081;
```

If this software should run in a containerized environment, such as Docker, we need to expose this port in the dockerfile, too:

```
# make port 8081 available from outside of the container
EXPOSE 8081
```

Now, we created dependencies between a code artifact and a configuration file, such that we need to keep both properties consistent. This is challenging as both properties are in very different development artifacts and there is no obvious connection or tracing available. Of course, those dependency problems can get much more complicated. Some settings can even contradict each other. A good example, given by Sayagh et al. [7], is the configuration of the maximum memory a PHP script can use in a web application. This can be specified in the web server, the PHP interpreter, or the content-management system.

There is a high demand for a good solution for these kinds of problems, as called by Google engineer Matt Welsh "an escape from configuration hell" [8] or "*the* next big challenge for the system research" by John Wilkes [9].

To improve these tedious and error-prone manual configuration processes of software, researchers tried to develop mechanisms to detect and solve those conflicts automatically. Unfortunately, existing solutions are often not very reliable or are restricted to isolated toolsets (e.g. Docker and Java, or PHP) [10].

The goal of this thesis is to develop a tool to represent the artifacts of the whole CD pipeline, independent of the used technologies as nodes in a network. Methods for integrating technologies into the network can be developed independently from other technologies. This means that providers of a technology can develop their own parsers for their configuration files without having to look into parsers of other technologies. Edges between those nodes correspond to a dependency between them.

In this thesis we contribute an approach that provides a basic for using networks to represent dependencies and a possibility to find configuration conflicts by interpreting the changes in the network. The conflict detection mechanism does not have to be aware of what actual technology or configuration options the nodes in the network represent. A categorization into concepts, artifacts, configuration options, and shared configuration items is enough. This allows developers to use this in future work together with an automatic detection of the used software stack and their used features, in existing projects to detect or prevent possible cross-stack configuration errors.

The next chapter presents some background on microservice architecture and the commonly used software. In the third chapter we will explain how we build and use configuration networks to solve dependency problems between the configurations of different

technologies in a project. Then we will evaluate our approach with a synthetic and a real-world example in the fourth chapter. Finally, in the last chapter we will discuss our results and give and overview of the future work for configuration networks.

2 Background and Related Work

In this chapter, we introduce the concept of microservices, its requirements of the software infrastructure and some of the most common software solutions for those requirements. Furthermore, we provide an overview of related approaches targeting the resolution of cross-component configuration errors.

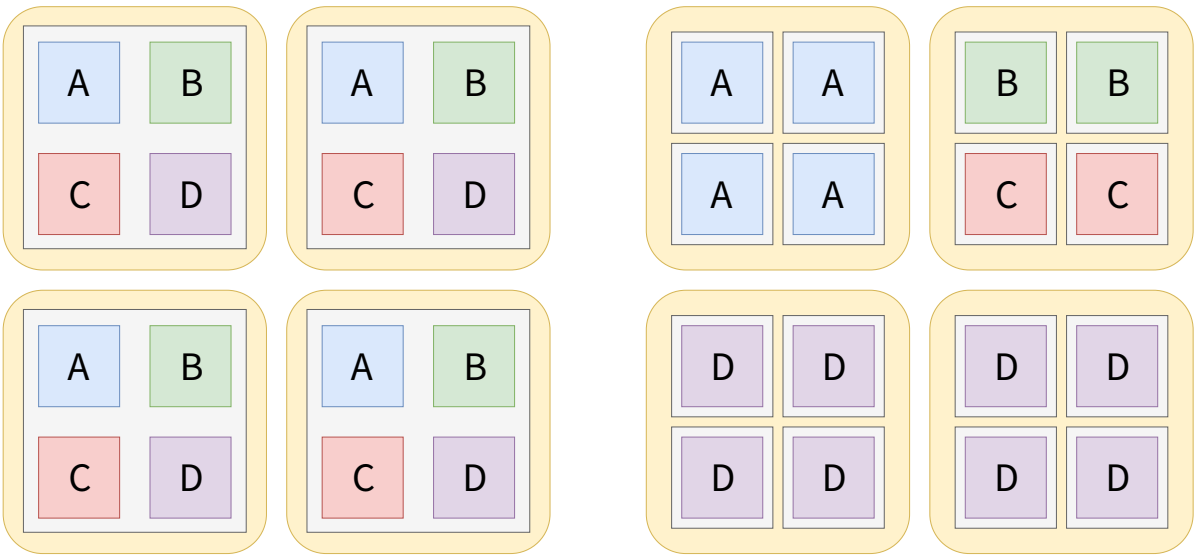
2.1 Microservices

In 2011, the software architecture term *microservices* emerged [11]. This architectural style became very popular since then. The main idea is to split up an application into small independent parts (the microservices) that each have their own process and can communicate with each other over the network.

The usual approach to explain microservices is the comparison to the monolithic architecture [11], where server applications are build as a single unit. This single unit handles *Hypertext Transfer Protocol* (HTTP) requests, usually contains a database backend and does all the business logic. Such a system is relatively easy to test. Scaling can be done by deploying additional instances of the whole system (see Figure 2.1a) behind a load-balancer. The problems with monolithic systems are, that you have to choose one software stack for all functionality even if it is not optimal for some parts. Also, even if the application is build very modular, it is often hard or impossible to make changes in one part without affecting others. Deploying can also only be done as a whole, because the entire system has to be rebuilt. This can often lead to slow development and frustrating legacy code bases that cannot be upgraded to avoid breaking other modules.

In the microservice approach, every microservice is an independent software developed by different teams respectively. Each team can choose the most fitting programming language [1], build system, database and the rest of the software stack. The different microservices communicate over lightweight interfaces like *Representational state transfer* (REST).

[1] Some organizations may have some language constraints, e.g. Netflix and Twitter mostly use the *Java Virtual Machine* (JVM) [12]



(a) One monolithic application on each server with components A, B, C, and D.

(b) A, B, C, and D as microservices in containers, distributed on the servers according to the demand.

Figure 2.1: Scaling of monolithic and microservice architectures.

This leads to two advantages over the monolithic architecture: First, when some team decides to rewrite everything in a new programming language, other microservices are not affected and do not have to be redeployed. Second, each service can be scaled independently from the others (see Figure 2.1b).

The *micro* in microservice does not relate to the number of lines of code but to the scope of functionality. Similar to the Unix philosophy [13], each microservice does one thing, and does it well.

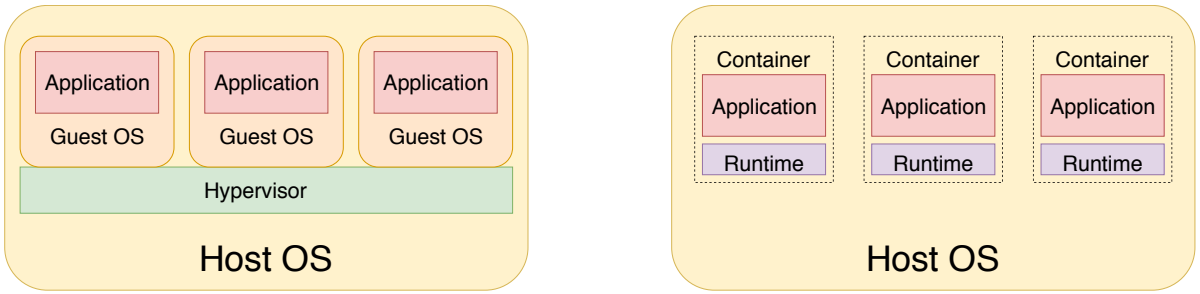
Microservices are not the best solution for all problems though. To successfully use microservices, a number of technologies has to be learned and used. This includes service discovery, monitoring, CD/CI which will be explained in later subsections, and finding the right size for the various microservices. Also, building robust microservice infrastructure is very difficult since testing is much harder. In addition, one has to factor in the network when one microservice calls another one. The two main problems here are latency and lost packages.

In the following subsections, we will explain some of the concepts and capabilities that are required for microservices.

2.1.1 Containers

Containers are one of the key technologies for microservices. One physical machine for each microservice is very impractical since there can be a huge number of service instances that have to be started and destroyed on the fly. Each physical machine should be able to handle multiple kinds of microservice instances. Those should not be restricted to live on a certain physical machine, since both would lead to either idling machines or a physical constraint on the scalability.

A naive solution is to have a *Virtual Machine* (VM) for each instance but this would be too much overhead. Each instance of a microservice requires the same configuration of the *Operating system* (OS) and usually all microservices run on Linux OSs. Therefore, there would be a lot duplication for each client, which demands a lot of space (see Figure 2.2a). In addition, a hypervisor is needed and the startup time is relatively high since a whole OS has to be booted.



(a) Virtualization (b) Container

Figure 2.2: Comparision of Virtualization and Containers [14].

A solution for this problem is the use of containers. Containers share the kernel with the host which saves space and reduces the startup time. Because of the small footprint, containers are highly portable. The current container technologies originated in *BSD Jails* [15, 16]. Paul Menage implemented the first version for the mainline Linux kernel [17]. Those were renamed to *control groups* later [18]. *Control groups* enable the host to share and also limit resources (CPU and memory) each container can consume [19, p. 9].

Namespaces limit the visibility a process has on other processes, networking and file systems [19, p. 9]. This is necessary since deploy scripts for software that runs in containers often needs a root user (e.g. to install packages with the package manager) but this root user clearly should not have the same permissions on the host system. *Control groups* and *namespaces* are combined in *Linux Containers* (LXC) [20], which provides tools for using con-

tainers [14].

Since June 2015, there is also the *Open Containers Initiative* (OCI), a project under the Linux Foundation. Its goals are to create standards for runtime and image specifications for containers [21].

2.1.2 Container Orchestration

Containers have to be instantiated and destroyed continuously depending on the demand. They can be deployed on many machines/server hosts. Because deploying and monitoring them manually would be very inefficient, so additional container orchestration tools are necessary. Therefore, container-orchestration has a number of different features [22]. First, it manages the deployment of the containers. This includes the initial deployment, updating the containers when new images are built, and deploying additional containers for scaling purposes. Since the orchestration knows about all container hosts, it can also be used for service discovery, load-balancing, and health monitoring. With health monitoring, it can also replace crashed containers to ensure the availability of a service.

2.1.3 Service Discovery

In a microservice environment it is mostly unknown how many instances of each service exists since services are constantly instantiated and destroyed. For this reason, one cannot call a service by just knowing its IP address. There has to be some mechanism to make the correct addresses available to the other services, called *service discovery*. There are two possible variants of service discovery: server-side and client-side discovery. In both cases, services have to register themselves at a service registry.

In the server-side discovery model (see Figure 2.3), a client sends its request to a load-balancer which acts as a proxy [23]. This load-balancer then chooses an instance of a service from the service registry and propagates the request to the actual service. This approach is very easy to use and most of the time build in into the container orchestration. Because every request has to be handled by the load balancer, this could decrease performance and reduce the resilience.

Another possibility is the client-side discovery (see Figure 2.4). Here, the client directly receives the addresses of the instances from the service registry, caches them, and then chooses one instance with its own load-balancing strategy [23]. While this negates the dis-

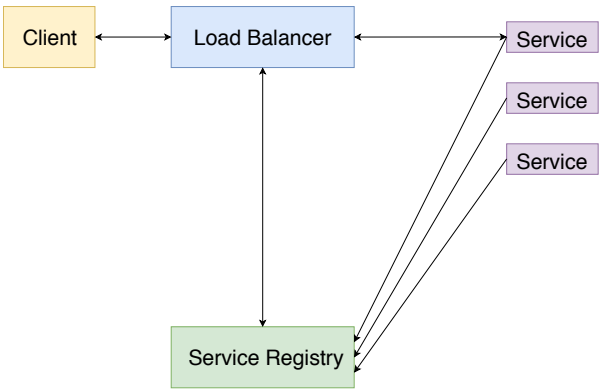


Figure 2.3: Server-side Discovery.

advantages of the server-side discovery, it increases the complexity in the clients. Entries in the cache of clients can become stale.

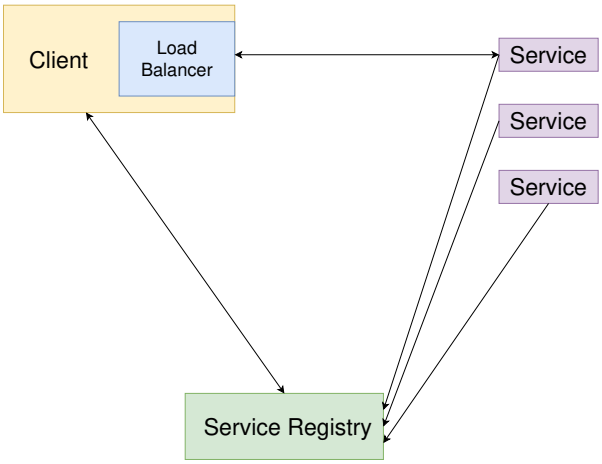


Figure 2.4: Client-side Discovery.

2.1.4 Continuous Delivery and Continuous Integration

Although CD and CI are different concepts, with different purposes, they are often used in combination and entwined with each other. The main idea of CD is that all changes should go safely, without hassle into production quickly and on demand. This requires the code base to be always in a deployable state. Thus, traditional integration and testing phases have to be eliminated. With separate integration and testing phases, it would take a long time for new features to be available in the master branch. Of course, all of this has to be achieved without sacrificing stability. In fact, there are surveys [24] that claim that reliability and stability increases with these methods. In CD, every repetitive task, such as unit and

regression testing and delivery, becomes automated in the *delivery pipeline* (see Figure 2.5), which is the starting point for every CD / CI development environment, to allow developers to spend their time on actually developing.

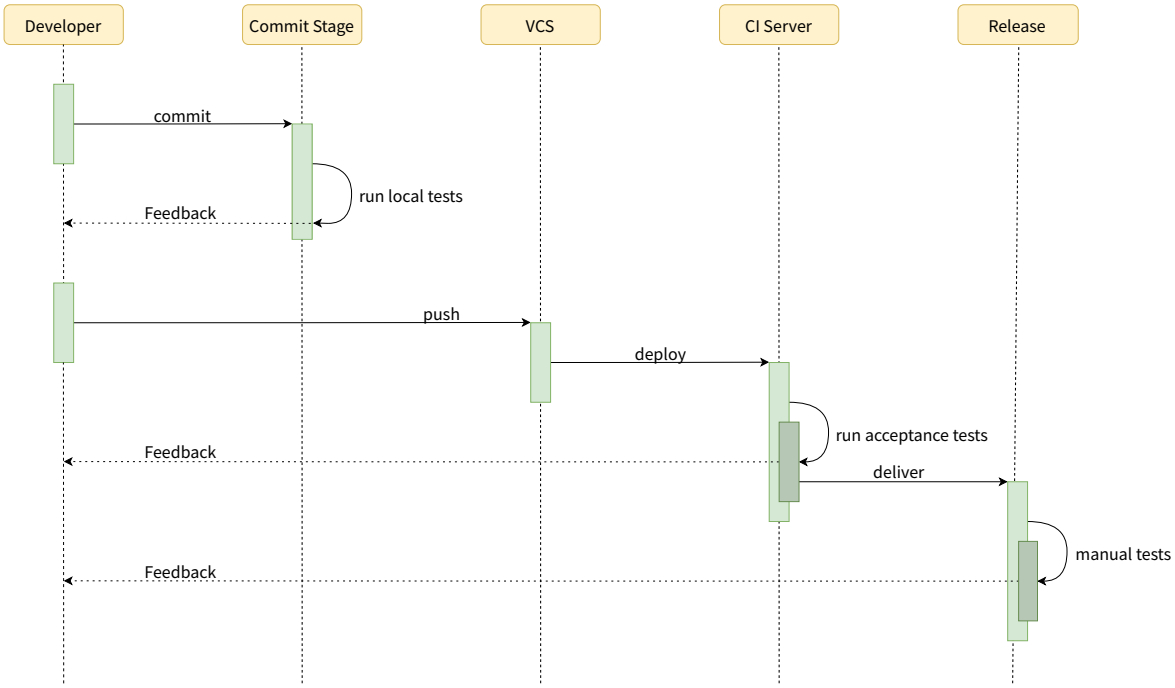


Figure 2.5: Delivery Pipeline

To remove integration and testing phases without reducing the quality, automated test suites are used. Unit and functional acceptance tests can be run locally after each commit into the VCS to discover bugs very fast. Those tests are automated since it costs a lot of time and is error prone when done manually by people. Developers have to write new unit tests every time they discover a new bug that was not caught by the test suite. After the local tests are run and the developer attempts to merge with the master branch, the automated acceptance tests are run on the CI server. This way, the software has high stability from the beginning for each release. Those releases can be already tested manually for nonfunctional acceptance, for example by customers, to give additional feedback that is not possible with automated tests. This also includes usability and exploratory testing. Problems with those or the demand for additional features can be brought to the developers early enough to allow them to work on those. Since code is deployable at any time, with an automated delivery process that is applied multiple times each day, this process becomes well-tested and thus robust and almost risk free.

Another principle is to work only in small batches. This allows new features to be finished

quickly. As soon as those features are ready, they are integrated in the master branch. In contrast to traditional development processes, everyone is responsible for everything, so developers cannot just work on their own isolated branch until an integration phase begins. Since code is constantly merged in the master branch, with proper testing, the integration happens automatically and immediately. This also gives the developers feedback very early, which prevents unnecessary work and is also more motivating for developers, thus increasing efficiency.

Popular automation tools for CD are Jenkins and Ansible. For Jenkins [25] the delivery pipeline is configured in a `Jenkinsfile`, a text file which can be written in either Java, JavaScript, Ruby, Python, or PHP that can be checked into a project's VCS repository [26]. Ansible [27] is an automation software, that connects to remote machines over *Secure Shell* (SSH) to run certain commands. It allows to either run ad-hoc commands or to write scripts in YAML, called playbooks [28].

2.1.5 External Configuration

Some services need to be configured on how they have to connect to external services, for example, URLs or login credentials [29]. This configuration can change during development or depending on the environment. For example, there could be a different database for the testing environment than in production. Since there can be any number of instances for each service, it is not feasible to restart or even recompile all of them after the configuration or the environment changes. Therefore, external configuration servers (see Figure 2.6 [30]) exist. At startup, a microservice gets the location of the configuration server via the service discovery and requests the configuration. The configuration server then returns the configuration that is stored in some VCS repository. Each service has an API to be forced to get the new configuration and apply them at runtime.

2.1.6 Monitoring

With a high number of different services working together, it becomes crucial to monitor all of them. When a failure in the system occurs, one has to determine in which service the failure occurred and what went wrong. A service can log multiple things such as health status, hardware utilization, number of database connections, number of requests, response times, health status of dependencies, or errors and exceptions. Good monitoring includes warnings for anomalies such that developers can fix bugs before they become a problem.

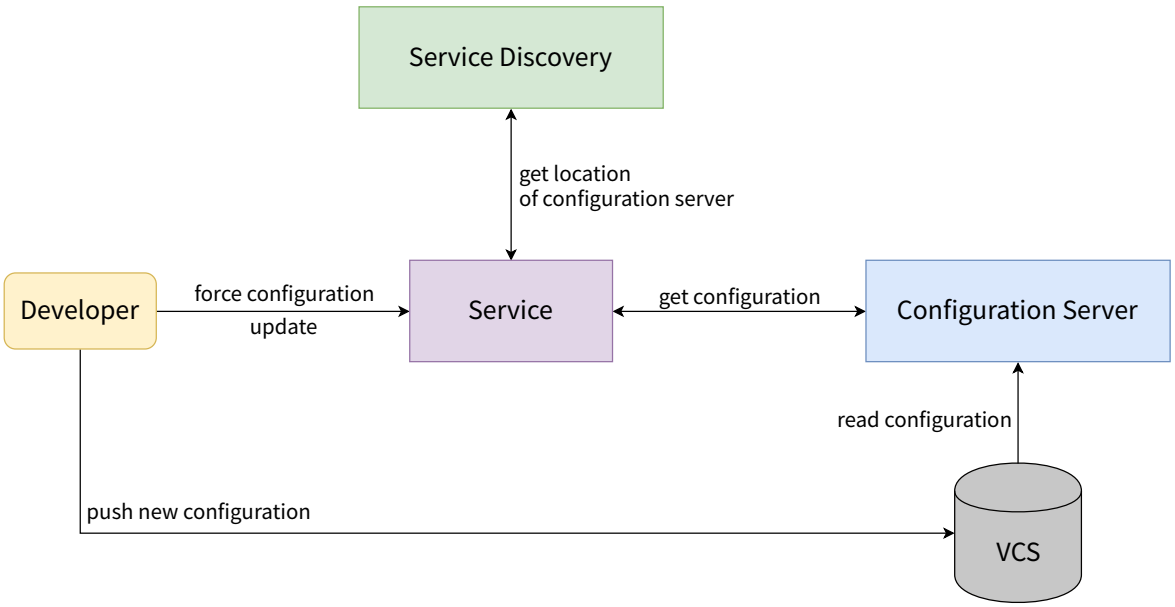


Figure 2.6: Configuration server setup.

Alerting means not only to warn of potential errors, but also to avoid too many false positives (i.e. noise). Monitoring can become very hard when the communication between services is asynchronous.

2.1.7 Sidecar

To get all the functionality needed for a microservice environment, different libraries were developed to reduce the amount of work for developers to integrate the different technologies. However, those libraries are often very big and complex and therefore restricted to one programming language and not reimplemented for others [31]. This negates the advantage that microservices theoretically can be implemented in different programming languages, since developers understandably made heavy use of those libraries and implemented their services in the corresponding languages.

To reclaim the possibility to use arbitrary programming languages for implementing a microservice, *sidecars* or service proxies are used. Those are services in their own container, deployed together on the same host with the services that need them [32]. By deploying them together with the microservices, they do not have to communicate over the network, reducing latency and package loss. This allows developers to use well tested technologies for basic functionality that is common between different services, such as monitoring or logging, while allowing them to implement their microservices in the programming lan-

guage of their choice. Additionally, since the sidecar functionality is separated from the microservice itself, errors are encapsulated in the sidecar container, therefore preventing cascading errors.

2.1.8 Build Tools

Build tools are an automation of the compiling and linking processes to generate software builds. They have different options for the configurations (targets) of the software they build, such as debug and release targets or targets for different hardware platforms. Additionally, they are responsible for running automated test suites and generating test reports. Modern build tools usually manage the integration of third-party libraries, for which some have dedicated repositories. They can often be extended through plugins. Plugins can provide other helpful functionality such as code-quality checks or preprocessing capabilities.

The configuration of build tools is usually done in specific configuration files. These files can be very different between different build tools. Some build tools are configured in markup languages, others are used with general-purpose programming languages, and some have their very own syntax. In a microservice environment, different build tools have to be used, since they are often tied to a specific programming language or a small set of programming languages.

2.2 Common Microservice-related Tools

For all of the previous mentioned technologies, there are multiple *open-source software* (OSS) tools that attempt to provide these functionalities. Many of those tools have more than one purpose and have different optional features. They can also sometimes integrate other tools. In this subsection, we give an overview over some of the more popular tools: how to include them, what is their functionality, which options they provide, and what dependencies they have.

2.2.1 Eureka

Eureka is a software by Netflix, designed for service discovery in the *Amazon Web Services* (AWS) cloud. It also has basic load balancing capabilities. Eureka provides a REST-based

endpoint and a Java client with additional capabilities. Since only the Java client provides all functionality, Eureka is hard to use in non-Java environments.

2.2.2 Docker

Docker is a popular open source container platform [33]. The purpose of Docker is to solve the *works on my machine*-problem [34, p. 8-9]. The problem is that developers usually have different versions of libraries on their machines than the ones on the production server. When the code is deployed to the production server, it does not work and the developers claim that everything worked fine on their machines and leave the problem to the server operators. It is often not possible (or even desirable) that every machine has the same version of the same libraries, runs the same OS, and every developer uses the same tools. Docker solves these problems with a software that makes containers easy to use on different OSs with *union file systems*, which are small images with a snapshot of the container's file system. [19, p. 10]. Therefore, developers can build their services for the desired and pre-built docker image, and test everything on their own machines.

Docker has its own built-in mechanism of container orchestration, named swarm mode, which replaced the now deprecated Docker Swarm [35]. The swarm mode is capable of container orchestration, service discovery, and load balancing [36].

The configuration of a docker container is in the `dockerfile`, a plain-text file which consists of Docker instructions.

2.2.3 Spring

Spring is a popular Java Framework by Pivotal Software for building Java-based web applications [37]. It contains the component Spring Boot that supports various APIs such as REST or Websockets. Applications built with Spring can either use SQL or NoSQL databases. As a runtime, developers can choose between Tomcat, Jetty, and Undertow. To configure Spring applications, Java annotations are used (see Listing 1).

With Spring Cloud, additional tools for distributed systems, such as external configuration [39] or service discovery, can be integrated easily.

They also provide Spring Initializr [40], a web service for generating Spring projects with configurable dependencies.

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
}

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Example.class, args);
    }
}
```

