Masaryk University
Faculty of Informatics

Clustering of CDI components in SilverWare

Master's Thesis

Slavomír Krupa

Brno, Fall 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Slavomír Krupa

Advisor: Mgr. Martin Večeřa

Acknowledgement

I would like to thank my supervisor Mgr. Martin Večeřa for his advice, support, and patience during my work on this thesis.

Many thanks are due to authors of open source libraries and frameworks for their excellent work that helps in many aspects of developers life.

Last but not least, I would like to express my thanks to my family and all friends for their support and motivate to complete this final step of mine studies.

Abstract

The aim of this thesis is to implement service discovery and remote calls between CDI components in the microservice platform SilverWare. These capabilities are implemented utilizing JGroups messaging framework, which supports all modern cloud platforms.

Versioning of microservices is another enhancement for service discovery introduced as a part of this thesis. The versioning allows running multiple instances of microservice with the different versions in the same cluster. Injection points can limit microservices that can be injected by specifying version queries. The last improvement for the SilverWare platform is an implementation of different load balancing strategies for remote invocations of microservices.

All implemented features are demonstrated in QuickStart that can be deployed to OpenShift or executed locally and developed code is open-sourced and publicly available as part of the SilverWare project.

Keywords

microservices, SilverWare, clustering, JGroups, SemVer, Docker, Kubernetes

Contents

List of Tables

List of Figures

1 Introduction

The number of Internet users is steadily increasing increasing, with reaching 3 billion users in 2014 [1]. Considering the size of the online market, it is worthwhile for companies to do business online. Gaining a competitive edge in the online market can result in more active users and more funds. The advantage can be achieved by introducing innovative features that competitors do not provide.

Changes in development cycle must be made to to release new features on the market sooner and agile software development methodologies are helping with decreasing the length of a development cycle. On the other hand, using inappropriate architecture style can prolong the deployment or results in a bad scalability of a developed application.

Microservice architecture promotes faster changes, scalability, and also shorter deployment time. In the microservice architecture, a system consists of many independent services that encapsulate one functionality and communicate with other services via messaging or remote procedure invocation. Microservices can be deployed, scaled, and changed as a single unit.

Implementation of service discovery and remote invocation between CDI[1] components in the microservice platform SilverWare is the main goal of this thesis, and it is extensively described in the following pages. The thesis itself is divided into six relevant chapters. The first chapter contains motivation of the thesis with the overview of the next chapters.

The second chapter of the thesis describes the key features of microservice architecture and compares it with monolithic architecture. The subsequent sections outline the advantages of microservice architecture and point out the disadvantages with potential solutions or workarounds.

The third chapter presents existing solutions for microservice platforms and compares them to SilverWare. Microservice frameworks used for comparison are written in Java, so the comparison is adequate.

The primary intent of the fourth chapter is an explanation of concepts used in messaging systems. The concepts were chosen with a

1. Context and Dependency Injection

focus on the messaging library JGroups, which is a base of clustering implementation in this thesis. This chapter also contains a description of integration JGroups library with cloud providers, a brief naming of possible configuration options, and information on SilverWare adapters.

The most important ideas of this thesis are in the third chapter, which describes SilverWare functionality behind the curtains. At the beginning of the chapter, the boot process is explained. Following section details the approach used in SilverWare that allows tight integration with other technologies. Later in the chapter, my contributions to the SilverWare project are explained with reasonings why they are useful for developing microservices using SilverWare.

The fifth chapter describes the development of QuickStart that could be deployed to OpenShift[2] with Maven build and benefit from running in the cloud. The following sections contain a step-by-step description of the technology stack that is used for demonstration of SilverWare capabilities. The last chapter is a conclusion of the work done.

2. Cloud computing solution by Red Hat.

2 Microservice architecture

This chapter will focus on describing microservice architecture as well as on outlining the positive and negative aspects of the microservices concept. The following sections are inspired by *Introduction to Microservices* [2], *Building microservices: designing fine-grained systems* [3] and *Microservices* [4].

2.1 Comparison with monolith architecture

A comparison with monolith architecture is probably the most descriptive way to introduce microservice architecture, because the differences can be clearly identified. According to Lewis and Fowler [4], an application written using the monolith architecure consists of three main parts:

- client-side application,
- data storage and
- server side application.

Data storage is usually a relational database or *NoSQL* alternative, and a client-side application often relies on the REST API provided by a server-side application. The server-side application then provides an adapter to work with data storage and exposes an API for the client-side application. The entire business logic is ensured by a huge code base, which is unfortunately the most convenient way of development in the currently available IDEs[1]. Another advantage of this approach is that it can be easily grasped by developers, because it is the approach they are taught at universities.

Application logic can be divided into modules using tools provided by a programming language (e.g., classes, packages, and projects in Java), and interaction between modules is handled by the programming language. The performance of monolith application can be scaled by running more instances of a server-side application behind the load balancer that redirect the requests to the instances.

1. Integrated Development Environment

The main disadvantage of this approach is that most of the actively developed applications tend to grow over time and become unmanageable by a single developer [2]. The deployment time usually rises to minutes and creating new features or fixing bugs in existing code takes more effort. Also, better hardware may be needed for the deployment in order to satisfy all the application requirements. Assuming that some modules are more CPU intensive, and others are more memory intensive, they would both require a faster memory and CPU because they are deployed as a single application.

The problems mentioned in the previous paragraph are effectively addressed by microservices architecture. The system consists of many independent services that encapsulate one functionality and communicate with other services via messaging or remote procedure invocation. Microservices can be deployed, scaled, and changed as a single unit. Thus programming languages and technologies are not bound to other microservices.

2.1.1 Api Gateway

On the other hand, communication of client-side application directly with microservices might bring undesired dependencies of the client application on the microservices API. These dependencies unveil a problem which can be eliminated by using an *API gateway pattern* suggested by Richardson[5]. This pattern introduces a microservice called gateway, whose purpose is to incorporate the API of all microservices, and provide it to a client-side application in a monolith manner. An equally important change is a reduced chattiness of internet communication and a decrease in latency achieved by performing microservice invocations on the local network.

In other words, a gateway can aggregate the responses from multiple microservices, cache results of previous calls, and solve authorization of the access in one place as can be seen in Figure 2.1. At the same time, API gateway reintroduces bottleneck for the application, and every API modification of the microservices must be propagated to the gateway. In either case, it promotes the idea of separation from the client by creating an abstract layer that manages application functionality and incorporates all microservices.

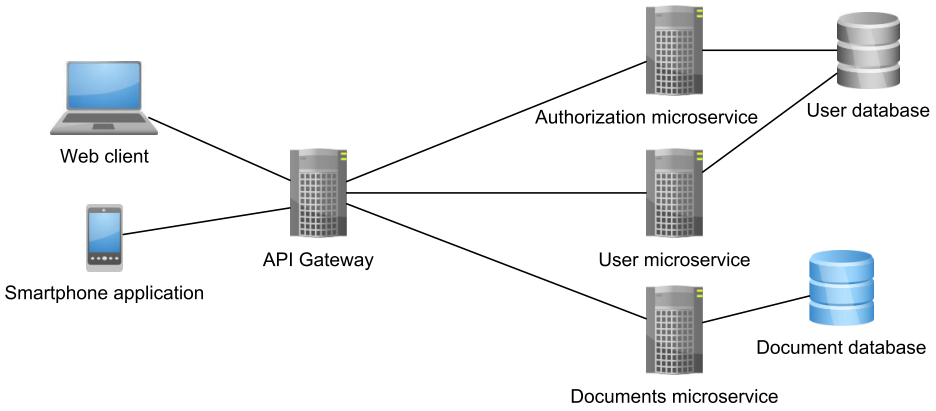

Figure 2.1: Illustration of the API gateway pattern[5]

2.1.2 Promote product

Development and production concepts are also treated oppositely to monolithic architecture. Monolithic applications are usually developed by one team, and another team is responsible for the running in the production. Developers are moved to other projects after project successfully passes the criteria to move to production. On the other hand, in microservice world, the application is treated as a product and its developers are made accountable for the components implemented by them. The production delivery time must be shorter so bugs can be detected in a production environment, while the developers have a fresh memory of the work done to ensure accountability.

2.1.3 Infrastructure

Another part of microservices architecture is the way updates to infrastructure are managed. The old fashioned way is to manage infrastructure manually, which can result in a difference in configuration between production and testing environments. The next step in the evolution of infrastructure management is creation of disk images from the first installed server and apply them to all other servers. This achieves the same configuration on all servers but also gather configuration which should not be shared between servers (e.g., IP addresses).

The mentioned methods are known as the snowflake server [6], because it becomes fragile over time as it becomes hard to understand and modify.

On the contrary, there are Phoenix servers [7] which are managed by automated tools and can be easily recreated from scratch. When using a Phoenix server, the `SSH` access should be forbidden to enforce using a configuration management tool. The advantage of this approach is that all changes are made by automated tools so that everything can be audited. A recreation of the server can be a slow operation so the management tools usually allow applying commands to all managed servers.

2.2 Relationship of microservices and Service Oriented Architecture

Microservice are occasionally compared to Service Oriented Architecture (SOA). There are similarities between these architectures, specifically: reusing parts of created services and other features based on this characteristic, remote procedure invocation as a part of an architecture. However, according to Lewis and Fowler [4], differences can be observed. SOA is mainly focused on using Enterprise Service Bus (ESB) which usually provides complicated methods of redirecting, transformation and applying business rules to messages. On the other hand, microservice architecture is more related to sophisticated services rather than routes connecting the services. For this reason, the main communication of the services is synchronous or asynchronous messaging executed via lightweight protocols (e.g., REST).

In the SOA service, discovery is done by querying a service directory, which is a single point containing a list of all available services. On the contrary, in microservice architecture, service discovery does not have prescribed implementation and may be done by querying services individually.

