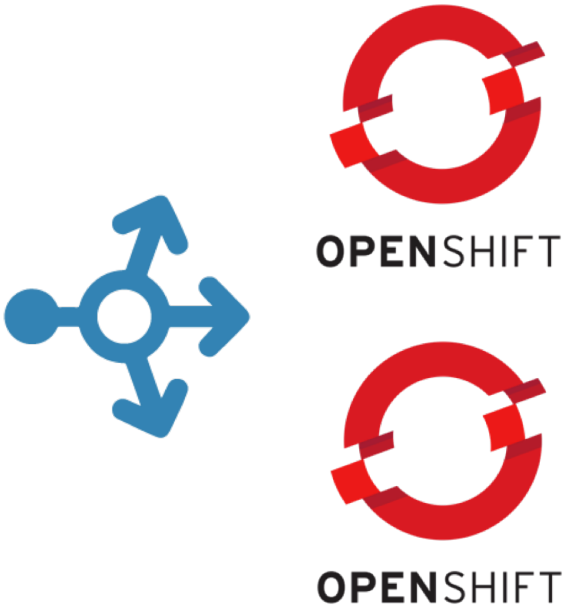
Berner
Fachhochschule

Smart cross cluster load balancing for OpenShift

Master Thesis



Study Path:	MAS-IT Enterprise Application Development (MT-HS17)
Author:	Lehmann Reto, reto.lehmann@me.com
Principal:	Schweizerische Bundesbahnen SBB
Guardian:	Oswald Baltisar, SBB, baltisar.oswald@sbb.ch
Expert	Eyup Koc, Redhat, eykoc@redhat.com
Date:	February 26, 2018

1 Management Summary

Swiss Federal Railways operates more than 750 applications on a container platform called OpenShift. As this container platform contains business critical applications - like the main mobile website or the backend of the main mobile app – the platform has an availability goal of 99.99%. The experience of operating the productive platform for two years showed that the main risk for unplanned downtime, is maintenance at the platform itself. Ahead of this master thesis, Swiss Federal Railways had an idea of how to address this problem. To verify this solution idea is the goal of this master thesis.

The solution idea claims that in the existing OpenShift setup, only one piece is missing in order to achieve risk free cluster maintenance. The missing component is a smarter load balancer. In the following thesis, the solution idea will be tested using a prototype of that smarter load balancer. At the beginning the problem was analyzed in detail. Then the existing architecture and source code of the OpenShift platform was evaluated and assessed. Out of this information a solution architecture concept was created and visualized. Finally, the prototype was implemented in order to test the ideas premise and the designed solution architecture.

The results of this master thesis are based on functional and load oriented test cases of the prototype. The test results showed that the solution idea does work out. **OpenShift cluster maintenance is possible without risk and downtime**. The test results also revealed one performance limitation of the created prototypical implementation.

The findings of this master thesis are reported back to Swiss Federal Railways. There the solution architecture may be adapted to an existing load balancer. Furthermore, the results of this thesis will be communicated to Redhat, the vendor of the OpenShift container platform.

Table of Contents

2 Introduction

Swiss Federal Railways operates more than 750 of their business applications based on container technologies [1]. These containers run on a platform called OpenShift [2].

Swiss Federal Railways started operating containerized applications in 2015. Within two years the adoption rate of the new platform increased up to 50%. Today this includes several business critical applications, like the backend systems for the main mobile app, the main website as well as multiple systems that operate all points of sales. As trains operate 24/7, those applications must be available at all time. Therefore, stability and availability are the main goals for the container platform.

To achieve those goals, everything is set up scaled and redundant. This begins with redundant data centers and basic infrastructure like network and internet access. It continues up to the application level where OpenShift provides tools to operate application containers horizontally scaled [3]. Applications can be scaled to be high available and also to handle increased load.

The platform itself is also based on the principles of scalability and redundancy and is thus, very resilient to outages and changes to the topology during normal operation. Unfortunately, this is not the case during updates of the platform software. This is currently **the biggest risk** to the above mentioned goals. Swiss Federal Railways had an idea of how to address those problems. During this master thesis those problems will be analyzed and an architecture concept, based on Swiss Federal Railways solution idea, will be build. Then the solution idea will be tested using a prototype based on the architecture concept.

3 Distinction

This document and its figures hide a lot of details, components and layers of the OpenShift platform. Only the parts that are relevant for the subject are illustrated and discussed. Furthermore, only the traffic that flows from external systems to the OpenShift platform is considered. Internal traffic is not discussed.

[1] (Docker, 2017)
[2] (Redhat, 2017)
[3] (Gartner, IT-Glossary: Scalability, 2017)

4 Initial position

To start out, the initial position of Swiss Federal railways is shown. Swiss Federal Railways uses OpenShift, which is based on Googles kubernetes [4], as orchestrator of their containers. OpenShift is responsible for the containers and their relationship with each other and their connections to outside of the platform.