Listing 1: Hello World example from Spring Boot documentation [38, chap. 11.3].

2.3 Problem Definition: Configuration Errors

Due to the high complexity of distributed cloud-based systems, the main cause for their failures are because of misconfigured components, which are hard to debug [41]. Systems often have a large number of configuration options. For example, the MySQL database has over 400 different parameters, and usually the number of options increases with new releases [42]. Those configuration options and their constraints are often not well documented or designed [43]. Additionally, parameters can depend on other parameters of the software or the system/software stack [44]. One big problem is that the occurrence of a configuration error does not necessarily mean that parameters have illegal values. Yin et al. [45] state that about half of the configuration errors have legal values, but do not work with the underlying system of the desired functionality.

Xu et al. [41] categorize configuration parameters as follows:

- *Environment*: Contains information that is required during the runtime of a program. Examples for this are local ports, location of service discovery servers, or the location of databases. These kind of parameters can easily break for a component, when another component changes its configuration.
- *Features*: Specifies used optional functionality.

- *Resource*: Defines which resources are provided and with which constraint, such as hardware, memory allocation, or software resources such as the number of connections to databases.
- *Performance*: Allow to optimize the software's performance.
- *Security*: Restricting access on resources and API of a software.
- *Reliability*: All parameters for specifying the availability of a service, such as data backups or standby servers.
- *Modules*: Used libraries.
- *Versions*: The specific versions of used libraries.

The configurations are usually stored in text files in various formats (*JavaScript Object Notation* (JSON), XML, software-specific formats) for cloud-based systems since they are often run on Unix-like systems. Another possibility would be a centralized configuration database such as the Windows Registry that stores settings of the Windows OS.

Configuration gets much harder in big systems with many services, each managed by a different team. Those teams have no view or control over the components so that cross-component errors can easily occur [45] and are hard to debug when developers are not even aware of the influence of other components [8]. Additionally, most solutions for configuration management consider only single components.