Last but not least, the SOA allows content negotiation [8]. This means that services can prescribe multiple supported formats, and the used format is negotiated between the services.

2.3 Advantages of microservice architecture

The advantages mentioned in the following paragraphs are the reason why microservice architecture exists.

1. **Independent technologies stacks** – Separating an application into microservices implies that every microservice can use different technology stack which suits best the task performed by the component. If the performance critical services are implemented in programming languages which are optimized for the given task, performance boost can be achieved with less effort as optimization would take. Also when creating a new microservice there are no frameworks prescribed and they can be chosen based on the knowledge of the team.

   Another advantage is that a technology upgrade or change is more likely because it is relatively easy to adapt the small unit to a newer technology compared to a large monolith application.

2. **Single responsibility** – Microservice is usually responsible for a single unit. This fact allows the company to be structured similarly to microservice architecture and create teams responsible for a single microservice. The teams can master the subject of the microservice and become experts in the field. Additionally, new employees are efficient sooner because the code base of the microservice as separated part of the larger application is significantly smaller than the code base of monolithic application of a similar size.

3. **Deployment time** – One of the biggest disadvantages of the monolith applications is the deployment time. Even the tiniest change results in building a whole application anew which can take few minutes for large code bases. Furthermore, all tests must be run, and an upgraded version must be started. The previously mentioned process can take a few minutes, and the developers usually need the result of the process which increases their ineffective time. On the contrary, in microservices architecture, only a separate unit needs to be deployed which usually takes no more than a few seconds.

4. **Independent scaling** – The performance of the application is commonly limited by a few application components which cannot perform as fast as other components. A small portion of an application brings down the performance of the whole application. In microservice architecture, it is possible to scale service limiting performance to more instances or better hardware so it will not be a bottleneck anymore.

2.4 Drawbacks of microservice architecture

When new architecture is introduced, it is impossible for it to work without drawbacks. Microservice architecture has many drawbacks which are a consequence of splitting the application into services and invoking remote services. Some of the drawbacks are similar to SOA drawbacks and the solutions found for SOA can be applied.

1. **Data consistency** is one of the biggest weaknesses of the Microservice architecture. Imagine a scenario where multiple entities are changed (these entities are maintained by microservices with different technology stacks) and all of these changes should be part of a transaction. The consistency of the microservice data stores cannot be sustained by the distributed transactions because they are not supported by all message protocols and all data stores (i.e., not every NoSQL database supports transactions). Data consistency is often preserved by implementing a mechanism which is able to find inconsistencies and execute compensating operations. Another approach to maintaining data consistency recommends creating one microservice responsible for managing transactions, and all microservices which are part of a distributed transaction should contact this microservice when they start, abort or finish the transaction.

2. **Handling failures** is trivial for monolithic architecture where the problem can be reduced to exception handling. In the microservice world, it is possible that some services will be slow or unavailable for a short period of time, and other services must be designed to handle these unexpected scenarios. Im-

plementing microservices in this defensive manner demands more development time.

Failure scenarios can be simulated by frameworks. One of these frameworks is Chaos Monkey[2] which destroys randomly chosen service, and developers can observe whether it would break the whole system. This process should be done during working hours so developers can act in case of problems.

3. **Monitoring** the state of microservices is critical in preventing their failures. Semantic monitoring (e.g., number of processed and failed requests) should prevent application scale failures by notifying developers about a problematic service. Developers should analyze reported problems and try to restore working state. Another approach suggests running *health checks*[3] which restarts the service in case of any problems.

4. **The size of microservice** – Separating monolithic service to components can be challenging, and there is no definition for the right size of microservice. As a result, some microservices are created as *nanoservices*[4] and other components are exceptionally large. Currently, there is no universal approach for partitioning application to microservices; however, techniques existing for *Service Oriented Architecture* suggest that one component should cover functionality with one entity (e.g., recommendation, account) or standalone functionality. This is similar to Single Responsibility Principle (SRP)[9].

5. **Order of deployment** – Implementation of change which covers multiple microservices might break the interface of other microservices. Although one of the purposes of microservice architecture is a faster and simpler deployment, in this scenario specific order of microservices deployment may be required.

---

2. Available at `https://github.com/netflix/chaosmonkey`
3. The health check is a procedure which test the service's basic functionality.
4. Nanoservice is service which is remarkably small and should be merged with other services in order to keep the overhead of Inter-process Communication (IPC) to the manageable level.

6. **Increased footprint of application** – Microservices run as multiple services using an Inter-process Communication (IPC). First of all, these services run as separated programs which means that all libraries required by the microservices, even if they are used by multiple microservices, must be loaded to memory multiple times. The next step is that microservices are often implemented using modern programming languages executed in sandboxes (e.g., Java, C#) to isolate the program from OS which adds a second layer of overhead. The final step is using some orchestration service which often uses virtualization for the microservice environment (this is described in section 6.1.1), adding another layer of overhead to the application.

7. **Testing** – To properly test microservice interaction with other microservices, all dependent microservices must be deployed. Hence, mechanism for deploying a subset of the microservices is required. Another approach of testing requires creating stubs for all microservices used directly from a tested microservice. Both methods require additional non-trivial activity to be done by developers in order to run the tests.

3 Microservice Frameworks

A number of frameworks for creating microservices is on rise, and this chapter contains a brief overview of existing solutions, which can be considered as competitors for SilverWare. All the mentioned solutions are written in Java, and they are closest competitors currently available. Vert.x project is not mentioned because it is integrated to SilverWare as one of the modules.

Most of current Java solutions are using *fat jars*. A *fat jar* is a Java archive which contains all necessary dependencies and is runnable on its own by Java Virtual Machine without any additional containers.

3.1 SilverWare

SilverWare is an open source framework written in Java. It provides tight integration with multiple technologies which help to create simpler microservice solutions in a more straightforward manner. By using SilverWare, the developer can entirely focus just on the development of the application and does not waste time with service integration. The source code of the framework is available on GitHub [10] and a current release can be downloaded from Maven Central Repository[1]. Demo Repository [11] contains quickstarts for essential use cases and demos which combine multiple use cases into real life applications. These demos were created for presentation purposes at conferences or community meetups.

Applications written in SilverWare are built as `jars` with external dependencies. Build process allows modularity of created application by defining dependencies in the project dependency management. Developers can then choose only the necessary modules and decrease the size of the resulting application. External dependencies also bring an advantage of easier deployment and configuration changes.

SilverWare is part of multiple projects in SilverSpoon family that includes various projects for creating a simple Internet of Things applications. In order to make applications runnable on small devices with minimal performance needs, Silverware is a minimalistic form

1. `http://mvnrepository.com/artifact/io.silverware`

of implementation. The core module of SilverWare is called *microservices* and contains application logic for start of the framework and functionality shared by other modules. Other modules are named according to technologies they integrate into SilverWare. Currently there are seven modules supported and more in the development. The following chapter and especially the section 5.1.1 describes how SilverWare is implemented.

3.2 WildFly Swarm

WildFly Swarm is an open source server from Red Hat which was created from WildFly[2] server. The main purpose of this server is to make a transition from the monolithic architecture to microservice architecture smoother [12]. WildFly is the third generation of JBoss application server, and it was designed with a focus on dynamic modules loading and parallel platform start. Modules in WildFly Swarm are called fractions, and they have their dependencies (e.g., logging for JAX-RS). Fractions which are often used together are grouped to MicroProfiles.

At the time of writing this thesis, there are around 80 fractions available. Almost half of them are based on Java EE standards, and the rest of them provide integration with other tools demanded in the microservice world. However, the integration is not as tight as in other frameworks and it often is just added as a dependency. An application running on WildFly Swarm can be created by specifying the dependency on a WildFly Swarm Maven plugin. The Maven plugin will then try to auto-detect which fractions should be loaded. Autodetect can fail to find the correct fractions, and it is possible to skip this step by manually specifying the fractions in POM[3] as dependencies or specify them in its main class of application. During the build process, the fat jar is created, and that can be run directly on JVM.

The environment-specific configurations allow developers to change the configuration of the application server at different stages of the development cycle. For example, debug level logging can be allowed

2. `http://wildfly.org/`
3. Project Object Model

at the testing stage, but in production, just info and error messages are logged.

WildFly Swarm also allows invocation of the remote services. Remote invocation uses third party component Consul[4] which is key-value storage used for service discovery. Using a third party brings the benefits that application developed in other languages can use the same storage and format for the service discovery. On the other hand, it brings the single point of failure and somewhat contradicts the microservice architecture principles.

WildFly Swarm provides many fractions enveloping other libraries. The next enumeration is a just brief incomplete list of components which I considered worth mentioning:

1. Health status of the microservice can be exposed to the outside world (or other microservices) by using annotation `@Health`.

2. Resilience is implemented similarly to SilverWare using Netflix Hystrix library.

3. *Distributed tracing* tracks the latency of service invocations between microservices and can help to find the bottlenecks of the application.

3.3 Spring Boot

Spring boot[5] is another open source framework which allows creating a standalone runnable fat jar from Spring-based Java applications. Fat jar creation is done similarly to WildFly Swarm by maven plugin which is developed by a spring community.

Spring as a platform supports an enormous amount of libraries and third party systems just with adding a dependency on the spring component which adapts the component. The philosophy of the spring platform is convention over configuration, so the integration with other services is done with minimal effort on the developer's part (similarly to SilverWare).

4. `https://www.consul.io/`
5. `https://projects.spring.io/spring-boot/`

The different configurations per stage are supported by spring cloud config server [13] which is a rest-based application acting as an external provider of configuration for the other applications. By default, this server internally uses git for storing the data which enables different configuration branches for various stages of deployment. Another option is to use Vault[6] for the encryption of shared properties. It is necessary to manually trigger configuration reload on depending spring boot applications manually after the change was done on the configuration server.