There are different approaches to ensure the correctness of configurations. The simplest and most common one is probably to define certain rules for valid configurations. Those rules can be checked before running an application to detect invalid configurations. However, as mentioned earlier, this approach only covers some cases of configuration errors.

Another approach is to try to use automated learning to recognize patterns of correct configurations of systems. Then, new configurations can be checked whether they are similar to the correct configurations and if not, assume that they are erroneous. Yet, these tools have to be very accurate to become useful for developers. Too many false negatives are obviously bad because this means that developers can feel that those tools do not make a difference. On the other hand, too many false positives can lead to developers ignoring all warnings of possible configuration errors [46].

The current solution for system operators that have to deal with these configurations is to make use of automatic deployment tools. Instead of configuring heterogeneous systems manually, with configuration scripts, a specific configuration is generated for each machine. The available commands for those automations are very low level (e.g. "copy this file on this machine") which makes the resulting scripts usually very long and complicated [47].

Finding easy-to-use solutions for ensuring correct configurations becomes even more necessary today because cloud computing is very accessible even for less technical users.

2.4 Graphs

In this thesis, the dependencies of different configuration artifacts will be represented as a graph. Therefore, we use the following definitions for graphs and their properties by Valiente [48].

Definition 2.1. A *graph* $G = (V, E)$ consists of a finite nonempty set V of vertices and a finite set $E \subseteq V \times V$ of edges.

Definition 2.2. A *walk* from vertex v_i to vertex v_j in a graph is an alternating sequence

$$[v_i, e_{i+1}, v_{i+1}, e_{i+2}, \ldots, v_{j-1}, e_j, v_j]$$

of vertices and edges in the graph, such that $e_k = (v_{k-1}, v_k)$ for $k = i + 1, \ldots, j$. A *trail* is a walk with no repeated edges, and a *path* is a trail with no repeated vertices (except, possibly, the initial and final vertices). The *length* of a walk, trail, or path is the number of edges in the sequence.

Definition 2.3. Let $G = (V, E)$ be a graph, and let $W \subseteq V$. A graph (W, S) is a *subgraph* of G if $S \subseteq E$.

Definition 2.4. A graph $G = (V, E)$ is *undirected* if $(v, w) \in E$ implies $(w, v) \in E$, for all $v, w \in V$.

2.5 Related Work

Solving configuration problems became the subject of many researchers in the last years. An overview and categorization of the different kinds of configuration errors that can occur in software systems is given by Xu et al. [41]. Their work further discussed the cause of these errors and the difficulties that developers have to deal with when trying to prevent or fix configuration errors. Furthermore, they categorized approaches to solve or prevent those misconfigurations either by designing the configuration options of the systems itself or by additional tools and gave examples of implementations of those tools.

Work on analyzing what kinds of configuration errors occur and how relevant they are, is done by Yin et al. [45]. They analyzed 546 misconfigurations in CentOS, MySQL, Apache

HTTP Server, and OpenLDAP. In addition to classifications for the types of configuration errors, they looked into how systems communicate erroneous configurations and how helpful their error messages are for developers to fix the configurations. Their conclusion was, that a large number of those misconfigurations are very hard to debug.

Gunawi et al. [49] did similar work with a focus on the cloud systems Hadoop MapReduce, HDFS, HBase, Cassandra, and ZooKeeper. Over the course of a year, they build a database consisting of 3655 issues from bug trackers of these systems. They divided the configuration errors they found in two categories: The first are configurations that are invalid, often caused by updates of some parts of a system and the second are misconfigurations on parameters that lead to errors in combination with the configuration of other parameters.

Misconfigurations in the popular content management system Wordpress on LAMP stacks are studied by Sayagh et al. [50]. In these systems, configurations can be done in multiple places, such as the Linux operating system, the Apache web server, the MySQL database, PHP, the Wordpress application and its plugins. These configurations can set the same parameter in multiple places or indirectly depending on other parameters. They measured the direct and indirect usage of configuration options. They found out that the average number of configuration options of a Wordpress plugin in 87% and the majority of Wordpress's configurations are reused by at least two plugins and most configuration options. They concluded that multi-layer systems are very error-prone because often a lot of configuration options are used indirectly, which also makes the configurations of these systems hard to debug.

To solve configuration errors, Verbowski et al. [51] tried an approach where configuration changes are monitored and have to be approved by other developers with their software LiveOps. This is a program that takes the logs of different systems and relevant operating system logs like load or access to configuration files. Then it generates, depending how LiveOps is configured for that system, web reports, alerts or messages for other software.

Another approach for configuration management is integrated into Puppet by Vanbrabant et al. [52]. It interprets a subset of Puppet's configuration language to compute the operations to change a system's configuration by editing textual configuration files. They also used an authorization mechanism to prevent unprivileged developers from making uncontrolled critical configuration changes.

ConfValley by Huang et al. [53] is a framework for validating configurations in the deployment process. It can be used in the deployment process to ensure that configurations follow their specifications, written in their language CPL. They are also able to automatically gen-

erate tests from either constraints extracted from the source code or from configuration data. While testing their approach on Microsoft Azure, they also found 40 new configurations errors.

Sayagh et al. [7] presented another approach to find misconfigurations. They take an error message and then analyze the source code to build a dependency graph, starting from the error message. At the end, their algorithm outputs a list of possible incorrect configuration option. With their approach they were able to find the misconfigured options with a high accuracy in the first suggestions of their algorithm.

Hassan et al. [10] developed RUDSEA, a tool that analyses source code and configuration files to recommend updates to the related dockerfile. They evaluated their approach on 40 different projects, taken from Github. With RUDSEA they could recommend the correct location for an update with 78.5% and the correct update with 44.1%.

These approaches all have some problems in our opinion. Either they have a restriction on the toolset used for a project, for example they require to be deployed with Puppet, or new software has to be configured and learned, such as ConfValley or LiveOps. In contrast to that, we want to develop an approach, which is not specific to any tools and does not require developers to lean of a new software.

3 Configuration Networks

To model the configuration options and dependencies of software projects, we model those as graphs, called *configuration networks*. Each used software, configuration option, and software artifact is represented as a node in this network and is connected to all its dependencies. For example, a dockerfile is only useful if Docker is used, and the Docker instruction `FROM` can only occur in a dockerfile. Nodes representing a configuration option can have connections to other configuration options, that is they share a configuration with them.

Each software has to be implemented differently, depending on the possible configuration options, their meaning and the actual used options. After each change in the project we have to validate our graph again, include new software and exclude unused software. With this configuration network we can track which nodes and therefore which options in which software artifacts are affected by changes in a configuration option.

In the following, we present our terminology for different components of software projects and how we define configuration networks. Then, we will explain the construction of the initial configuration network of a project and how to use this network to find cross-component errors.

3.1 Network Definition

A configuration network is an undirected graph, built from the following types of nodes (see Figure 3.1): A root node, concept nodes, option nodes and *Shared Configuration Item* (SCI) nodes. The root node is followed by an arbitrary number of *concept* nodes which represent used software or more abstract concepts like source code, which has to be followed by a programming language specific subconcept. Their purpose is mainly to reach a granularity needed for certain configuration properties. Concept nodes can be followed by other concept nodes.

Files are modelled as *artifact* nodes and can occur multiple times in a concept. Their loca-

tions are attributes of the nodes. Artifact nodes are followed by *configuration options* which stand for commands or lines of code in the artifact that configure a setting of the project. They have the location (i.e. line number) in the artifact as an attribute. Options can have other options as children to model nested configuration files like XML files. SCI nodes are the leaf nodes of options and represent actual configuration parameters. SCI nodes can have edges to multiple options that is, all of these options configure the same setting in the project. Finally, options can have *attributes*. Each option has always the attribute *mutable*, which is set to `True` by default. For options of external services that cannot be changed by the developer team of the current software solution, this attribute can be set to `False`.

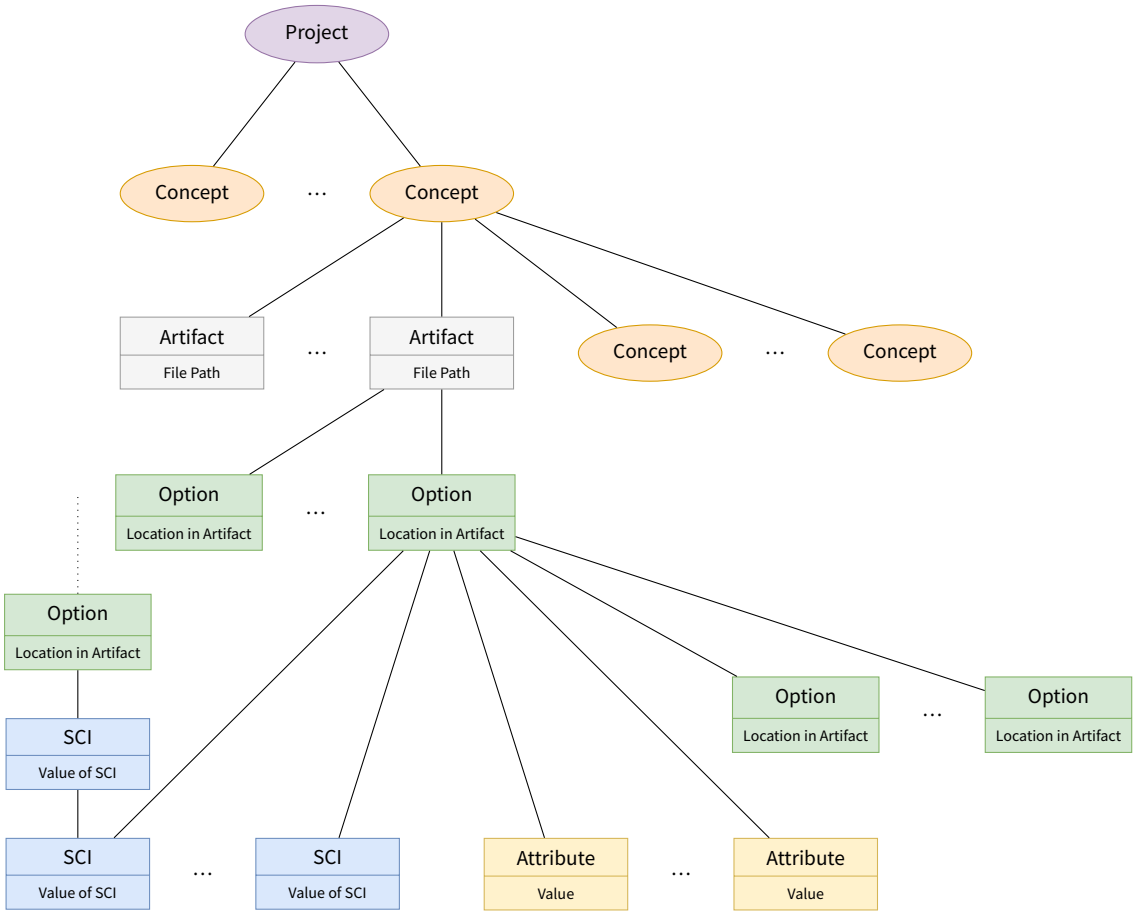


Figure 3.1: Types of nodes and their hierarchy in configuration networks.

Concepts have a static network which defines all possible artifacts, options, SCIs, and sub-concepts for a technology. We also have to define the cues, with which we can see that a concept is used. To construct a configuration network for a project we parse its VCS repository and identify used concepts. After that, we use the concept's static network to see which artifacts belong to the concept. From those artifacts and the static network, we can deduce

the used options and the values of the SCIs.

Next, we explain how to build the configuration network, using an exemplary microservice. For our running example, we assume the following file structure and the use of an external Eureka instance to realize service discovery:

```
.
├── src
│   ├── a.java
│   └── b.java
└── dockerfile
```

As a first step to create the configuration network, we define a root node. Then, we have to extract the *components* that exist in the software solution. As for now, this extraction process is manual, but should be automated in the future. One concept of our example is the source code. Based on the files we find, we know that we have Java code. We model both as *concepts*. Then we add artifact nodes for the two source files `a.java` and `b.java` (see Figure 3.2).

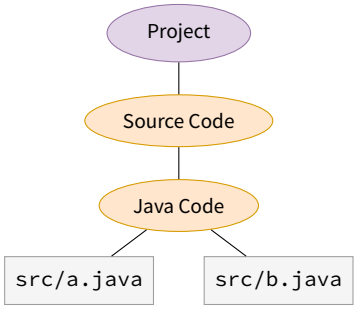


Figure 3.2: Example microservice with source code files.

The file `a.java` contains the definition of the port this microservice listens to and `b.java` the location of the service discovery. This is done by setting variables in the source code, which get modeled as options. The actual values for the port and the service discovery location are added as SCIs to the network (see Figure 3.3).

Because the microservice is deployed via Docker, the VCS repository contains a `dockerfile` (see Figure 3.4a). The `dockerfile` exposes the port of the microservice. This is the same port as specified in `a.java`. Therefore, we can connect both port SCI nodes to represent that they can not be changed independently (see Figure 3.4b).