Spring integrates technologies from Netflix for microservices, allows sharing of the security provider for all services, and simplifies the deployment on a several cloud providers. Besides that, it allows using distributed message queues and distributed invocation tracking by using the integrated Zipkin[7] system. Zipkin is an invocation monitoring system based on Dapper [14] which measures invocation times across microservices. Zipkin module is packaged as another microservice within the Spring platform which provides an endpoint for gathering the data and user interface which supports sorting and filtering of traces. The user interface comprehensively displays retrieved data and can be a great help in troubleshooting issues within microservices.

Service registration and discovery is done using a third party Eureka[8] library from Netflix. This library runs as a separate microservice which contains the data needed for service discovery and service invocation. Remote invocation is executed as REST call from a proxy. This proxy is a REST client created by Feign library[9].

6. https://www.vaultproject.io/intro/index.html
7. https://github.com/openzipkin/zipkin
8. https://github.com/Netflix/eureka
9. https://github.com/OpenFeign/feign

4 Messaging systems

Messaging is a well-known and widely used concept in computer science which is usually implemented to integrate applications with other applications using a messaging system [15]. Messaging systems are libraries or frameworks which enclose the functionality of sending and receiving messages between two or more applications (possibly written using different technology stacks). The main goal of a messaging system is message delivery which can be achieved using multiple methods depending on application demands. It may be requested to pair responses with requests or timeout messages after a certain period. These claims can be satisfied by message systems in multiple ways. This chapter gives a brief walk-through of the concepts used in the implementation part of this thesis.

A message system can be introduced by the following figure which shows the main components. These components may differ in implementation, or some of them may be skipped, but usually, they are part of the system.

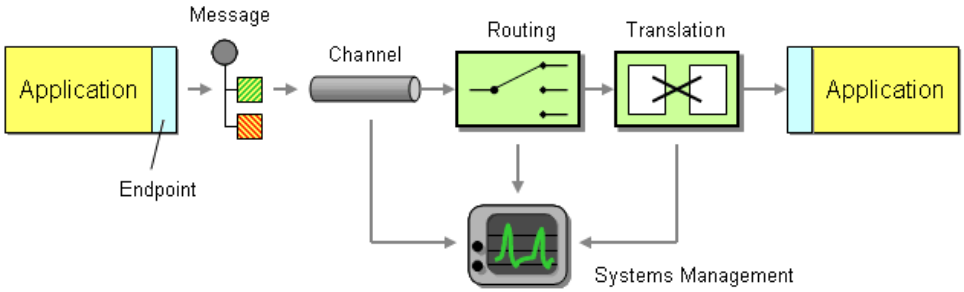

Figure 4.1: Illustration of messaging system components [16].

4.1 Message

A message is a unit of data whose purpose is to wrap the content which should be transmitted to another application. The content of the message needs to be an array of bytes or an object which is deterministically serializable to a byte array. Usually, a message is composed

of a header and a body. The body contains content which should be delivered and is not modified by the messaging system. On the other hand, the header contains information which advises the messaging system and may change during transmission. The described concept is similar to concepts used in real life or computer networks. Messages managed in a message system are classified by purpose into three groups of messages differing merely in the meaning, not their syntax [15, p. 67].

1. A command message invokes a method in a remote application. Enveloping the invocation into a message object allows asynchronous processing [15, p. 145]. The invocation is executed after delivering and processing of the message by the addressee. Response to the command message containing the result is then delivered to the sender usually as a document message.

2. The purpose of the document message is the transfer of data between applications. The main difference between the command and document message is that a document message does not specify how it should be processed by the receiving application. The application decides whether and how the message is processed.

3. When an event which should be spread to other connected systems occurs in the application, an event message should be sent immediately. Event messages are similar to document messages but the main contrast is in primary liabilities of these messages. While the main intent of the document message is to deliver the content of the message, the main intent of the event message is the time when it is delivered. Event messages delivery and processing should be prioritized in a messaging system.

Request and reply [15, p. 154] were mentioned as concepts in the previous paragraphs. Pairing a response message with a request asynchronously can be achieved using correlation identifier [15, p. 163]. This identifier introduces two fields in a message header: the mandatory message identification, which uniquely identifies the message in a messaging system, and an arbitrary correlation identifier which

contains the message identification of the request message. Using this concept allows unlimited chaining of requests and replies without creating a conversation.

Last but not least, the message can specify its expiration time in the header [15, p. 176]. This parameter may be useful when the message validity is limited by time (e.g., availability of free resources). The messaging system guarantees the eventual delivery but not in a timely manner. Adding expiration time can save some resources by skipping messages which are no longer valid.

4.2 Channel

Messages sent from one application do not magically appear in a receiving application. The messages are transported using a connection between two nodes represented by a channel [15, p. 60]. Channels hide the logic of a transportation message between nodes and are used as logical addresses in the messaging system. There is no universal approach to implementing a messaging system, and every vendor can adjust the implementation to use cases which are demanded by the customer.

The channels are usually created during deployment of the application but some messaging systems also support a dynamic creation of channels. The problem with dynamic creation is that the receiving application must be notified about a newly created channel. Some messaging systems may allow a creation of the channel directly using the API, but others may just allow connection to already channels created. Nevertheless, channels are not created automatically, and developers must know which channels are required and create them ahead of time.

The channels which represent a connection between two applications are called *point to point* channels [15, p. 103]. If there are multiple receivers on one channel they are called *publish-subscribe* channels [15, p. 107].

Point to point channels can be used for faster communication between two nodes but can be resource-consuming if the node's network is represented by a complete graph. Guaranteeing message consumption on the level of consumers is not a trivial task, and it is more

straightforward for the channel to allow exactly one consumer to be connected.

On the other hand, there is a publish-subscribe model which extends the observer pattern from *Gang of Four* [17]. In this model, the messages are received by all applications which are subscribed to the channel. To guarantee delivery, the channel must fulfill the following properties [15, p. 108] :

1. Messages should not be deleted from the channel or the sender before they are processed by all consumers.

2. Messages should disappear after being handled by all consumers.

Publish-subscribe channels are a more scalable solution for a higher number of subscribers. Another use case of the publish-subscribe channel is debugging. The debugging application could be one of the subscribers attached to the channel which has a minimal impact on performance and does not change the application logic.

The disadvantage is that mechanisms used for debugging can also be misused to eavesdrop on messages flowing into the channel. No messages will be lost so another detection mechanisms must be utilized. Auditing of active subscriptions is a good mechanism for detection of unwanted subscribers, and a manual subscription approval could also prevent creating this security issue.

4.2.1 Channel hierarchies

As was mentioned before, the channels represent logical addresses and may form a hierarchy also represented in the messaging system. An example of a hierarchy can be a `MyCompany` channel with `MyCompany\Orders` and `MyCompany\Packets` subchannels. Subscribers can use wildcard `MyCompany\*` for channel subscriptions from all subchannels of `MyCompany`.

4.2.2 Datatype channel

A subtype of a point-to-point channels are *datatypes channels* [15, p. 111] which allow just a single message type to be transferred via this

channel. The unified format of the message can save traffic and make the processing of the messages faster as header contains less data.

4.3 Receiving messages

Processing of incoming messages can be achieved by two approaches:

1. polling consumer,
2. event driven consumer.

The polling consumer implementation is fairly straightforward as one thread of the application is dedicated to periodically challenging a message system with a request for newly delivered messages. When a message is delivered, the consumer processes the message (or invoke asynchronous message processing) and continue to poll the message system for new messages.

The advantage of this approach is that the consumer is in charge of requesting messages. Therefore, the application is not over-flooded with messages as it polls the new messages at the rate it manages to process them [15, p. 494].

In case the messaging system encloses multiple channels, there should be a consumer running in a distinct thread for each of them. There may be cases when some channels are regularly empty and receive messages just occasionally. These channels can be processed by one consumer to save thread resources.

Another method of processing incoming messages is not explicitly requesting messages from a channel but letting the channel inform the application when a new message was delivered. Thread resources are saved, but on the other hand, the application can be over-flooded with messages because the rate of message processing is managed by message delivery and not the application itself. This approach is called event-driven [15, p. 498] as an event of the received message is in charge of the processing messages.

The messaging system must be informed which code should be invoked when the message is delivered. This is done by registering a callback in the message system. The callback can be a method or an object which is capable of processing the message and it is invoked to

process every received message. The messaging system may run the invocation of the callback in multiple threads so the operations which are done in callback should be thread-safe.

Most of the real life messaging systems do not use *datatype channels*[1] because having a channel for every message type soon becomes unmanageable. In channels with multiple message formats, it is necessary to find a correct consumer for the message. As was mentioned before, multiple consumers require coordination between consumers which is not easily achievable. *Message Dispatcher* pattern [15, p. 508] brings a solution which introduces one consumer called dispatcher which is aware of all message formats and consumers which can process them. The dispatcher implements the mechanism which chooses the right consumer[2] based on predefined rules. Rules can be based on the message header or content type as a dispatcher can also work with the content of the message.

4.4 Sending messages

Sending a simple message via a message system is more straightforward. Firstly, a message with all necessary attributes is created and then handed to the message system. The message system takes care of successful message delivery. This approach is called *send and forget* because sending application does not care whether the message is delivered and leave the responsibility to the messaging system. On the other hand, there are scenarios where the user needs a reply from the application as a confirmation that the message has been delivered or the remote command has been invoked. In the section 4.1 the *corellation identifier* has been mentioned which matches reply messages to request messages in messaging systems. The matching is done by the messaging system, and there are two options how the reply can be retrieved from the messaging system.

The first option is synchronous waiting for the reply message. This approach is similar to local invocation because the messaging system wraps the complexity of the problem and developers use API that they are familiar with to retrieve the response. The disadvantages are

1. Datatype channels were introduced in section 4.2.2.
2. The consumer is called a performer in the dispatcher pattern nomenclature.

similar to the *polling receiver* because after sending the message, the task of the messaging system is effectively transformed to *blocking wait* for the reply. Moreover, the thread waiting for the answer is not a thread which was created especially for this task and usually blocks the application execution which degrades the performance. Hence the situations where the synchronous waiting is the correct option are rare.

Next option, which is used more often, is asynchronous processing of the incoming replies. The receiving process is similar to *event-driven receivers* and the response handler is registered during the message-sending phase. The main advantage is that the sending thread can continue the execution and is not blocked anymore. However, this solution also brings its own problems. Asynchronous processing of responses is not as common as a local method invocation. The message handler must be registered and it must save the context of the request message for processing of the reply message. The registered handler is invoked later at the arrival of the response which makes debugging and troubleshooting trickier.

4.4.1 Scather gather

Another typical use case is sending a request to multiple applications and retrieving responses from them as one message. The mechanism allowing this use case is called *scatter gather* [15, p. 297]. The use case can differ in the amount of responses required to end gathering responses. The most common approach is that a sending application has a list of recipients to whom the message should be distributed and which should respond to requests. This method expects that all responses are delivered before processing of reply messages can be executed. A problem may occur when part of the recipients is temporarily or permanently disconnected from the messaging system. In the second case, the application waiting for the responses is in danger of starvation. The simplest solution is to use timeout period after which all unreceived messages are considered delivered with an error response.

Another useful approach may consider just the first response and throw away other responses. The first response can be regarded as a sample representing a state of all applications connected to the

messaging system. Last widely used use case considers just a subset of replies and starts processing after a specified necessary amount of messages has been delivered.

4.5 JGroups

JGroups [18] is a Java library which provides tools for implementing reliable messaging systems. The messaging system is called cluster in JGroups nomenclature, and the member applications of the communication are called nodes. Nodes are identified by an address which is JGroups interface with multiple implementations.

The main component of the JGroups is the class named `JChannel`. `JChannel` represents a messaging channel[3] between a group of nodes which is identified by the channel name and the owner. The owner of a channel is often called the leader in network terminology, and after creation, it is the node which creates the channel. Disconnection of the previous owner of the cluster promotes the oldest member of the cluster to a leader position.

After setting a channel name and establishing a connection to cluster, `JChannel` provides a pack of functionality which makes basic communication in a cluster straightforward for the developer. The functionality can be divided into three main groups:

1. retrieving cluster information,
2. sending messages,
3. receiving messages.

The collection of functions which is mentioned as the first obtain the information about members of the cluster and is implemented directly on the `JChannel` API. Tracking connected nodes of the cluster is the basic functionality of JGroups. State of the cluster is represented by the class called `View`, which allows to retrieve addresses of alive nodes and observe changes. Whenever the cluster node crashes, connects, disconnects or is suspected from being disconnected from the cluster the `View` on all other nodes is updated and JGroups release event with this change via `MembershipListener` attached to `JChannel`.

3. More about channel can be found in section 4.2.

Another part of the API allows a developer to set a custom state of the node which can be retrieved from other nodes by calling `getState` method with node address as a parameter on `JChannel`. The state object is Java byte stream so large objects can be transferred more efficiently using `JChannel`. A common approach is sharing one state across the cluster, and it could be implemented using channel creator as state holder and distributing all changes from an owner.

`JChannel` provides just asynchronous message delivery to other nodes identified by an address without guaranteed delivery time and order of delivery. The content of a message must be an array of bytes, but the JGroups provides utility methods for conversion of *Serializable* classes which extends the possibilities but still does not remove limitation for allowed content.

The reason for this minimalistic implementation was keeping channel implementation similar to sockets as a well-known concept of communication and fulfillment of a *Single Responsibility Principle* [19, p.95].

Most of the functions which belong to the sending messages group are implemented in *Building blocks* [18]. The building blocks are internally using a `JChannel` for cluster communication, but they are adding value to the developer by implementing tasks which are often performed in messaging systems. One of the most recurring pieces of code is an association of requests and responses allowing synchronous communication between cluster nodes. Synchronous messaging is implemented in `MessageDispatcher` and allows different invocation and processing of the replies.

`MessageDispatcher` provides methods for sending synchronous and asynchronous messages and also allows setting an amount of responses needed for *scather gather* variances. The result of non-blocking synchronous messages is processed by class `FutureListener`[4] which is tied to the request by `NotifyingFuture`. Provided code example explains the relation between classes.

4. In JGroups with major version 3 `FutureListener` extends `java.util.concurrent.Future` but in JGroups with major version 4 just java `Future` is used.

Listing 4.1: Relation between classes used for non-blocking messages

```
NotifyingFuture<RspList<T>> notifyingFuture =
   dispatcher.castMessageWithFuture(...);
notifyingFuture.setListener(new FutureListener<
   RspList<T>>() {
  @Override
  public void futureDone(final Future<RspList<T>>
     future) {
     //what should be done with messages
  }
```

Listing 4.1: Relation between classes used for non-blocking messages

The code in the `futureDone` is invoked in own thread managed by the JGroups and response message for the request is passed as the parameter. The `RspList` wraps the multiple messages from *scather gather* and also result. The result can be an exception thrown on a remote node or class containing the response.

The last group of the provided functionality allows processing of incoming messages. Registering an implementation of a `RequestHandler` or `MessageListener`[5] on `JChannel` is necessary to enable message processing. Registered callbacks are then invoked automatically in thread pool managed by JGroups when a new message is delivered. Regardless of the thread pool, the best practice is to keep the processing time to a minimum and not invoke any blocking operations at all to avoid thread pool drain. Fully asynchronous message processing is possible by registering a callback using `AsyncRequestHandler` interface. `AsyncRequestHandler` contains a method with parameter `response` which wraps all required information for associating the response to the request on the sending node so the processing can be done in other thread and the response can be sent later.

4.5.1 JGroups configuration

JGroups library provides modifiable structure and contains multiple implementations of protocols for transport, service discovery, group

5. `RequestHandler` is interface for synchronous message processing and `MessageListener` is for asynchronous messages.

membership observation, view fragmentation and failure detections [20]. The configuration of JGroups is stored in a *XML* format with own *XML schema* which covers standard protocols with attributes from JGroups. The configurations of different sections are independent which allows many combinations and different protocol stacks.

The basic configuration of JGroups includes especially setting a size of thread pool which handles messages, rejection policies, timeouts, and other attributes. Most functionality is built on transport protocol which supports three main implementations: UDP, TCP, and tunneling; which is the most secure variant for cluster over the Internet. The rest of possible protocols with their attributes and implementations are listed in the documentation [21].

4.5.2 JGroups Extras

The trend of nowadays is to migrate the application's deployments to the cloud. To satisfy this trend, the JGroups has to evolve and support cloud-based solutions. In the cloud environment, it is often not possible to support UDP and TCP service discovery.As a response, new discovery protocols, which are using provider-specific techniques, were implemented. These discovery protocols are distributed as separated projects in JGroups-extras[6] and other projects mentioned in JGroups 4 documentation [22]. Currently, every major cloud provider is supported by external libraries. QuickStart created as part of this thesis contains the configuration of JGroups with JGroups extras library. More information about QuickStart is in the chapter 6.

4.5.3 SilverWare adapters

In SilverWare there are use cases for the messaging and it makes sense to hide the complexity of JGroups by creating adapters. These adapters take care of correct setting of JGroups and expose simpler methods which provide the necessary functionality.

The first adapter `JgroupsMessageSender` provides methods for sending synchronous messages and asynchronous messages with single node address variant and cluster-wide variant. `JgroupsMessage-`

6. `https://github.com/jgroups-extras`

`Sender` methods also allows filtering recipients by adding a blacklist of addresses which should not receive the message.

The second adapter `JgroupsMessageReceiver` is registered to `JChannel` for observing cluster state and processing of incoming messages. Concept of the class was inspired by *message dispatcher* and it allows registration of message `Responder`[7] which is tied to type of content. When the message is received selection of `Responder` is based on type of content. New implementations of `Responder` can be registered later which makes the solution more scalable.

7. `io.silverware.microservices.providers.cluster.internal.message.responder.Responder`

5 SilverWare implementation

The first part of this chapter is focused on internal implementation of the Silverware platform. The following sections describe the clustering implementation which was the topic of this thesis, and the last sections are dedicated to microservice versioning which allows the developer to solve the order of deployments problem[1] and grant the possibility of running multiple versions of microservices in the same cluster.

5.1 SilverWare internals

The core module of SilverWare is called *microservices*. The main goals of this module are executed in the `Boot`[2] class. The boot process resolves the configuration for the platform, sets the shutdown hook and starts a thread with the bootstrap of the platform. The configuration of the platform is stored in class `Context`[3] whose instance is accessible in all modules and allows storing data shared across the platform. Moreover, storing the configuration in one object instance allows SilverWare to run multiple instances on one JVM.

As was mentioned in section 3.1, SilverWare contains multiple modules which are integrated with core project. Microservice integration contract with SilverWare is explicitly made by implementing `ProvidingSilverService`[4]. All classes which implement this interface are dynamically looked up, instantiated, and initialized with `Context` at the boot of the platform using reflection. `ProvidingSilverServices` are isolated from other providers by running in a separate thread.

5.1.1 CDI provider

Most of the integrations in SilverWare are done by dependency injection described in *Contexts and Dependency Injection for the Java EE platform* [23]. Developers specify microservices they want to inject by

1. Mentioned on page 5.
2. `io.silverware.microservices.Boot`
3. `io.silverware.microservices.Context`
4. `io.silverware.microservices.ProvidingSilverService`

type, qualifiers, and annotations; then SilverWare will resolve and inject the correct microservices for them.

The CDI[5] provider is a vital component of SilverWare integration with other microservices because it creates proxies which later look up microservices from all providers.