The service discovery depends on an external instance of Eureka. We assume that the configuration of the used Eureka instance is given by some file artifact available to us. This ar-

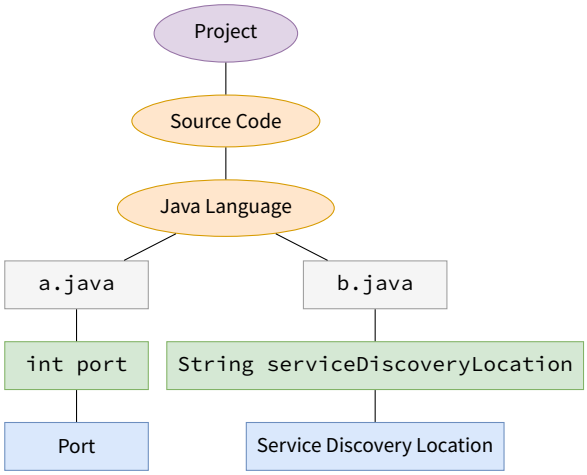


Figure 3.3: Example microservice with added properties and configuration values.

tifact contains the definition of the location URL. Since this configuration may be readable by us, but not modifiable, we set the *Eureka URL* node's *mutable* attribute to `False` (see Figure 3.5).

When a developer changes the port in `a.java`, we can check which other option nodes are connected to the port SCI to get the locations in the other artifacts where the developer has to make changes. In this case, it would require to adjust the `EXPOSE` command in the `dockerfile`.

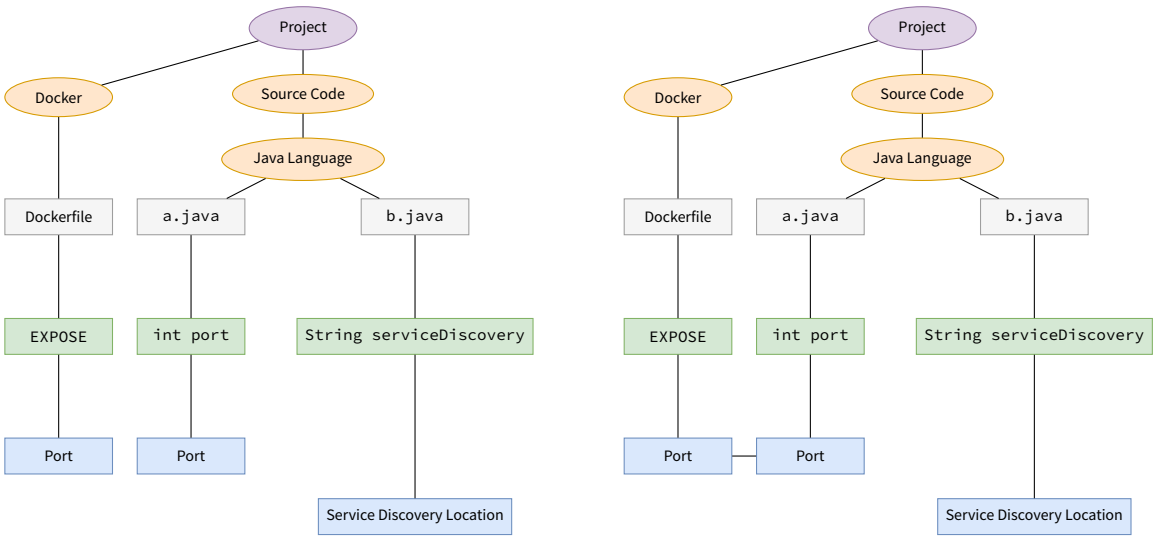
To find the exact locations in the artifacts where the developer has to make the changes, we stored these locations of the options in their corresponding artifact as attributes of option nodes.

When an artifact is changed we can look at its underlying concept nodes to see whether they match the changed locations. To find out the exact location where changes are required, we can follow the paths from the Port SCI node over the option nodes to the artifact nodes (see Figure 3.6).

If a developer would change the URL of the service discovery in `b.java`, we see that this would affect the external Eureka node. Since this is not in the scope of the current microservice, we want to give a warning that this configuration probably breaks the service.

To summarize the example, when a change in a SCI occurs, we have to follow all paths from that SCI until we reach artifact and external nodes. Artifact nodes have to be changed, external nodes cannot be changed. Hence, we have to warn the developer.

To make use of this configuration network, we need the following:



(a) Multiple instances of the port SCI. (b) `EXPOSE` and `int port` are connected.

Figure 3.4: Example microservice with source code and docker.

1. A Database of static configuration networks for different concepts, either build manually or automatically.
2. A network initialization tool, that parses a project for its used concepts and then constructs the initial configuration network for this specific project. Concepts should be implemented as plugins, to allow an easy integration of new concepts.
3. On changes in the source code, we need a tool that tracks these changes in the configuration network to find possible errors
4. When detecting a conflict, we want to recommend solutions.

In the following we will construct prototypes of parsers of different concepts to build a project's initial configuration network and focus on detecting changes and resolving errors.

3.2 Network Initialization and Update

For the initial construction, we checkout the current version of the master branch from a VCS. We assume that this initial configuration is correct. This allows us to parse the source code for occurrences of existing SCIs (from other concepts) and connect those to the corresponding option nodes later instead of having to validate the initial network first. Based on the detected files, we can deduce some used technologies, such as the existence of configu-

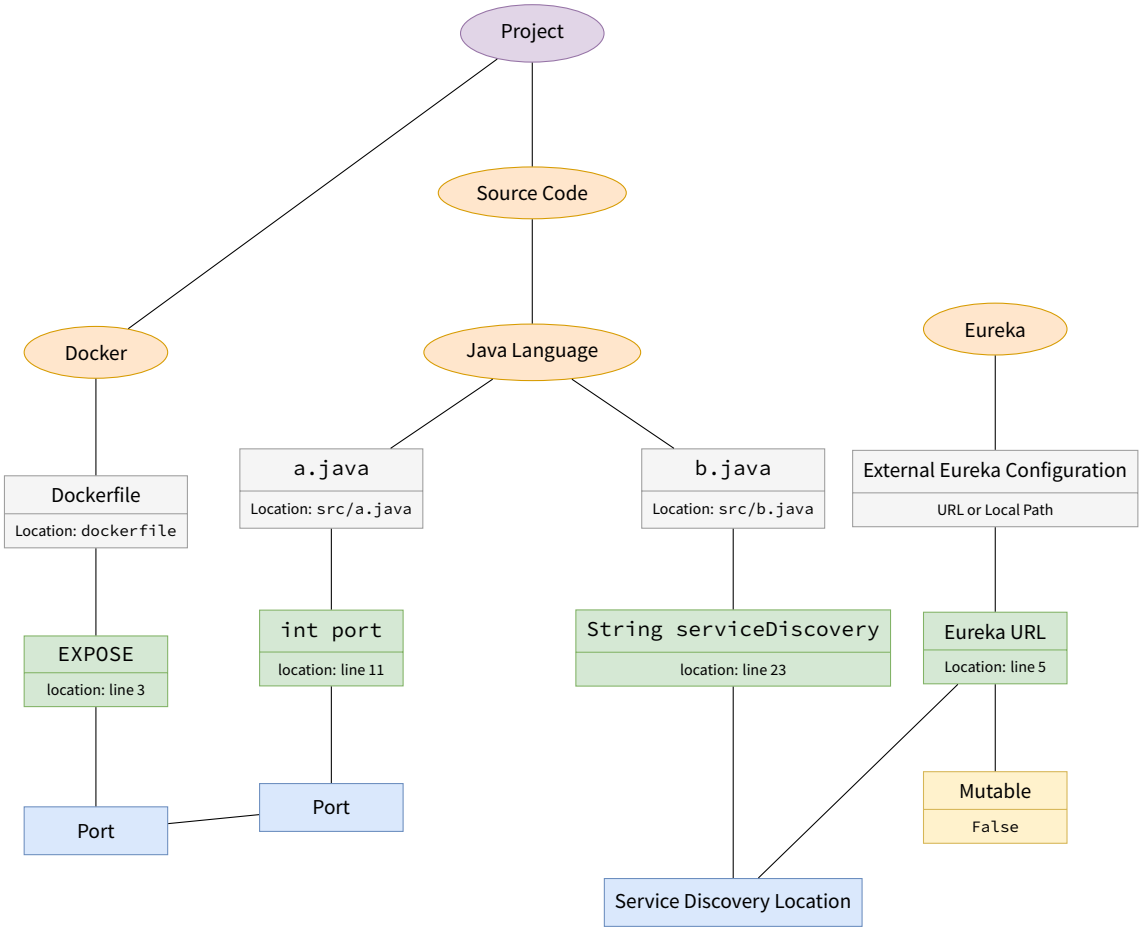


Figure 3.5: Example network with Eureka and additional attributes for nodes.

ration files for build systems or Dockerfiles. This gives us the first SCIs. From the source files, we can easily get the used programming language. When we analyze these files further, we can obtain additional information. Include statements in source files or build system configurations give us more information about the used libraries and external services. For example, if we find the element in Listing 2 in the `pom.xml`, we know that Eureka is used (and can further deduce that *Java Development Kit* (JDK) 1.8 or higher is used, since this is a requirement). Furthermore, annotations in Java files provide concrete configuration commands and can be grasped to further create the configuration network.

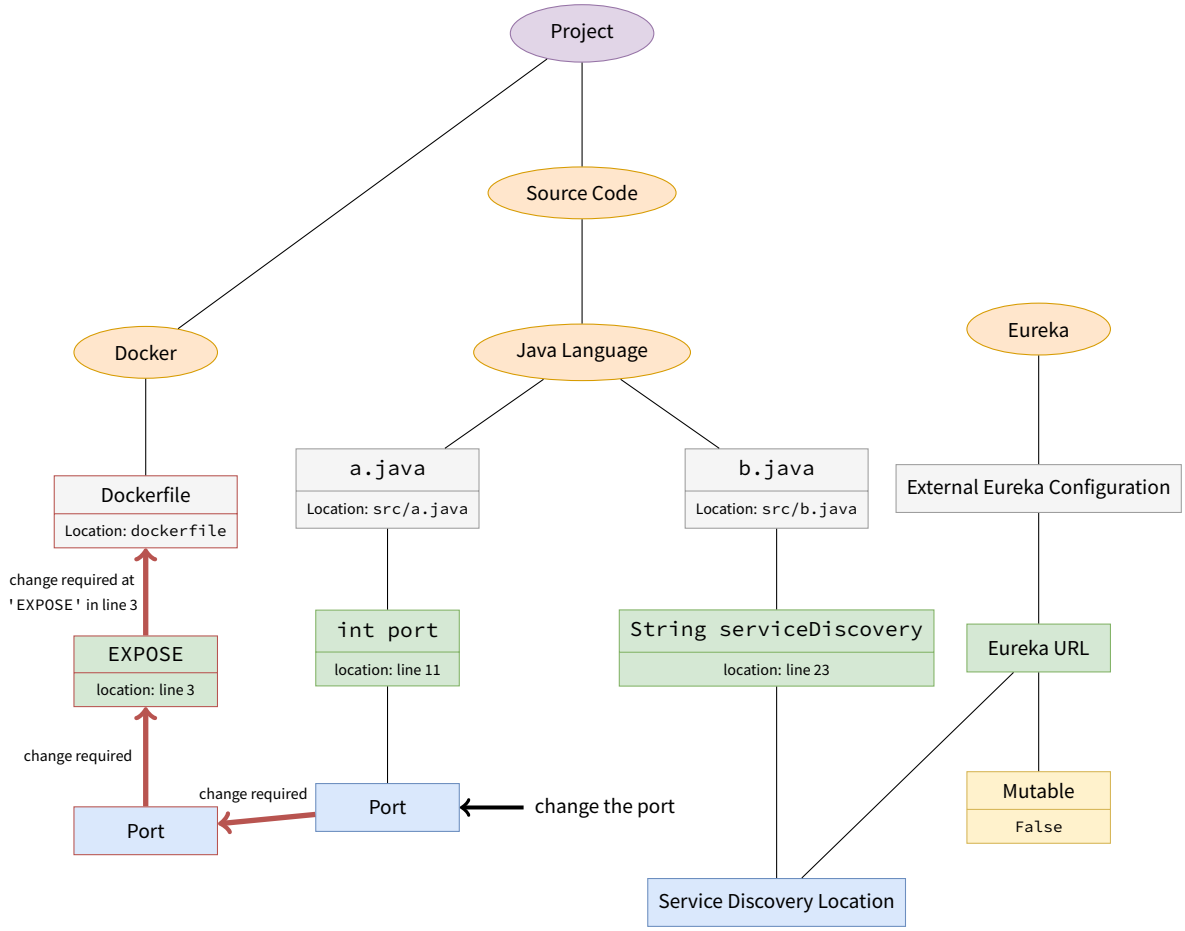


Figure 3.6: Impact of changing the port.

```
<dependency>
    <groupId>com.netflix.eureka</groupId>
    <artifactId>eureka-client</artifactId>
    <version>1.1.16</version>
</dependency>
```

Listing 2: Including Eureka with Maven.

The properties of external services such as service discovery have to be included in some way. A solution would be a repository where services can export the values that are visible and/or relevant to other services.