Explanation of SilverWare internals should be started with an example of the microservice bean and the injection point.

Listing 5.1: Microservice bean and the injection point.

```
@Microservice
public class ReferencedBean{
    @Inject
    @MicroserviceReference
    private OtherBean injectedBean;
```

Listing 5.1: Microservice bean and the injection point.

Annotating class with `Microservice` annotation make this class life-cycle managed bean by SilverWare[6]. SilverWare managed beans can use annotation `Inject` and `MicroserviceReference` on their fields for the automatic injection of other SilverWare managed beans. The field `injectedBean` is an injection point and it is possible that multiple injection points for same managed bean exist.

CDI provider internally uses Weld[7] for a discovery of injection points and managed beans, but it implements its own CDI extension [23] which handles just `MicroserviceScoped`[8] beans [24]. Beans are registered as `MicroserviceScoped` by CDI extension[9] during startup of Weld if they are annotated with a `Microservice` annotation. For every injection point with `MicroserviceReference` the instance of custom bean[10] is injected. Injection points which are the same from the Weld's point of view share same custom bean. The bean must be

5. Context and dependency injection
6. Rules specifying how can class become managed are more strict and all are mentioned in *Contexts and Dependency Injection for the Java EE platform* [23].
7. http://weld.cdi-spec.org/
8. `io.silverware.microservices.providers.cdi.annotations.MicroserviceScoped`
9. `io.silverware.microservices.providers.cdi.internal.MicroservicesCDIExtension`
10. `io.silverware.microservices.providers.cdi.internal.MicroserviceProxyBean`

added to Weld context and unique relationship with injection point must be created to guarantee that Weld will match the bean with injection point. The relationship of the bean and injection point is arranged in Weld by using the same metadata for the created bean.

According to the CDI specification [23], the bean holds an instance of proxy for invocation. `MicroserviceProxyBean` returns new proxy for every injection point using `MicroserviceProxyFactory`[11].

This proxy internally invokes a chain of method handlers [24] which work as interceptors and can interrupt the invocation chain or modify the invocation. Implementations of `Methodhandler` are looked up dynamically after platform start by reflection and the order of `MethodHandler` in the chain is defined by annotation `Priority`. Dynamic lookup of `Methodhandler` implementations allows developers to create their custom method handlers, which intercept all invocations. No matter what interceptors are implemented, the last `MethodHandler` in the chain is always the `DefaultMethodHandler`[12] which has the highest priority and contains the metadata of a bean and injection point. This metadata is used to resolve microservices within SilverWare.

5.1.2 Microservice lookup

`DefaultMethodHandler` is the object where the SilverWare internal microservice lookup is started. It stores the instance of `LookupStrategy`[13] which describes the providers that should be queried and the microservice that will be chosen in a case when multiple microservices are returned from providers. Currently, there are three implementations[14] of `LookupStrategy` interface which differs just in small details.

1. `FirstFoundLocalLookupStrategy` is the fastest strategy with a minimal footprint which can look up only local microservices and always returns the first found microservice. This implementation is used by default.

2. `RoundRobinLookupStrategy` looks up the microservices from both remote and local `ProvidingSilverService` instances and

11. `io.silverware.microservices.providers.cdi.internal.MicroserviceProxyFactory`
12. `io.silverware.microservices.providers.cdi.internal.DefaultMethodHandler`
13. `io.silverware.microservices.silver.services.LookupStrategy`
14. All of them are in package `io.silverware.microservices.silver.services.lookup`

in case multiple microservices are matching the query, it chooses the microservices in a cyclic order.

3. `RandomRobinLookupStrategy` is similar to `RoundRobinLookupStrategy`, but when the query returns multiple microservices, the result is picked randomly.

Lookup strategies use `Context` to find the microservices. As was previously mentioned `Context` contains shared configurations between providers and also created instances of the `ProvidingSilverService` which have methods to perform a search of local and remote microservices. The results from all providers are merged into a collection and returned to the lookup strategy.

It is possible to adjust lookup strategy for injection point by specifying `Invocation` annotation with `lookupstrategy` atttribute.

Listing 5.2: Injection point with RoundRobinLookupStrategy.

```
@Inject
@MicroserviceReference
@InvocationPolicy(lookupStrategy =
    RoundRobinLookupStrategy.class)
private OtherBean injectedBean;
```

Listing 5.2: Injection point with RoundRobinLookupStrategy.

5.1.3 Remote lookup

As was mentioned before, every instance of SilverWare has its registry which caches the results of previously discovered remote microservices and it is stored in `RemoteServiceHandlesStore`[15]. `RemoteServiceHandlesStore` is thread-safe storage that keeps a list of proxies for previously queried microservice metadata. When a remote SilverWare instance disconnects from the cluster, all proxies for the disconnected node are removed from the registry.

Before starting the actual lookup, the cluster provider checks whether the same metadata was not queried before and, in case they were, the

15. `io.silverware.microservices.silver.cluster.RemoteServiceHandlesStore`

query is sent just to a subset of nodes which were not marked as processed for the given metadata. Marking queried nodes implies that nodes are not queried multiple times with the same query. A search of remote microservice is done asynchronously, so it does not increase the delay of the invocation of microservice in case there were previous results. If it is the first lookup for the given metadata, the provider will wait for a short period of time for the results. The length of the time period in milliseconds can be set by parameter `silverware.cluster.lookup.timeout`.

On the remote node, the lookup is processed by `MicroserviceSearchResponder`[16] which executes the local lookup in SilverWare and creates `LocalServiceHandle`[17] for all results. Created handle is assigned a unique identification, is capable of invocation of methods on local microservice, and is stored in the context. The unique identification of `LocalServiceHandle` is then returned in a reply message.

The Silverware instance which established lookup for the microservice then processes the reply message containing node address and unique `LocalServiceHandle` identification. The result of processing is `RemoteServiceHandle`[18] which is a proxy capable of a remote invocation and is stored in `RemoteServiceHandlesStore`.

5.1.4 Remote invocations

When a `RemoteServiceHandle` is found in `DefaultMethodHandler`, the remote method is invoked. Invoking a method on a `RemoteServiceHandle` proxy creates a `MicroserviceRemoteCallRequest`[19] containing a unique `LocalServiceHandle` identifier, method name, method parameters and method parameter types. The message is then sent to address stored in a remote handle which means that all of the parameters must implement `Serializable`[20] interface.

---

16. `io.silverware.microservices.providers.cluster.internal.message.responder.MicroserviceSearchResponder`
17. `io.silverware.microservices.silver.cluster.LocalServiceHandle`
18. `io.silverware.microservices.silver.cluster.RemoteServiceHandle`
19. `io.silverware.microservices.providers.cluster.internal.message.response.MicroserviceRemoteCallRequest`
20. `java.io.Serializable`

On the remote instance of the SilverWare `MicroServiceRemoteCallResponder`[21] search the context for the handle and invoke method on found service. The result of invocation is sent as `MicroserviceRemoteCallResponse`[22] to the instance of SilverWare which was the source of invocation.

If exception is thrown during invocation, it is wrapped to `SilverWareClusteringException`[23] which adds a unique identification of the exception so it can be tracked in logs of both SilverWare instances. `RemoteServiceHandle` then logs `SilverWareClusteringException` and re-throw the previous cause of exception.

5.2 Versioning

In the previous sections, it was suggested that all messages sent via JGroups must be serializable, which causes version incompatibility. This problem is also directly related to parameter and result types of the microservices methods which are serialized and deserialized for purposes of remote invocation. The mechanism of microservice versioning, which allows specifying queries on the injection points and tagging microservice beans with version, is introduced in following sections and is supposed to solve this problem.

5.2.1 Serialization version problems

As it was previously mentioned, messages sent via JGroups must contain an array of bytes as content. Creating byte arrays is not a trivial task, and it is more straightforward for developers to work with classes. To simplify collaboration with JGroups, mechanism that allows automatic conversion of Java objects to an array of bytes is used and this mechanism internally uses Java serialization.

Java serialization was created to satisfy a need of saving Java objects in files or as blobs in databases. Java object serialization allows creating

21. `io.silverware.microservices.providers.cluster.internal.message.responder.MicroServiceRemoteCallResponder`
22. `io.silverware.microservices.providers.cluster.internal.message.response.MicroserviceRemoteCallResponse`
23. `io.silverware.microservices.providers.cluster.internal.message.responder.SilverWareClusteringException`

byte streams from classes automatically. Moreover, it is also able to recognize and load classes that match the class which was used to create byte stream. The format of a byte stream for a class is identified by Stream Unique Identifier (SUID) [25]. If it is not explicitly defined, it is computed as a hash of class attributes. When the SUID is expressly configured it means that programmer is aware that class is serialized, because serialization and deserialization are possible just for classes with the same SUID.

The changes to Java classes are divided into compatible and incompatible, and all are mentioned in *Versioning of Serializable Objects*[25]. The list of compatible changes includes [25]:

- adding or removing of `writeObject` and `readObject` methods which can add additional information to a serialized class and must call default serialization or deserialization methods,
- switching the field access type.

On the other hand, changing the fields, moving a class in a hierarchy or deleting fields breaks the compatibility of classes. The developer should keep track of changes and note when any incompatible changes occur because serialization would break. The remote invocation allows different versions of microservices running in a cluster and invokes the methods just on compatible versions without a need to specify SUID.

5.2.2 SemVer

Semantic versioning is a formal specification [26] describing how version changes of libraries or applications should be handled. The specification is written briefly, but it contains answers to many corner cases. The main idea can be reduced to following [26]:

1. Version should be in form of X.Y.Z Where X, Y and Z are the non-negative integers where
   (a) X is a major version,
   (b) Y is a minor version
   (c) and Z is a patch version.

2. Major version should be increased when the backward incompatible API change is introduced.

3. Minor version should be increased when the API is extended, or the backward compatible API change is introduced.

4. Patch version should be increased when the release contains bug fix and does not change API.

5. Specification allows build labels or pre-release tags to be appended after minor version (e.g., 1.0.0-SNAPSHOT or 2.0.0-BETA).

5.2.3 Version queries

Using SemVer makes the changes to versions clear. Developers can assume what changes are allowed for version updates. When all developers are aware of this, other mechanisms built on top of SemVer can be introduced. For example, commonly used Javascript package management tool NPM [27] allows specifying queries instead of explicit versions. All available repositories are then queried and the latest version of the library which satisfies the query is returned.

The grammar for queries is defined in git project readme [28] and allows basic comparison (e.g.,less, more, equal) joining of expressions with *logical or*, and more advanced features mentioned in table 5.1

The exception to the mentioned rules are versions with pre-release tags (beta, snapshot, ...). These versions are not considered as stable releases, so they do not satisfy queries without explicitly mentioning the pre-release tag. Explicitly mentioned pre-release tag moves the responsibility of using an unstable library to the developer.

5.2.4 SilverWare versioning

The versioning implemented as a part of this thesis is built on top of customized implementation of SemVer written in java which allows specifying of microservice versions as was explained in the previous section.

Name	Example	Description
X-range	*.X.x	Characters 'x', 'X' and '*' matches all versions.
Partial version	1	Missing version tags are treated as X-range characters.
Hyphen ranges	1.0-2.2	Hyphen range satisfies all version higher or equal to the first version specified and lower or equal to the second version. In this special case, the first version is partial, so it is interpreted as lowest possible (1.0.0) and the second version is partial, so it is interpreted as highest possible. As it is not possible to say what patch version will be the highest for current minor version rather 'lower than 2.3' operation is used.
Tilde range	~1.2	When a just major version is specified tilde operator allows minor and patch upgrades. When the minor version is also specified just, patch updates are allowed.
Caret range	^0.0.2	Caret ranges are similar to tilde ranges but treats differently the versions starting with zero. Versions updates are allowed just to versions which are right from the zero. • Versions with non-zero major version are allowed to do minor and patch updates. • Versions with zero major and non-zero minor version are allowed to do patch updates. • Versions with zero major and zero minor version are not allowed to do updates at all.

Table 5.1: Advanced SemVer queries [28].

Microservice bean version

The version of the microservice is specified by the annotation `MicroserviceVersion`[24] on the Java class as can be seen in the following listing.

Listing 5.3: `MicroserviceVersion` on microservice bean.

```
@MicroserviceVersion(api = "1",
   implementation = "1.3.4")
@Microservice
public class ReferencedBean{
   @Inject
   @MicroserviceReference
   private OtherBean injectedBean;
```

Listing 5.3: `MicroserviceVersion` on microservice bean.

Annotation `MicroserviceVersion` contains two fields. The first field holds API version and the second hold implementation version. Both fields are voluntary, but both versions must be resolved for beans. The simplest way of a specifying version of a bean is using the `MicroserviceVersion` annotation with both version specified directly on the bean class. The process of resolving version of the microservice take the version for the bean from different locations because `MicroserviceVersion` annotation is not mandatory. Lookup process continues the search in defined order till both versions are resolved. It is possible that API version is defined by another location than the implementation version.

Alternatives for defining version are analyzed in the following order and when a version was successfully resolved, rest of list is skipped.

1. `MicroserviceVersion` annotation on bean class,
2. `MicroserviceVersion` annotation on interfaces[25],
3. `MicroserviceVersion` annotation on parent classes hierarchy,

24. `io.silverware.microservices.annotations.MicroserviceVersion`
25. In case multiple interfaces are anotated by `MicroserviceVersion` annotation exception is thrown.

4. version from manifest file of classloader for the bean class.

Process of resolving versions for beans is executed during registration of beans in `MicroservicesCDIExtension` and all results are stored in `MicroserviceMetaData` objects in the `Context`. Best practice is to use the same version of microservices in one SilverWare instance because problems with different version can occur.

Injection point version

`MicroserviceVersion` can also be defined on the injection point. Defining version on injection point limits microservices which can be injected to this injection point.

Listing 5.4: `MicroserviceVersion` on injection point.

```
@Microservice
public class ReferencedBean{

  @MicroserviceVersion(implementation = "^1")
   @Inject
   @MicroserviceReference
   private OtherBean injectedBean;
```

Listing 5.4: `MicroserviceVersion` on injection point.

During the lookup, version from the injection point is in a role of a query and just microservices whose versions satisfy the query are resolved by lookup mechanism.The mechanism for checking whether a query satisfies a version is using implementation from *Java SemVer*[26] library. Both fields of `MicroserviceVersion` are voluntary and if they are not defined, then all versions of microservice can be injected to this injection point.

This mechanism allows multiple instances of microservice with the different version running in the same cluster and invokes methods on compatible versions. However, compatibility of microservices must be defined by developer in `MicroserviceVersion` annotation.

26. Available at `https://github.com/zafarkhaja/jsemver`

6 QuickStart

This chapter starts with a summary of the technologies that were used for the QuickStart. For an introduction of OpenShift, it is necessary to explain Kubernetes which is a premise of OpenShift. The Same situation applies to Kubernetes and Docker as Kubernetes uses Docker technologies internally.

Later in the chapter QuickStart architecture and features are described. QuickStart can be deployed to OpenShift or run locally and developed code is open-sourced and publicly available as part of the SilverWare quickstarts project [11].

6.1 Docker

Docker is an open-source platform that allows packaging of the applications into images, storing the images in the registry and deployment of created images into containers. Docker is developed by Docker Inc and released under Apache 2.0 license.

6.1.1 Virtualization

Virtual machines are the most common way how the virtualization was handled earlier. The problem is that virtual machines use hypervisor virtualization, where the code is executed virtually on physical hardware via an intermediation layer. This layer adds extra latency and overhead for the code execution which decreases the number of virtual machines which can run on one physical machine.

On the other hand, containers run in user space as a standard application and invoke functions of operating system kernel. Therefore, container virtualization skips the hypervisor and is often called operating system-level virtualization [29, p. 6]. Multiple containers can run on a single host because container instances are isolated. At the same time, running in the user space brings the drawback of lower flexibility because the operating system emulated in the container must have a similar structure to host operating system. In other words, it is not possible to run a Linux-based operating system on Windows operating system and vice versa. A workaround can be an emulating

of operating system in a virtual machine and running container as an application in the emulated system.

Another issue with containers is that they are considered less secure than virtual machines because sandboxing of software is not so advanced as for virtual machines. There are multiple approaches which are trying to solve this problem, and new implementation of containers are using advanced isolation techniques which are unique to the vendor. Docker developers have chosen modern Linux kernel features, such as control groups and namespaces, which allows great isolation with own network and storage stacks per container [29, p. 7]. Containers are not a new concept, but Docker simplified their creation management and automation which increased their usage.

6.1.2 Architecture

Docker platform is divided to Docker client and Docker daemon which fulfills server-client architecture. Daemon provides API for the binary client which is bundled with installation, and also REST API that allows remote access. The architecture consist of the following components:

1. *Dockerfile* is the source code of the image that specifies: file content of the image, required software, environment variables, network and file system settings. The main advantage is that *Dockerfile* allows specifying patch versions for the applications and operating system.

```
FROM java:7
COPY . /usr/src/myapp
WORKDIR /usr/src/myapp
EXPOSE 8080 8778
RUN javac Main.java
CMD ["java", "Main"]
```

Listing 6.1: Example of *Dockerfile*.

2. *Docker images* are the basic building blocks because they are built from *Dockerfiles* and deployed to containers. Images use

Union file system which allows step-by-step description of created image. For every step, there is a new layer created in Union File System and the final image contains all layers.

3. *Registries* are stores for the created images. Registries are similar to the Maven repositories because every machine running Docker contains private repository for storing images built locally or caching downloaded images. Official public Docker registry is called *Docker Hub* and after registration, it allows sharing own custom images. It also contains images officially released and supported. In case that company required the images to be private and locally hosted it is possible to host own private registry.

4. *Containers* are the execution environments of the docker architecture. Container is launched from image, can contain multiple running processes and is defined by:

   (a) image format,

   (b) execution environment and

   (c) standard set of operations [29, p. 12].

The fundamental difference between virtual machines and containers is a level of abstraction. While the hypervisor abstracts an entire device for virtual machines, containers abstract just the operating system kernel. Another difference is that multiple containers from the same image can be started at the exact moment without a need to write anything because containers use copy-on-write mechanism.

Docker treats software containers like standard transportation shipping containers in the real world. The container contains packaged application, but Docker does not need any information about content because the container is considered a standalone package that provides a set of operations for changing the state of the container. These operations include creating, starting, stopping and destroying. Also, containers are stackable, portable and interchangeable because they should be as generic as possible. The last similarity with real world containers is that containers can be shipped to diverse locations. It is plausible to set up a local development environment that is later

promoted to testing environment and even production with same Docker image [29].

6.1.3 Features

From information in previous paragraphs, it is clear that *Docker* containers are a great demonstration of Phoenix architecture and they also encourage microservice architecture. It is recommended to run a single application or process for each container, which favors a distributed application model. The difference is that distributed application is represented by interconnected containers which package the applications instead of just applications. However, the packaging of applications makes it easy to distribute and scale components of distributed application.

Another advantage is that developers now can tune execution of an application in a container and be sure that it will perform in the same manner everywhere else. The consistency between development and production environments requires less effort for deployment of a new version and can shorten the release cycle. Docker uses copy on write model so when a change is needed it can be deployed quickly by changing the file in Docker Image.

6.2 Kubernetes

Kubernetes is an open source system for automating deployment, scaling, and management of containerized applications across cluster nodes. Google developed it and released under Apache 2.0 license. The main goal of a project is to simplify using of clustered infrastructure for deployment of applications and hide the complexity of this system. Administrator of such environment should not need to know anything about the infrastructure and should just schedule work and Kubernetes takes care of execution. Kubernetes uses Docker images and containers for scheduling the work.