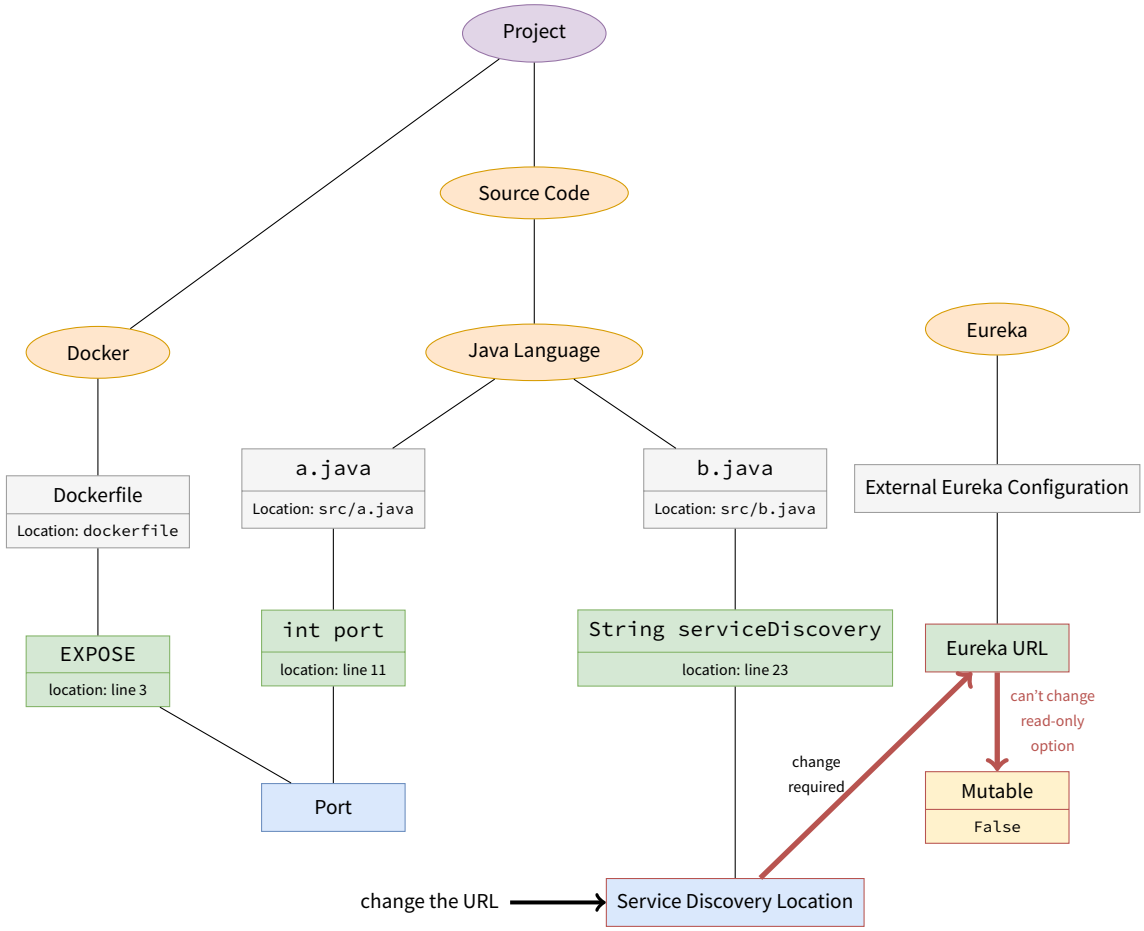


Figure 3.7: Change of service discovery URL in the service.

3.2.1 Example

To implement our configuration network application prototype, we need a real-world example of a microservice environment. We chose Kenny Bastani's example project of a microservice environment [54] because it has a variety of services in a very minimal configuration.

We will focus on Bastani's implementation of an Eureka server instance, which can be found in `discovery-microservice/`. As shown in Listing 3, its source code consists only of a minimal main function, three imports and two Spring Boot annotations. Since there is almost no source code, the most probable errors that can occur are configuration errors.

The service's directory also consists of a `pom.xml` and a `dockerfile`. Therefore, when constructing the configuration network we will get three top-level concepts below the root project node: Maven, Source Code and Docker (see Figure 3.8). In the following we will

```
package services;

import ... SpringApplication;
import ... SpringBootApplication;
import ... EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Listing 3: Source code for the Eureka instance.

discuss these three subgraphs to construct the complete configuration network.

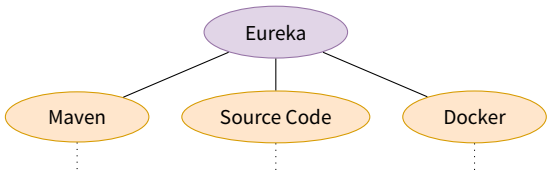


Figure 3.8: Eureka microservice root and top-level concepts.

First, we analyze the Maven concept (see Figure 3.9). For now, we will skip the build part, which consists of plugins used for building the service. The important parts for the construction of our network are the `artifactId`, the `packaging`, and the `version`. From the combination of those three configuration items, we can directly deduce the binary name, which will be `<artifactId>-<version>.<packaging>`.

Then, we analyze the Docker concept (see Figure 3.10). It consists of a single artifact node for the `dockerfile`, containing option nodes for the occurring Docker instructions. The internal port and the protocol are implicitly defined by omitting their values in the `EXPOSE` instruction. It has some internal dependencies between some of the instructions. For example, the `ADD` instruction copies the binary of the service into the container as `app.jar`. Therefore, all following instructions will refer to this name of the microservice.

Now, we can create edges for dependencies between Docker and Maven. As mentioned earlier, the `pom.xml` gives us information about the binary name, which will be used in the

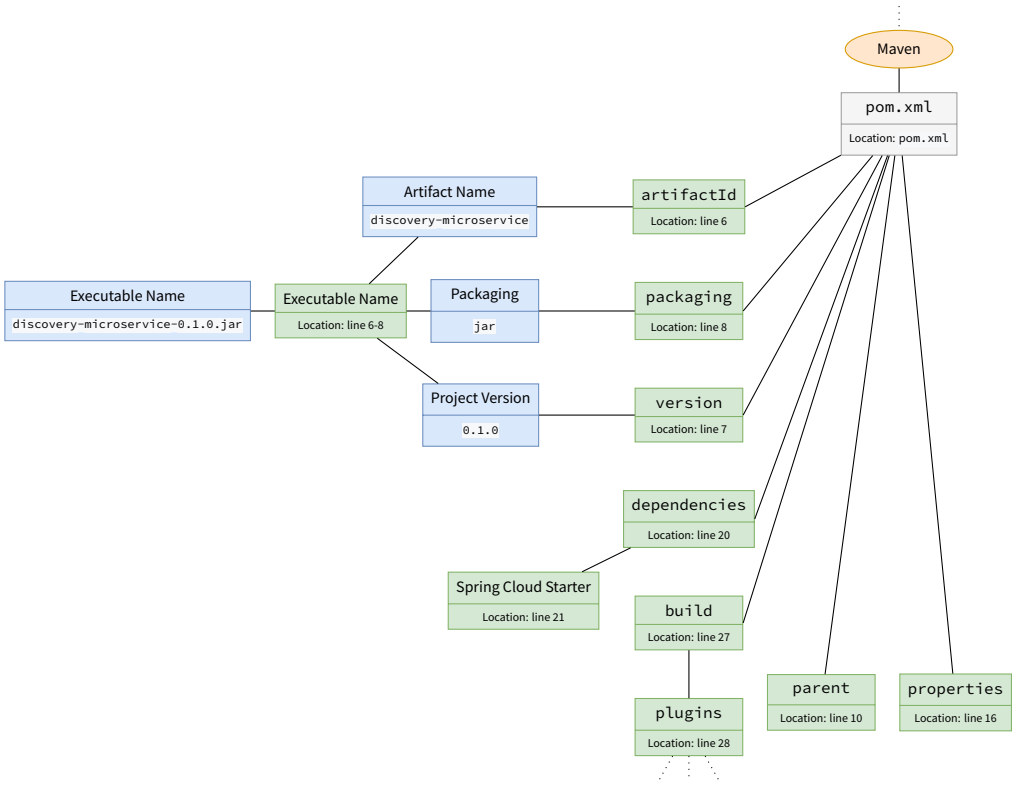


Figure 3.9: Maven subnetwork for Eureka microservice.

`dockerfile`'s `FROM` instruction. Additionally, the second parameter of `ENTRYPOINT` depends on the packaging, defined in the Maven build file.

The Source Code concept has a subconcept for the Spring library. The imports and annotationsset some options implicitly, which we skip here.

Spring applications can be configured in the `application.yml` and `bootstrap.yml`. That is, the inclusion of the Spring libraries adds dependencies in our network from the Spring imports and annotations to the options of the configuration artifacts.

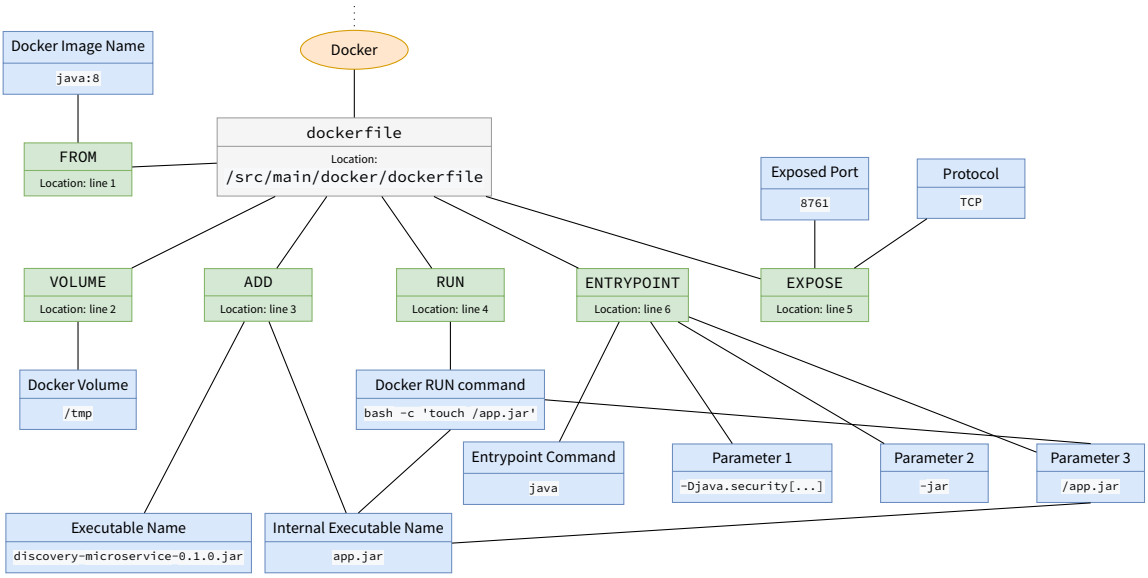


Figure 3.10: Docker subnetwork for Eureka microservice with internal dependencies.

3.3 Implementation

To evaluate the practicality of configuration networks, we build a prototype in the Python 3 programming language. We chose Python 3, because it is a very popular programming language, which means that a lot of useful libraries are available. In specific, we used the `networkx` library to construct our graphs. It also provides functions to draw the graphs with `matplotlib`. Python is also reasonably fast since most libraries use C-functions for performance critical code.

3.3.1 Network Construction

This prototype processes a Git repository and builds a network of root, concept, artifact, option and SCI nodes from the tracked file blobs of the `HEAD` commit. To construct the network, the prototype searches the blobs for artifacts that denote the existence of a concept, for example a `dockerfile` for Docker or a `pom.xml` for Maven. These configuration files are parsed to extract the corresponding subnetwork of option and SCI nodes. Then we search for the occurrence of configuration options that lead to other concepts, for example a dependency element in the `pom.xml` (see Listing 4) and add other concept nodes to the root.

```
<dependency>
    <groupId>com.netflix.zuul</groupId>
    <artifactId>zuul-core</artifactId>
    <version>2.1.2</version>
</dependency>
```

Listing 4: Include Zuul with Maven [55]

For composed values, such as the executable name that depends on the values of `artifactId`, `version` and `packaging` elements of the `pom.xml`, additional SCI nodes are added to the network and connected to the SCI nodes of the values that compose the value of the new node (see Figure 3.11).

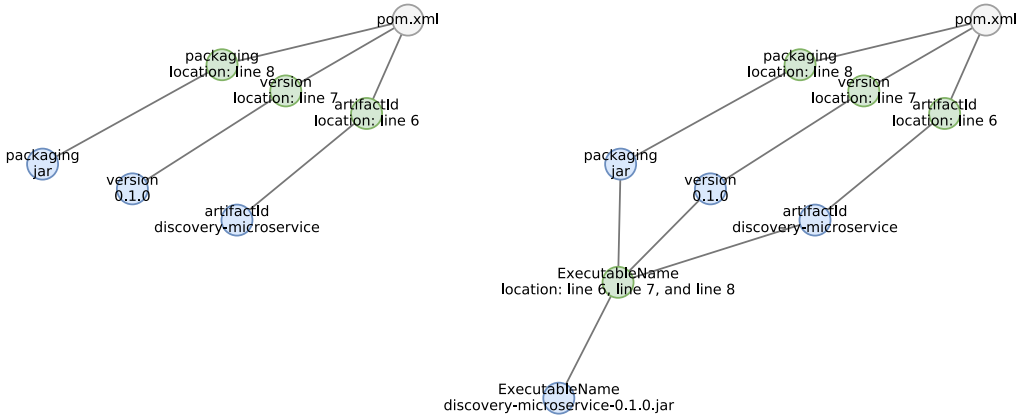


Figure 3.11: Composing executable name from artifactId, version and packaging SCI nodes of a pom.xml artifact node of the Maven concept.

After each artifact is converted to nodes, depending on the concept and file, links between SCI nodes that have the same value are made.

Finally, all unprocessed concepts, e.g. those without child nodes, are processed and further links between SCI nodes are added.

3.3.2 Conflict Detection

To detect conflicts in the configuration we use the algorithm shown in Listing 5. First, the implementation builds a new network with the current status of the blobs in the Git repository. Then it iterates over all leaf option nodes of the old network. We call an option node a leaf option node if its children are SCI nodes, that do not have an option node as their child,

```
function DETECT_CONFLICTS(old_network, project_root)
    new_network ← INITIALIZE_NETWORK(project_root)
    options ← GET_LEAF_OPTIONS(old_network)
    for option in options do
        conf_items ← GET_SCI(option)
        for sci in conf_items do
            if sci ∈ new_network then
                CHECK_NEIGHBORS(sci, new_network)
            else
                WARNING("SCI was removed from the network.")
            end if
        end for
    end for
end function
```

```
function CHECK_NEIGHBORS(sci, new_network)
    neighbors ← GET_SCI_NEIGHBORS(sci)
    for neighbor in neighbors do
        if neighbor ∈ new_network then
            new_neighbor ← neighbor ∈ new_network
            if not ARE_CONNECTED(new_neighbor, sci) then
                ERROR("Configuration conflict found.")
            end if
        else
            WARNING("Neighbor was removed from the network.")
        end if
    end for
end function
```

Listing 5: Detecting changes in the configuration network.

e.g., are not composed to other SCI nodes. We do not need the non-leaf options, since their configuration values are represented in the composed nodes. If the node was not in the old network, we can skip it since it cannot have any conflicts in the new network and we will assume it is correctly configured. For nodes that are found in the old network, we check whether all SCI neighbors in the old network still exist and if they are still connected to the node. When both conditions are satisfied, no configuration conflict occurred.

We use a subnetwork of our example service, consisting of a Maven and a docker concept to explain the implementation of the conflict detection (see Figure 3.12).

In our example, the leaf option nodes are the option nodes ExecutableName, ADD, RUN, and

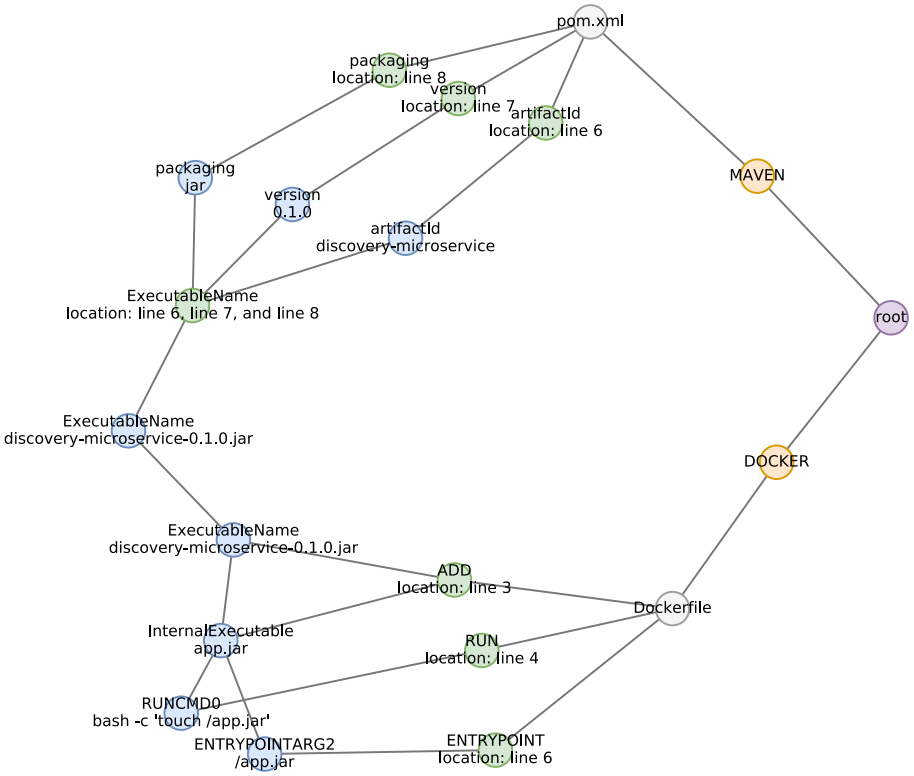


Figure 3.12: Initialized example subnetwork

ENTRYPOINT. For each SCI child node of an option, we search the new network for the corresponding node. We identify nodes by their names which are constructed of the node type, the parent artifact, the path of options and the name of the SCI. For example, the InternalExecutable node is named `SCITEM::Dockerfile::ADD::InternalExecutable`.

This would be the case if, for example, a developer changes the `version` element in the `pom.xml` from `0.1.0` to `0.2.0` and applies the same change to the `ADD` instruction in the `Dockerfile`. After the construction of the new network (see Figure 3.13), there are no new SCI nodes and all SCI nodes have the same neighbours. The change of the values of the two ExecutableName nodes has no effect on the network since both get reconnected. This results in a correct configuration, therefore we do not need to display any errors or warnings.

If a neighbor was removed from the network, we print an according warning message. In our example this could be due to a developer removing the `ADD` instruction entirely from the `Dockerfile` (see Figure 3.14). This does not necessarily mean that the configuration is wrong, but needs the review of the developer in case the option was removed by mistake.

If the neighbor does exist, but is not connected anymore, we print an error message, stat-

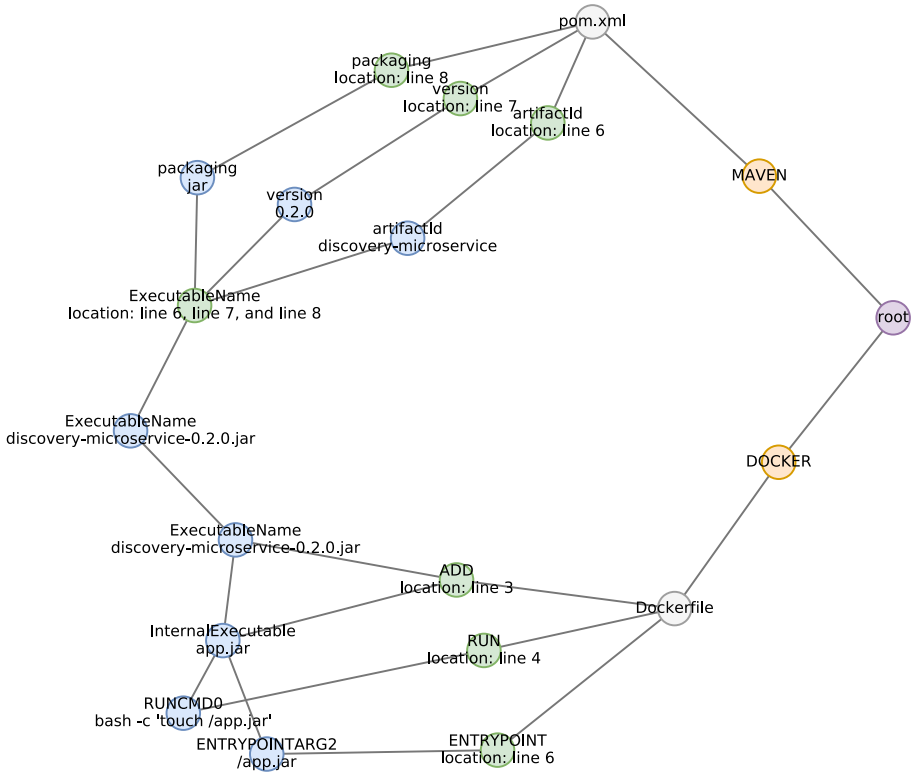


Figure 3.13: No conflict in the new network

```
#!/bin/sh

detect_network_changes .configuration_network
```

Listing 6: Using our implementation in a simple Git pre-commit hook.

ing that the values of those options have to be changed such that they comply with each other. For example when a developer changes the `version` element in the `pom.xml` from `0.1.0` to `0.2.0` like before, but does not applies the same change to the `ADD` instruction in the `Dockerfile` (see Figure 3.15). We can look for the parent artifact node and the parent option node to find the file and line number that have to be changed to resolve this conflict. In this case the developer needs to change line 3 of the Dockerfile to `ADD discovery-microservice-0.2.0.jar app.jar`.

Our `detect_network_changes` script returns with the exit code 1, Git will abort the `git commit` command, when we detect a configuration conflict. This means that is easy to integrate in existing development pipelines, because it can be used in a simple one-liner Git pre-commit hook (see Listing 6).

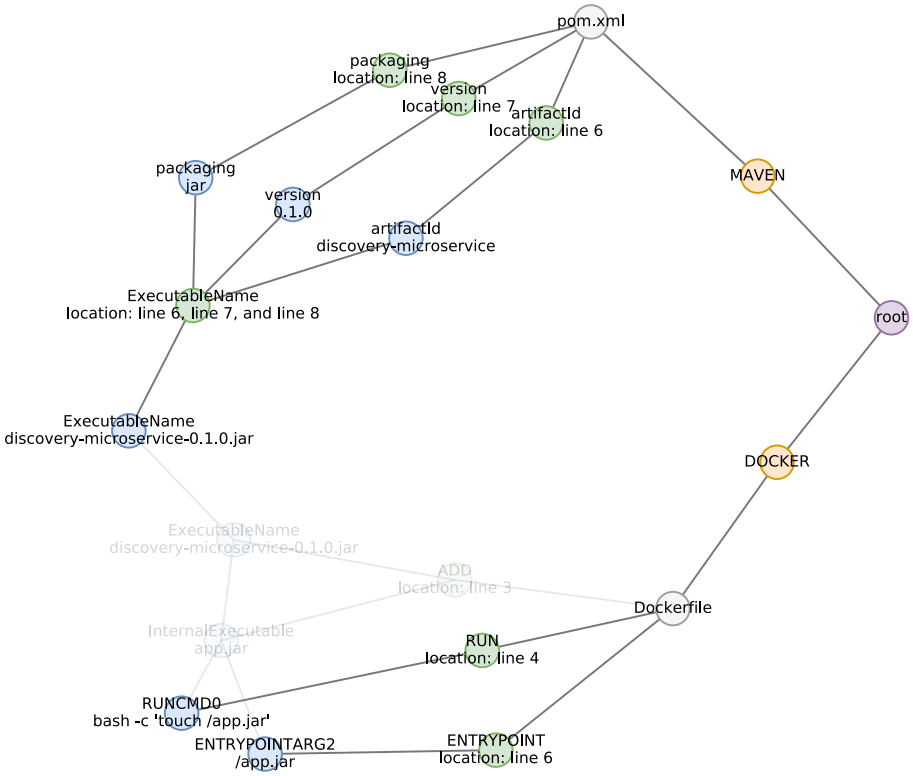


Figure 3.14: Removing the ADD instruction produces a warning

We also used the `networkx` library to plot the complete configuration network with `matplotlib` (see Figure 3.16).

The current prototype only supports files in a Git repository, therefore we cannot track external configuration files. Because of this, attribute nodes are also unimplemented since their main purpose is to mark external nodes as read-only.

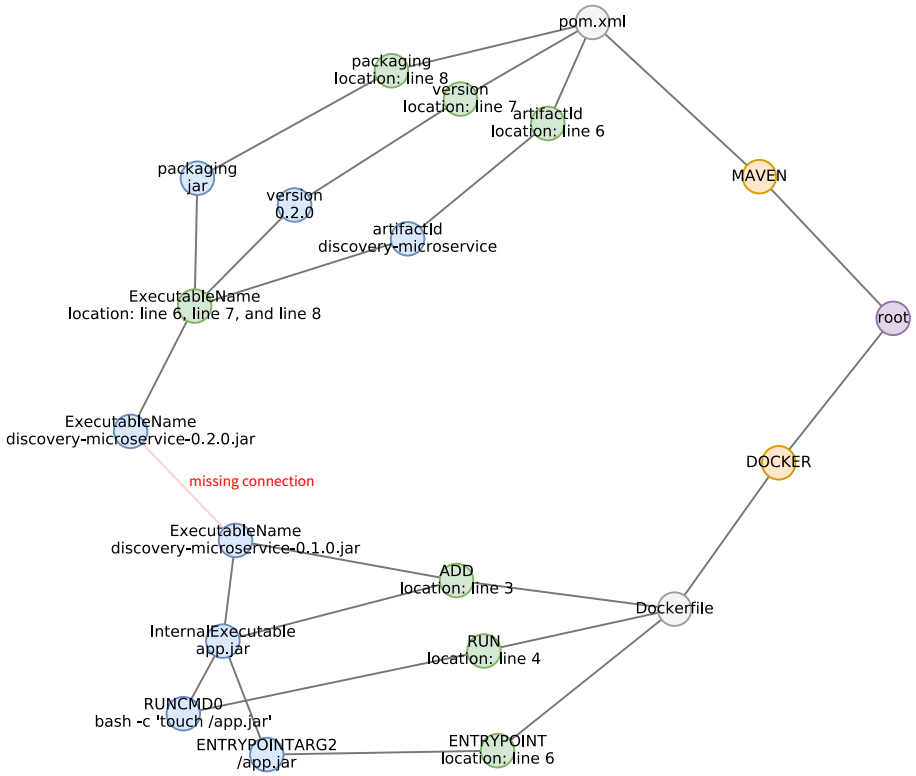


Figure 3.15: The SCI nodes of ExecutableName and ADD are not connected anymore

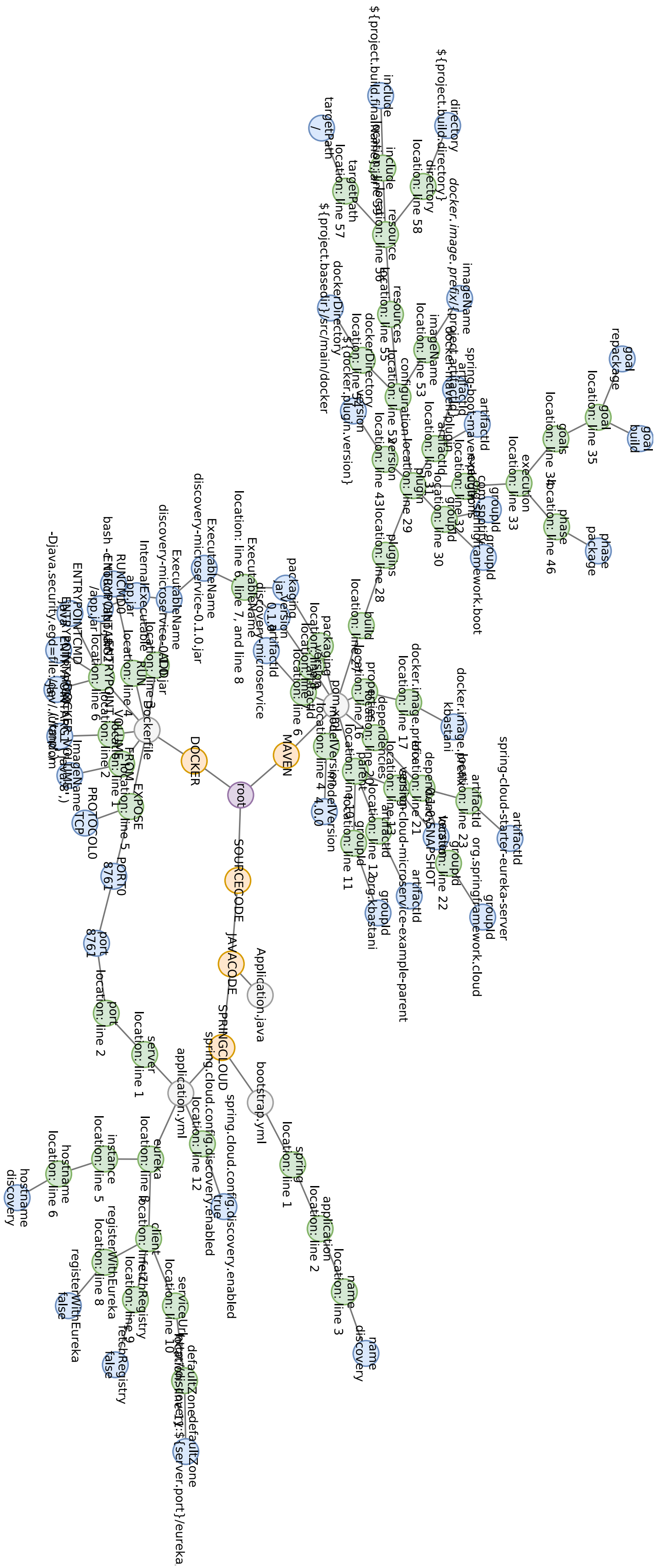


Figure 3.16: Plot of an example project.

4 Evaluation

The goal of the evaluation is twofold: First, we want to determine whether we find configuration errors of known linked configuration items. Second, we are interested whether we find configuration errors in the wild, that is, in a real-world software project.

We evaluated both projects on a machine with 32 GB RAM, an Intel Core i7-7700K 4.20 GHz CPU, running a 64bit Linux distribution.

4.1 Artificial Configuration Errors

In our first experiment we aim at evaluating whether we can detect configuration conflicts in a controlled setting. For this project, each of our implemented concepts and configuration files parsers have to be used for the network construction. It consists of a single Java file where a Eureka discovery service is implemented with Spring libraries. The Spring components are configured in `application.yml` and `bootstrap.yml`. To build the service, a `pom.xml` is provided. This also defines the dependencies and the name of the service. The repository also contains a `Dockerfile` for the deployment with Docker.

4.1.1 Setup

We checked out the current master branch (`5e8dfa...`) of Bastani's microservice demonstration [54] and copied the `discovery-service` folder in a new Git repository and initialized the configuration network with