6.2.1 Components

This section introduces Kubernetes components knowledge of which is necessary to understand how the QuickStart deployment to OpenShift works.

- Pod is the smallest manageable unit in Kubernetes. It can contain one or more Docker containers which represent a single application and share same IP address and storage. The reason for support of multiple containers are tightly coupled containers (e.g., monitoring running in another container than application).
- *Service* as a term was used on multiple occasions in this thesis, but service in Kubernetes nomenclature means a single point of access with stable IP address for a group of pods. Pods can be scheduled on other node or destroyed, and their IP addresses are not stable, so service takes a role of the load balancer and abstracts pods to more solid object.
- *Label* is a concept which is used for coordination of resources in Kubernetes. Labels are key-value objects which are used as tags on resources and as selectors for logical sets of resources. For example, service contains selector for label `name` with value `app` and all pods which are tagged with mentioned label belongs to the service.
- *Namespace* allows dividing clusters resources between multiple users and separating scope of their resources. For instance, name of the service must be unique in a namespace and it is possible to configure different access rights for namespaces.
- *Service account* identifies applications running in Pods. It is possible to add roles or access rights to service account so an application running in pods can access Kubernetes API.

6.2.2 Architecture

Kubernetes cluster consists of master and nodes. Master contains components which manage Kubernetes nodes and schedules containers

which run on nodes. Master is a single point of failure, although cluster survives a failure of a master with losing part of the functionality. In order to prevent failures, it is possible to run multiple masters for high availability [30].

Master

Master contains following components:

- *etcd* is lightweight distributed persistent key-value data store developed by CoreOS. The main instance is running on the master, and other nodes are watching changes and reacting to them [31].
- *API server* exposes REST interface for manipulating Kubernetes cluster and validates received queries and commands. It checks whether information for pods stored in *etcd* are valid and current. In case there is inconsistency it updates pods. Also, it can recreate pods on another cluster node.
- *Controller manager* is periodically watching changes in *etcd* storage and performs changes which eventually bring current state to the desired state stored in *etcd*.
- *Scheduler* keep track of all nodes and their workloads. It is also responsible for scheduling work on nodes which are suitable and with minimum load. Scheduler service was designed for easy extensibility [32] and also considers policies in the decisions.

Nodes

The nodes are hosts incapable of self-existence which are watching *etcd* for changes and adjust to them. They need access to shared Docker network so the containers can communicate together and contain following components:

- *Kubelet* is agent application running on each node [32]. It interacts with master and retrieves manifest file. The manifest file is in JSON or YAML format, and it specifies a state of a node

which is expected by the master (e.g., number of pods and all their settings). Kubelet is responsible for executing commands which result in the state described in manifest.

- *Proxy* can do simple TCP and UDP stream forwarding of services defined in Kubernetes. In other words, it allows using services for external communication and forwards communication to corrects Pods.

6.2.3 Features

Kubernetes is failure resistant because when a cluster node dies all, pods from the node are rescheduled to other nodes. It also allows to define *health checks* for the Pods which inform Kubernetes about a state of the Pod. Kubernetes automatically redeploy Pods which do not respond to the defined *health checks*. Another feature of Kubernetes does not expose Pods to traffic till *health checks* reports that it is ready [32].

Kubernetes also allows simple scaling of application to multiple instances: manually via API or automatically by monitoring resources of the application.

6.3 OpenShift

OpenShift is PaaS[1] from Red Hat built on Kubernetes [33] with the aim to simplify installation and provide more functionality.. For example, it is not necessary to manually install each component for master and nodes because there are installation packages available. There are also docker images accessible for an easy trial of features. OpenShift also add some features and components which are not present in Kubernetes, specifically:

- *Routes*[2] describe how the services should be exposed to the internet. It is possible to use a hostname instead of IP addresses, and DNS record should point to the node which is running

1. Platform as a Service is a category of cloud computing services.
2. Ingress is similar concept for Kubernetes but was introduced later. OpenShift still use Routes.

Router. *Router* is a component of OpenShift which enables external communication for created Routes.

- *Build* represents process of transformation input parameters to a image. For example input parameters can be a location of a source code and image capable of building provided code to Docker image. There are few build types mentioned in the documentation [33] which cover most of the use cases, and it is also possible to create custom build type. All parameters of build process are stored in *BuildConfig* which allows repeatable builds from the same source (e.g., GitHub repository).

- *Imagestream* groups related Docker images by label selectors to one consistent view. It allows to group builds of the same application and automatically update existing application pods to a newer version when available.

6.4 Deployment to cloud

Fabric8 (read as fabricate) maven plugin is part of Fabric8 platform which promotes Kubernetes and OpenShift usage by creating tools which simplify interaction with these services for developers [34]. Subsequent paragraphs are addressed on interaction with OpenShift because it was used as a demonstration platform.

For instance `fabric8-maven-plugin` (from now on just FMP) handles deployment to OpenShift in a similar fashion as other maven plugins treat deployment to a locally running server. It generates necessary OpenShift resources, build Docker image. and applies changes to a local or remote instance of OpenShift so the developer unaware of the plugin may not even notice that application is running on OpenShift [34]. FMP integration with OpenShift is done by a lookup for OpenShift client binaries and retrieving login credentials and current namespace from a configuration file.

FMP is configurable and allows many customizations of deployment which can be defined in plugin configuration in POM or by creating partial resources which are bundled together and form final OpenShift resources. Following operations with applications running

on OpenShift are implemented in FMP: debugging, tailing logs, scaling, and automatic redeployment [34].

6.5 JGroups for Kubernetes

Clustering for SilverWare is implemented with JGroups. It was mentioned before that JGroups does not support cloud services natively and needs additional library which enables service discovery. For Kubernetes the library is called simply kubernetes[3] and is part of JGroups extras.

To enable JGroups discovery for Kubernetes,the following steps were done:

1. Add library with `KubePing` to the classpath so JGroups can use it. In the case of SilverWare, it was a trivial task, because adding the library as maven dependency makes it part of the deployment.
2. Change of configuration of discovery protocol in JGroups to `KubePing`.

   Listing 6.2: JGroups configuration of KubePing.

   ```
   ...
   <kubernetes.KUBE_PING/>
   ...
   ```

   Listing 6.2: JGroups configuration of KubePing.

3. Set environment variables for Pod which runs clustered application with KubePing. It is possible to set following properties:
   - `OPENSHIFT_KUBE_PING_NAMESPACE` – namespace which should be used for Pod discovery.
   - `OPENSHIFT_KUBE_PING_LABELS` – limitation of Pods which could be discovered[4]. Default allows discovery of all pods.

3. Available at `https://github.com/jgroups-extras/jgroups-kubernetes/tree/0.9.1`
4. Labels were mentioned in section 6.2.1.

- `OPENSHIFT_KUBE_PING_SERVER_PORT` – port of server which communicate with Kubernetes API. By default it is 8888.

4. Expose additional Pod ports which are used for service discovery and communication of JGroups.

5. Allow service account communication with Kubernetes API.

First two steps are same also for other cloud providers, but the rest may vary as an implementation of discovery protocols is using vendor specific features.

6.6 QuickStart

The QuickStart application is available in SilverWare quickstarts [11] in `openshift-base` directory, and it contains three modules. The `openshift-core-module` is a dependency for other two modules and contains SilverWare health check implementation and API classes. `openshift-core-module` is not executable as a standalone application, because its purpose is to include shared functionality of the other two modules.

For presentation purposes, there is a `openshift-gateway-module`, which exposes REST endpoint, but does not contain any business logic. All REST requests invoke methods on injected CDI beans managed by SilverWare. An interface of these beans is in `openshift-core` module and implementations are in the `openshift-cluster-worker` module. There are two implementations of the interface with different `MicroserviceVersions`. These beans are injected to two different injection points in REST endpoint.

As it was mentioned before, quickstart can be deployed to OpenShift or executed locally. These options are covered by the following two Maven profiles, which are present in the base POM:

1. `openshift` profile uses FMP for deployment of modules to OpenShift. FMP configuration contains setting of custom Docker Image that contains SilverWare configuration files, which enable `KubePing` service discovery. Also, not all OpenShift resources are generated automatically, and `Route` is provided as OpenShift resource fragment. After a successful deployment, `openshift-cluster-worker` is scaled to three instances.

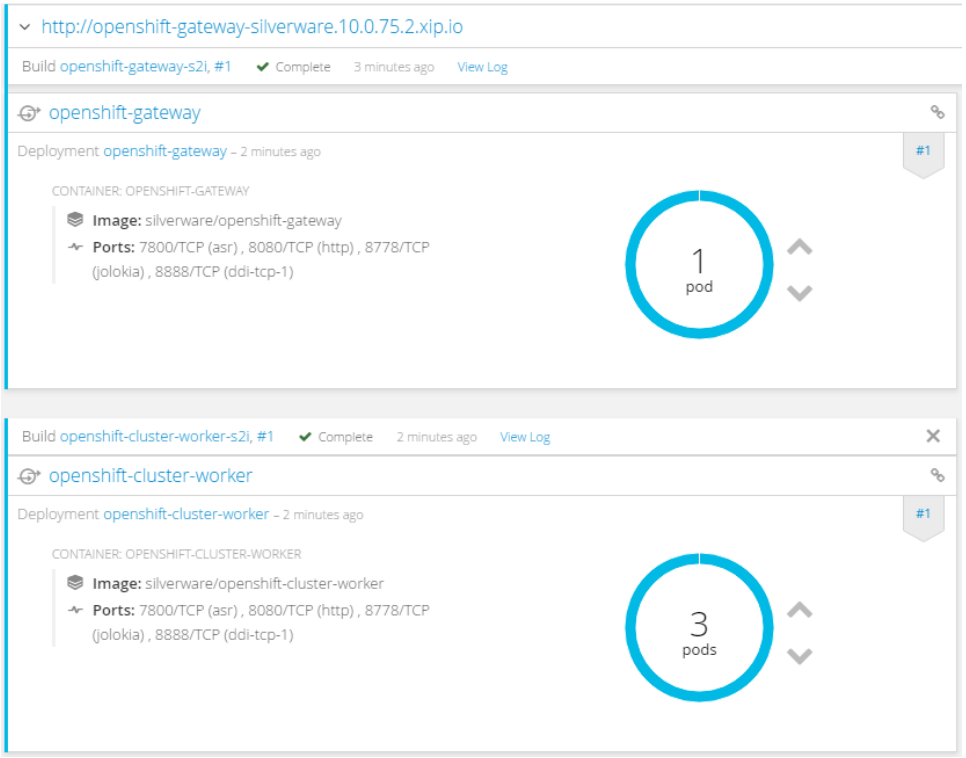

Figure 6.1: QuickStart deployment to OpenShift

2. `standalone` profile builds modules `openshift-cluster-worker` and `openshift-gateway-module` to executable `jars` with external dependencies. Built jars[5] can be run on Java Virtual Machine and form a cluster automatically.