```
$ initialize_network .
```

Then, we introduced several changes (as described below) and checked the configuration network with

```
$ detect_network_changes .configuration_network
```

We then introduced the following changes in the project:

1st change (conflict between Docker and Spring)

The port on which a service is reachable is one of its basic configurations. This port has to be defined at different places and these configurations have to be consistent. We changed the value of the `EXPOSE` instruction from 8761 to 8762 in `src/main/docker/Dockerfile`.

Expected result:

- Warning that the `EXPOSE` option connected to the value `8761` was removed from the network.
- Error for a conflict of the `EXPOSE` instruction with the `port` value `8761` in `src/main/resources/application.yml`.

2nd change (fixes conflict from 1st change)

To fix the error from the first change, we we changed the value of the `port` in `src/main/resources/application.yml` from 8761 to 8762.

Expected result:

- No errors or warnings.

3rd change (conflict between Maven and Docker)

The version of a binary can get updated multiple times during development. We changed the `version` element in `pom.xml` to `0.2.0`. This changes the compiled binary name and therefore the option node for the executable name but not at the places where the binary is reused in other concepts.

Expected result:

- Warning that the executable name option node connected to SCI `discovery-microservice-0.1.0.jar` was removed from the network.
- Error that the `ADD` instruction has to be updated according to the value `discovery-microservice-0.2.0.jar` of the executable name node.

4th change (fixes conflict of 3rd change)

To fix the conflict from the third change, we updated the first parameter of the `ADD` instruction in `src/main/docker/Dockerfile` to `discovery-microservice-0.2.0.jar`.

Expected result:

- No errors or warnings.

5th change (Docker internal conflict)

To check the dependencies that Docker instructions can have to each other, we changed the second parameter of `ADD` in `src/main/docker/Dockerfile` to `service.jar`. With this change we renamed the executable in the Docker container to `service.jar`. The old internal name still gets used by the `RUN` and `ENTRYPOINT` instructions.

Expected results:

- Warning that the `ADD` node connected to value `app.jar` was removed from the network.
- Error that the `RUN` instruction in the Dockerfile has to be updated according to the value `service.jar` of the `ADD` instruction.
- Error that the `ENTRYPOINT` instruction in the Dockerfile has to be updated according to the value `service.jar` of the `ADD` instruction.

6th change (fixes conflicts of 5th change)

We changed the parameter of `RUN` in `src/main/docker/Dockerfile` to `bash -c 'touch /service.jar'` and the parameter of `ENTRYPOINT` to `/service.jar`.

Expected result:

- No errors or warnings.

7th change (swap two Docker instructions)

Often, the order of Docker instructions does not matter. To test this, we swapped the positions of `EXPOSE` and `RUN` in `src/main/docker/Dockerfile`.

Expected result:

- No errors or warnings. because this should not change the network.

8th change (adding new Docker instruction)

During development of a service, many new configurations will get added. We added `EXPOSE 1234` to `src/main/docker/Dockerfile`. This adds a new node that has no dependencies to other nodes and therefore cannot produce any conflicts.

Expected result:

- No errors or warnings.

The first of these changes aim at producing conflicts in the configuration network by changing only one of two connected SCI nodes. We did this with the dependencies of three different combinations of concepts. After each of those changes, we introduced additional changes to fix these conflicts. Finally, we also introduced two changes which should not produce any conflicts.

4.1.2 Results

For our synthetic experiment, we were able to produce the expected output as follows:

1st change (conflict between Docker and Spring)

```
WARNING: Option src/main/docker/Dockerfile::EXPOSE == 8761 was
 ↪  removed from src/main/docker/Dockerfile. Previous location was
 ↪  in line line 5.
ERROR: File src/main/resources/application.yml, (location: line 2):
 ↪  Value of option
 ↪  `src/main/resources/application.yml::server::port` (8761) has to
 ↪  be changed to fit value `8762` of option
 ↪  `src/main/docker/Dockerfile::EXPOSE` (in line 5 of
 ↪  src/main/docker/Dockerfile).
```

2nd change (fixes conflict on 1st change)

No errors or warnings.

3rd change (conflict between Maven and Docker)

Result:

```
WARNING: Option pom.xml::artifactId::artifactId::ExecutableName ==
 ↪  discovery-microservice-0.1.0.jar was removed from pom.xml.
 ↪  Previous location was in line line 6, line 7, and line 8.
ERROR: File src/main/docker/Dockerfile, (location: line 3): Value of
 ↪  option `src/main/docker/Dockerfile::ADD`
 ↪  (discovery-microservice-0.1.0.jar) has to be changed to fit
 ↪  value `discovery-microservice-0.2.0.jar` of option
 ↪  `pom.xml::artifactId::artifactId::ExecutableName` (in line 6,
 ↪  line 7, and line 8 of pom.xml).
```

4th change (fixes conflict of 3rd change)

No errors or warnings.

5th change (Docker internal conflict)

Result:

```
WARNING: Option src/main/docker/Dockerfile::ADD == app.jar was
 ↪  removed from src/main/docker/Dockerfile. Previous location was
 ↪  in line line 3.
ERROR: File src/main/docker/Dockerfile, (location: line 4): Value of
 ↪  option `src/main/docker/Dockerfile::RUN` (bash -c 'touch
 ↪  /app.jar') has to be changed to fit value `service.jar` of
 ↪  option `src/main/docker/Dockerfile::ADD` (in line 3 of
 ↪  src/main/docker/Dockerfile).
ERROR: File src/main/docker/Dockerfile, (location: line 6): Value of
 ↪  option `src/main/docker/Dockerfile::ENTRYPOINT` (/app.jar) has
 ↪  to be changed to fit value `service.jar` of option
 ↪  `src/main/docker/Dockerfile::ADD` (in line 3 of
 ↪  src/main/docker/Dockerfile).
```

6th change (fixes conflicts of 5th change)

No errors or warnings.

7th change (swap two Docker instructions)

No errors or warnings.

8th change (adding new Docker instruction)

No errors or warnings.

Summary

In summary, we could verify that our network is capable of detecting all configuration errors as expected. The time needed for the analysis is marginal considering the network size. Hence, we conclude that our technique can be useful in practice when we complement the network with additional links of configuration options.

4.2 Real-World Project

To determine whether our prototype can detect real-world configuration problems, we took Jonas Hecht's cxf-spring-cloud-netflix-docker [56]. We also wanted to know what additional misconfiguration types have to be checked in future version of our configuration network approach. The original purpose of the project was to show how to combine Spring Boot applications with Spring Cloud, Docker, and microservice-related tools from Netflix. We chose this project because it uses common technologies, which can give us a good feedback in how our prototype performs in the real world.

4.2.1 Setup

We checked out the master branch of the project and used a script (see Listing 7) to go through all commits of the master branch. At each commit, we called our conflict-detection mechanism, printed the output and measured the time it took, and then went to the next commit. The network we created consists of about 3,800 nodes.

4.2.2 Results and Discussion

Unfortunatly, our prototype produced only some warnings, but we could not find any configuration conflicts in this project. We asked the developer of the project in a semi-structured

```
#!/bin/sh

# Traverses through the master branch of a Git repository and
# tries to detect configuration errors.

GITCOMMAND='git checkout'
INITIALIZE='time(initialize_network .)'
DETECT='time(detect_network_changes .configuration_network)'

eval 'git checkout master'

COMMITS=$(git log --reverse master --pretty=format:"%H")

FIRST=true
for c in $COMMITS
do
  echo ''
  eval "$GITCOMMAND $c"
    echo "git commit $c"
  if [ "$FIRST" = true ]
  then
    echo "initializing network..."
    eval "$INITIALIZE"
    FIRST=false
  else
    echo "detecting conflicts..."
    eval "$DETECT"
    echo "updating network..."
    eval "$INITIALIZE"
  fi
done
```

Listing 7: Script for checking every commit of the master branch for configuration conflicts.

interview about the correctness of our findings. He confirmed that these types of errors are not present in this project. Interestingly, the developer added that the demo projects, such as the project we chose, and highly stared projects on Github are unlikely to contain such errors. Their purpose is to show best practices and proof of concepts, therefore they are often very simplistic and well tested.

The errors that we can find with our current prototype can often be found by testing. For example, if a port is wrongly configured, the service does not work. Nevertheless, the developer believes that our approach might find the errors faster since we do not have to start the whole microservice environment. Moreover, when no or insufficient tests are present or performed, our approach still works. Additionally, he confirmed that our tool can be useful for real-world projects with average developers, because in his daily work as an IT consultant he encounters a lot of similar errors at real-world projects.

On the first commits, the error detection and network update took less than 0.5 seconds, on later commits where the network contained 3,000 nodes and more the detection took more than 2.3 seconds. On bigger projects or when more concepts are implemented and therefore more nodes are added to the graph, we expect an even higher processing time, making the integration as a pre-commit hook not yet ready without further optimizations.

4.2.3 Threats to Validity

In the following, we will discuss the threats to the validity of our evaluation, based on the implementation of our prototype and our chosen test projects.

Internal Validity

For the first project, we used only manually added configuration changes. These are the kinds of changes we thought of when implementing our concept parsers for the prototype. There might be other projects using the same concepts but with more sophisticated options that require changing our algorithms because they are impossible to implement otherwise.

External Validity

We only used one real-world project to test our approach in detail. This service could be not representative for the majority of microservice projects. On the contrary, when comparing

the second project with other highly starred repositories one can see that these often have a very similar structure. Also, the interview with the developer confirms the validity ouf our approach.

5 Conclusion

We developed a graph-based approach for detecting cross-component configuration errors and recommending solutions, called configuration networks. Once these configuration networks are built, the conflict detection mechanism can find conflicts by comparing two networks while being agnostic of the details of the used technologies' configuration options.

While we cannot solve any real-word configuration problems yet, our experiments as well as the interview with a developer show that our configuration network approach is feasible in theory and has already shown promising results. If technology that is used in microservices in production is translated to a network and connections between nodes that share a configuration can be made, we are very optimistic that our approach will work when more concepts are implemented.

Our current prototype is able to solve some basic configuration conflicts where the same property is is some way directly shared between two artifacts. This can be improved in the future by implementing more concepts and parsers for their corresponding configuration files. More complex problems such as database connection limit and PHP timeout as mentioned in the introduction of this thesis, where configuration files are not available in the same Git repository and affect each other only indirectly, are not considered yet.

5.1 Future Work

This section outlines directions for future work, such as which problems of the prototype have to be fixed and which features should be implemented.

5.1.1 File Renaming

In the current implementation we are not able to track the renaming of files. The renaming of an artifact results in changed names of all following nodes, such that we cannot find them

on the conflict detection mechanism. Since we work on Git repositories, we should be able to get the necessary renaming informations from Git.

5.1.2 Extending the Implementation

The configuration network implementation has to be extended with parser for more concepts and the existing parsers have to be extended to detect more connections between SCI nodes.

5.1.3 Adding New Incorrect Options

When an invalid configuration node is added to the network, we cannot detect this as an error. Suppose, there is a microservice project with only one service A. Later, the developer adds another service B and copies the `Dockerfile` from service A which contains `ADD target/serviceA.jar` and does not adjust this line for the new `Dockerfile` for service B. When constructing the new network, no conflicts are detected and a new connection between the `ADD` instructions of both `Dockerfile`s are made. Then, if a developer corrects this mistake, for example by changing `ADD target/serviceA.jar` to `ADD target/serviceB.jar` our network identifies this as a conflict. While this is the correct behaviour for our network, it is not a real error. We have to find a mechanism to improve this behaviour.

5.1.4 Performance

When we tried to test our prototype with bigger projects, we got configuration networks with 3000 nodes or more. The initialization of the network and the detection of conflicts then each takes a few seconds. When we want to use our approach in production, for example as a Git pre-commit hook, it is preferable to have a run time of 0.5 seconds or less, to not interrupt the developer. The visualization of the networks takes several minutes and if we have more than about 3500 nodes `matplotlib` can take over an hour to compute a layout for the network and then is still not able to draw it.

To tackle these problems, we have to update our implementation. We can look into more performant programming languages such as C++ or Rust. Additionally, we can replace our data structure for our network with a graph-optimized database, such as Neo4j [57].

5.1.5 Source Code Parser

Parsing source code for option and configuration nodes is much harder than configuration-specific formats. One problem is that configurations can be composed of different statements in more than one line, for instance:

```
LOCATION=$PROTOCOL + "://" + $HOST + ":" + $PORT + $FILEPATH
```

5.1.6 Plugin API

Providing an API for developers of microservices related tools for our implementation is very important. Developers of those tools have to be able to easily write plugins for our configuration networks. To achieve this, we have to think about a classification mechanism with which the type on an configuration can be described. The goal is that for example Docker developers can specify that the `EXPOSE` instructions adds a configuration for a port. Then, we can connect port configurations with the same values during our network initialization. These types should be independent from the actual implementation of concepts, such that a developer for a new concept plugin does not have to look into the implementations of all other plugins.

5.1.7 Feedback from Microservice Developers

Finally, the most important step before doing any further development on the prototype, is to interview developers of real-work software systems using microservices. Those developers are able to tell us about the problems occurring in the real world when developing microservices. This way, we can determine in which direction we should develop our configuration network approach.

Bibliography

[1] Martin Fowler. *Continuous Integration*. May 2006. URL: https://www.martinfowler.com/articles/continuousIntegration.html (visited on 07/02/2018).

[2] Rafał Leszko. *Continuous Delivery with Docker and Jenkins*. Packt Publishing, 2017.

[3] *Git Documentation*. git-scm.com. Chap. Customizing Git – Git Hooks. URL: https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks (visited on 07/02/2018).

[4] Rouan Wilsenach. *DevOpsCulture*. July 2015. URL: https://martinfowler.com/bliki/DevOpsCulture.html (visited on 07/02/2018).

[5] Gene Kim et al. *The DevOps Handbook*. First Edition. IT Revolution Press, LLC, 2016.

[6] *Gradle User Manual*. Gradle Inc. Chap. The PMD Plugin. URL: https://docs.gradle.org/current/userguide/pmd_plugin.html (visited on 07/02/2018).

[7] *On Cross-stack Configuration Errors*. 39th International Conference on Software Engineering. Buenos Aires, Argentina, 2017. URL: http://mcis.polymtl.ca/publications/2017/icse.pdf.

[8] Matt Welsh. *What I wish systems researchers would work on*. May 2013. URL: http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html (visited on 07/24/2018).

[9] John Wilkes. *Ωmega: Cluster management at Google*. 2011. URL: https://www.youtube.com/watch?v=0ZFMlO98Jkc (visited on 07/24/2018).

[10] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. *RUDSEA: recommending updates of Dockerfiles via software environment analysis*. Sept. 2018.

[11] Martin Fowler. *Microservices, a definition of this new architectural term*. URL: https://www.martinfowler.com/articles/microservices.html (visited on 06/28/2018).

[12] Silvae Technologies. *The majority of Netflix services are built on Java*. May 2015. URL: http://silvaetechnologies.eu/blg/50/the-majority-of-netflix-services-are-built-on-java.

[13] Peter H. Salus. *A Quarter-Century of Unix*. Addison-Wesley, 1994.

[14] redhat. *What's a Linux Container?* URL: https://www.redhat.com/en/topics/containers/whats-a-linux-container (visited on 07/05/2018).

[15] thildred. *The History of Containers*. URL: https://rhelblog.redhat.com/2015/08/28/the-history-of-containers/ (visited on 06/13/2018).

[16] Matteo Riondato. *FreeBSD Handbook*. Chap. Jails. URL: https://www.freebsd.org/doc/handbook/jails.html (visited on 06/13/2018).

[17] Paul B. Menage. *Adding Generic Process Containers to the Linux Kernel*. Jan. 2007. URL: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf.

[18] Jonathan Corbet. *Notes from a container*. Oct. 2007. URL: https://lwn.net/Articles/256389/ (visited on 06/13/2018).

[19] Jonatan Baier. *Getting Started with Kubernetes*. Packt Publishing, 2017.

[20] IBM Corporation. *LXC – Linux Containers*. GitHub Repository. 2008. URL: https://github.com/lxc/lxc (visited on 06/13/2018).

[21] Open Containers Initiative. *About – Open Containers Initiative*. URL: https://www.opencontainers.org/about (visited on 07/06/2018).

[22] Janakarim MSV. *From Containers to Container Orchestration*. May 2016. URL: https://thenewstack.io/containers-container-orchestration/ (visited on 07/06/2018).

[23] Erik Lupander. *Go microservices, part 7 – Service Discovery & Load-balancing*. Apr. 2017. URL: http://callistaenterprise.se/blogg/teknik/2017/04/24/go-blog-series-part7/ (visited on 07/01/2018).

[24] Puppet Labs, Inc. 2015. *State of DevOps Report 2015*. 2015. URL: https://puppet.com/resources/whitepaper/2015-state-devops-report (visited on 07/15/2018).

[25] *Jenkins*. Project Homepage. URL: https://jenkins.io/ (visited on 07/20/2018).

[26] *Jenkins User Documentation*. Chap. Creating your first Pipeline. URL: https://jenkins.io/doc/pipeline/tour/hello-world/ (visited on 07/20/2018).

[27] *Ansible*. Project Homepage. URL: https://www.ansible.com/ (visited on 07/20/2018).

[28] *Ansible Documentation*. Chap. Intro to Playbooks. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html (visited on 07/20/2018).

[29] Chris Richardson. *Microservices.io. Pattern – Externalized configuration*. URL: http://microservices.io/patterns/externalized-configuration.html (visited on 06/13/2018).

[30] Alejandro Duarte. *Microservices: Externalized Configuration*. 2018. URL: https://vaadin.com/blog/microservices-externalized-configuration (visited on 07/15/2018).

[31] Pooyan Jamshidi et al. “Microservices. The Journey So Far an Challenges Ahead”. In: *IEEE Software* May/June (2018), pp. 24–35.

[32] Erl and Naserpour. *Container Sidecar*. Arcitura Education Inc. URL: http://www.microservicepatterns.org/design_patterns/container_sidecar (visited on 07/16/2018).

[33] Docker Inc. *Docker*. Project Homepage. URL: https://www.docker.com/ (visited on 07/18/2018).

[34] Russ McKendrick and Scott Gallagher. *Mastering Docker*. Second Edition. Packt, 2017.

[35] *Docker Documentation*. Chap. Swarm Mode. URL: https://docs.docker.com/engine/swarm/ (visited on 06/26/2018).

[36] *Docker Documentation*. Chap. Docker Swarm. URL: https://docs.docker.com/swarm/overview/ (visited on 06/26/2018).

[37] Pivotal Software. *Spring*. Project Homepage. URL: https://spring.io/ (visited on 07/20/2018).

[38] *Spring Boot Documentation*. Pivotal Software. Chap. Developing Your First Spring Boot Application. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-first-application.html (visited on 07/02/2018).

[39] *Spring Boot features*. Pivotal Software. Chap. Externalized Configuration. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html (visited on 06/13/2018).

[40] Pivotal Software. *Spring Initializr*. Web Application. URL: https://start.spring.io/ (visited on 07/20/2018).

[41] Tianyin Xu and Yuanyuan Zhou. “Systems Approaches to Tackling Configuration Errors: A Survey”. In: *ACM Computing Surveys* 47.No. 4, Article 70 (July 2015). URL: https://dl.acm.org/citation.cfm?doid=2791577 (visited on 07/24/2018).

[42] Tianyin Xu et al. “Hey, you have given me too many knobs! Understanding and dealing with over-designed configuration in system software”. In: 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’15). 2015. URL: https://cseweb.ucsd.edu/~tixu/papers/fse15.pdf (visited on 07/25/2018).

[43] Ariel Rabkin and Randy Katz. “Static Extraction of Program Configuration Options”. In: 33rd International Conference on Software Engineering (ICSE’11). Jan. 2011.

[44] Theophilus Benson, Aditya Akella, and Aman Shaikh. “Demystifying Configuration Challenges and Trade-offs in Network-based ISP Services.” In: 2011 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM’11). 2011.

[45] Zuoning Yin et al. “An empirical study on configuration errors in commercial and open source systems”. In: 23rd ACM Symposium on Operating Systems Principles (SOSP’11). 2011. URL: https://dl.acm.org/citation.cfm?id=2043572 (visited on 07/25/2018).

[46] By Al Bessey et al. “A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World”. In: *Communications of the ACM* 53 (Feb. 2010). URL: https://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext (visited on 07/25/2018).

[47] Paul Anderson and Edmund Smith. “Configuration Tools: Working Together”. In: 19th Large Installation System Administration Conference (LISA’05). 2005. URL: https://www.researchgate.net/publication/220900305_Configuration_Tools_Working_Together (visited on 07/25/2018).

[48] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag Berlin Heidelberg, 2002.

[49] Haryadi S. Gunawi et al. “What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems”. In: 5th ACM Symposium on Cloud Computing (SoCC’14). 2014. URL: http://ucare.cs.uchicago.edu/pdf/socc14-cbs.pdf (visited on 07/24/2018).

[50] Mohammed Sayagh and Bram Adams. “Multi-layer Software Configuration: Empirical Study on Wordpress”. In: 2015.

[51] Chad Verbowski et al. “LiveOps: Systems Management as a Service”. In: 20th Large Installation System Administration Conference (LISA’06). 2006. URL: https://www.usenix.org/legacy/events/lisa06/tech/full_papers/verbowski/verbowski.pdf (visited on 07/25/2018).

[52] Bart Vanbrabant, Joris Peeraer, and Wouter Joosen. “Fine-grained access-control for the Puppet configuration language”. In: 25th Large Installation System Administration Conference (LISA’11). 2011. URL: http://static.usenix.org/events/lisa11/tech/full_papers/Vanbrabant.pdf (visited on 07/25/2018).

[53] Peng Huang et al. “ConfValley: A Systematic Configuration Validation Framework for Cloud Services”. In: 10th ACM European Conference in Computer Systems (EuroSys’15). 2015. URL: http://opera.ucsd.edu/paper/eurosys15-confvalley.pdf (visited on 07/25/2018).

[54] Kenny Bastani. *spring-cloud-microservice-example. An example project that demonstrates an end-to-end cloud native application using Spring Cloud for building a practical microservices architecture.* GitHub Repository. 2015. URL: https://github.com/kbastani/spring-cloud-microservice-example (visited on 08/13/2018).

[55] *Zuul Wiki. Getting Started 2.0*. URL: https://github.com/Netflix/zuul/wiki/Getting-Started-2.0 (visited on 07/23/2018).

[56] Jonas Hecht. *cxf-spring-cloud-netflix-docker. Example project combining Spring Boot apps with Spring Cloud Netflix (Eureka, Zuul, Feign) & cxf-spring-boot-starter*. GitHub Repository. URL: https://github.com/jonashackt/cxf-spring-cloud-netflix-docker (visited on 09/16/2018).

[57] Inc. Neo4j. *Neo4j*. Project Homepage. URL: https://neo4j.com/ (visited on 09/16/2018).