Quickstart demonstrates different invocation policies, versioning, and remote invocation. Number of running `openshift-cluster-worker` instances can be changed, and clustering implementation is capable of reacting to cluster evolution. All instructions for deployment and information about demonstrated functionality is available in a readme file of the `openshift-base` project.

5. Before starting multiple `openshift-cluster-worker` instances, be aware that REST port configuration must be changed.

7 Conclusion

The primary goal of this Master's thesis was an implementation of service discovery and remote calls between CDI components in the microservice platform SilverWare. This functionality was implemented utilizing JGroups messaging framework.

Service discovery is delayed to the last possible moment, which saves network traffic but slows down the first invocations of service. After the first invocation lookup results are cached locally and the latency is increased just for remote invocations of CDI components. All objects that are part of remote call must be serializable, which limits possible microservices and developer must be aware whether his\her microservice supports remote invocation. The contract of supporting remote invocations is defined on injection point by changing `InvocationPolicy`. Integration tests of functionality were performed manually, and implementation is capable of reacting to cluster evolution.

Versioning of microservices is another enhancement for service discovery introduced as a part of this thesis. The versioning allows running multiple instances of microservice with the different versions in the same cluster. Injection points can limit microservices that can be injected by specifying version queries. For specifying the queries, a grammar based on SemVer [26] was chosen. This grammar was spread by NPM [27] and has been broadly accepted by the community as a solution for version specification. Integration tests for versioning are part of the SilverWare build and are automatically triggered for every pull request or commit to SilverWare GitHub repository [10].

All implemented features were demonstrated in QuickStart that can be deployed to OpenShift or executed locally. The deployment to OpenShift illustrates that using Silverware in a cloud environment is straightforward and extend the potential of the adopted technologies. Supported platforms for service discovery and remote invocations are limited by JGroups, whose vibrant community expands support to many modern cloud platforms. JGroups integration means that SilverWare can be considered as cloud capable framework.

Performance tests have not been executed, but they would not have significant informative value, as a performance of implemented solu-

tion depends on: network conditions, JGroups settings, and message size. If integration test will be performed in the future, all of these factors should be examined in different combinations.

Future intentions of SilverWare are specified in project readme [10] and soon third major version should be released. Also, security and transactions integrations are planned for near future. Another possible future extension can implement different approach of remote method invocation, which would allow cross-language invocations. Currently, SilverWare is strongly biased for Java.

Bibliography

[1] InternetLiveStats. *Internet Users*. [online]. 2016. URL: http://www.internetlivestats.com/internet-users/ (visited on 12/29/2016).

[2] C. Richardson. *Introduction to Microservices*. [online]. May 2015. URL: https://www.nginx.com/blog/introduction-to-microservices/ (visited on 10/20/2016).

[3] S. Newman. *Building microservices: designing fine-grained systems*. Sebastopol, CA: O'Reilly Media, 2015. ISBN: 978-1491950357.

[4] J. Lewis and M. Fowler. *Microservices*. [online]. Mar. 2014. URL: http://www.martinfowler.com/articles/microservices.html (visited on 23/10/2016).

[5] C. Richardson. *Pattern: API Gateway*. [online]. 2014. URL: http://microservices.io/patterns/apigateway.html (visited on 10/30/2016).

[6] Robert C. Martin. *Snowflake Servers*. [online]. 2012. URL: https://dzone.com/articles/martin-fowler-snowflake.

[7] Robert C. Martin. *Phoenix Server*. [online]. 2012. URL: http://martinfowler.com/bliki/PhoenixServer.html.

[8] Thomas Erl. *SOA with REST : principles, patterns & constraints for building enterprise solutions with REST*. Upper Saddle River, N.J: Prentice Hall, 2012. ISBN: 978-0137012510.

[9] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002, pp. 95–98. ISBN: 0135974445.

[10] SilverThings community. *SilverWare*. [online]. 2016. URL: https://github.com/SilverThings/SilverWare (visited on 12/29/2016).

[11] SilverThings community. *SilverWare Examples and Demonstrations*. [online]. 2016. URL: https://github.com/SilverThings/SilverWare-Demos (visited on 12/29/2016).

[12] H. Braun and D. Andreadis. *"Right Size" your Services with WildFly Swarm by Heiko Braun/Dimitris Andreadis*. [online]. 2016. URL: https://www.youtube.com/watch?v=S8L9qTaE07Y (visited on 11/12/2016).

[13] H. Braun and D. Andreadis. *Cloud Native Java by Josh Long*. [online]. 2016. URL: https://www.youtube.com/watch?v=JDcl4kT6Qmo (visited on 11/12/2016).

[14] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: http://research.google.com/archive/papers/dapper-2010-1.pdf.

[15] G. Hohpe and B. Woolf. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. Boston: Addison-Wesley, 2004. ISBN: 978-0321200686.

[16] G. Hohpe and B. Woolf. *Enterprise Integration Patterns - Solving Integration Problems using Patterns*. [online]. 2003. URL: http://www.enterpriseintegrationpatterns.com/patterns/messaging/Chapter1.html.

[17] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. ISBN: 978-0201633610.

[18] B. Ban. *Reliable group communication with JGroups*. [online]. Oct. 2016. URL: http://jgroups.org/manual/index.html (visited on 10/07/2016).

[19] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[20] Bela Ban and Vladimir Blagojevic. *Reliable group communication with JGroups 3.x*. [online]. 2013. URL: http://www.jgroups.org/manual/html/user-advanced.html (visited on 12/24/2016).

[21] Galder Zamarreno Brian Stansberry and Paul Ferraro. *JBoss AS 5.1 Clustering Guide*. [online]. 2010. URL: https://docs.jboss.org/jbossclustering/cluster_guide/5.1/html/jgroups.chapt.html (visited on 12/24/2016).

[22] B. Ban. *Reliable group communication with JGroups*. [online]. 2015. URL: http://jgroups.org/manual4/index.html (visited on 18/10/2016).

[23] Red Hat. *Contexts and Dependency Injection for the Java EE platform*. [online]. 2014. URL: http://docs.jboss.org/cdi/spec/1.2/cdi-spec.html (visited on 12/02/2016).

[24] S Krupa T. Livora. *CDI Microservice Provider*. [online]. 2016. URL: https://github.com/SilverThings/SilverWare/wiki/CDI-Microservice-Provider (visited on 12/02/2016).

[25] Oracle and/or its affiliates. *Versioning of Serializable Objects*. [online]. 2010. URL: http://docs.oracle.com/javase/8/docs/

platform / serialization / spec / version . html (visited on 12/16/2016).

[26] T. Preston-Werner. *Semantic Versioning 2.0.0*. [online]. Oct. 2016. URL: http : / / semver . org / spec / v2 . 0 . 0 . html (visited on 10/07/2016).

[27] NPM. *The semantic versioner for npm*. [online]. 2016. URL: https : //docs.npmjs.com/misc/semver (visited on 12/29/2016).

[28] I. Z. Schlueter. *semver(1) – The semantic versioner for npm*. [online]. July 2016. URL: https://github.com/npm/node-semver (visited on 11/10/2016).

[29] James Turnbull. *The Docker Book*. [online]. 2016. URL: https : // www.dockerbook.com/ (visited on 11/26/2016).

[30] Red Hat. *Kubernetes Infrastructure*. [online]. 2016. URL: https : // docs.openshift.org/latest/architecture/infrastructure_ components / kubernetes _ infrastructure . html (visited on 11/28/2016).

[31] Justin Ellingwood. *An Introduction to Kubernetes*. [online]. 2016. URL: https://www.digitalocean.com/community/tutorials/ an-introduction-to-kubernetes (visited on 11/28/2016).

[32] Kubernetes Authors. *Admin Guide*. [online]. 2016. URL: http : //kubernetes.io/docs/reference/ (visited on 11/28/2016).

[33] Red Hat. *OpenShift Origin Latest Documentation*. [online]. 2016. URL: https : //docs . openshift . org/latest/architecture/ core_concepts/index.html (visited on 11/28/2016).

[34] James Strachan Roland Huß. *fabric8io/fabric8-maven-plugin*. [online]. 2016. URL: https://maven.fabric8.io/ (visited on 12/29/2016).

A Content of the Attachment

1. **Directory SilverWare:** source code of SilverWare. Module `cluster--microservice-provider` was implemented in this thesis with many changes in other modules. All changes are tracked on SilverWare GitHub project [10].
2. **Directory SilverWare-Demos:** source code of SilverWare Quick-Starts. Module `openshift-base` was implemented in this thesis