This orchestration includes the following tasks [5]:

- Build and package applications as containers
- Start, run, stop containers
- Make sure that the application is healthy and available
- Deploy new versions of the applications without downtime
- Provide a virtual network for the containers
- Manage which containers can communicate with each other
- Manage traffic to and from the outside of the platform

The following figure shows the OpenShift setup at Swiss Federal Railways on a high level:

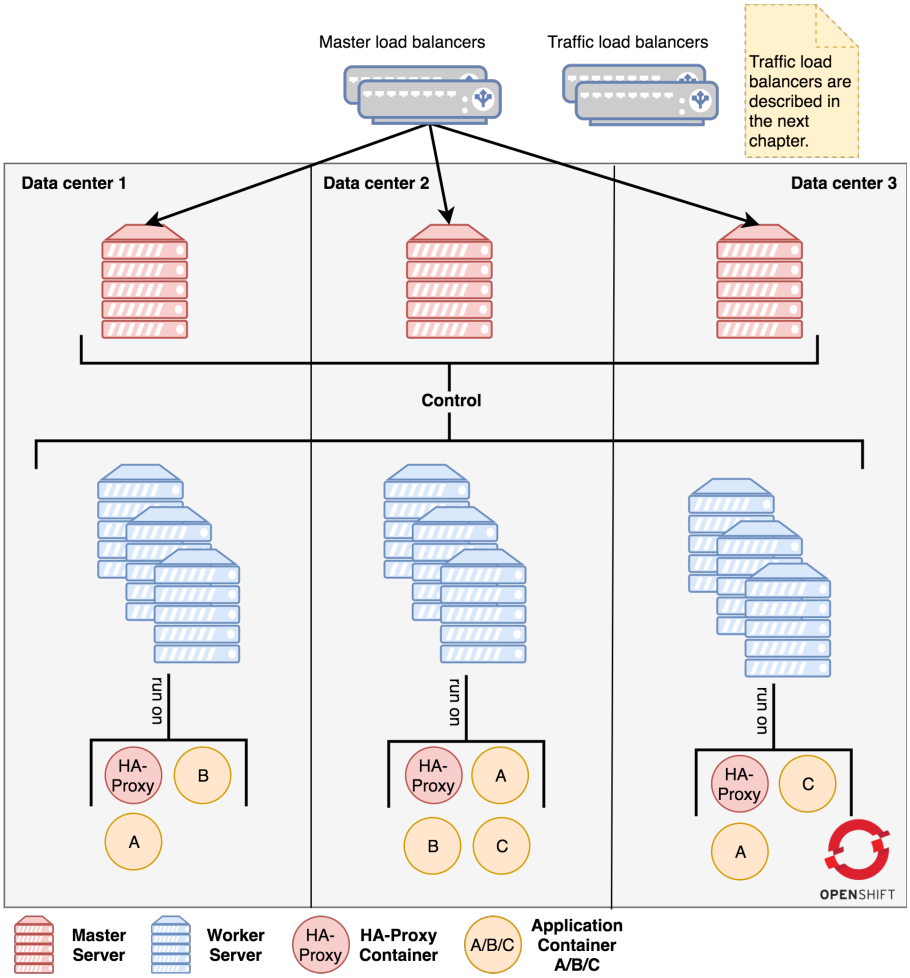


Figure 1: OpenShift Setup at Swiss Federal Railways.

[4] (Kubernetes, 2017)

[5] This is not a complete list.

The OpenShift setup consists of the following components:

Component	Description
Master Server	- Controls and manages the worker servers. - Assigns containers to worker servers. - Provides a graphical user interface (GUI) for users. - Provides an API.
Worker Server	- Runs application containers. - Is controlled by the master server.
Master load balancers	- Balances the API and the GUI across all the master servers. [6]
Traffic load balancers	- Balances the incoming web traffic for all applications to the worker nodes. More details see "Figure 2: OpenShift application traffic overview."
Containers	- A package format and runtime for an application. - In this diagram it is an example of three applications, that are deployed on different servers and data centers.
HA-Proxy	- Runs as a container on a worker server. - Is a component of OpenShift that balances requests to the corresponding application container.

Table 1: High-level components of OpenShift.

This master thesis will focus on the **application traffic load balancing**. This is where the communication between a user and an application flows along. The following figure provides further details on the traffic flow in Swiss Federal Railways OpenShift setup:

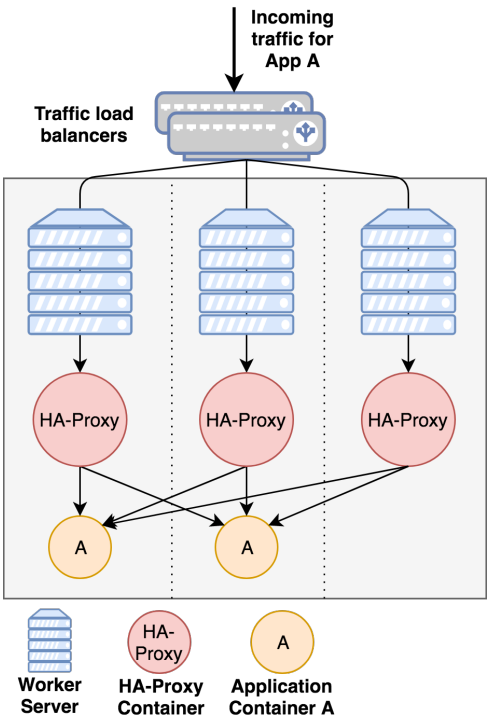


Figure 2: OpenShift application traffic overview.

[6] For more information about Load-Balancers see Glossary.

5 Problem description

5.1 Preface

As described in the introduction, Swiss Federal Railways operates more than 750 applications on the OpenShift platform. This also includes several business critical applications with a very high availability requirement as some of them are used 24/7. This results in a very high availability of the whole platform and all its services. The availability goal is 99.99%, which **results in a maximal unplanned downtime of 52 minutes a year** [7].

As an application platform OpenShift supports these requirements by its architecture of scalability. Everything in the platform can be scaled, which introduces redundancy at every level. In Swiss Federal Railways setup every component (e.g. Master servers, worker servers, ha-proxies, load balancers in front of the OpenShift cluster) is scaled. Also the basic infrastructure, like data centers and networking infrastructure in between them, is redundant.

The same principle applies to the applications themselves. All modern applications are built stateless and scaled [8]. A developer specifies how many instances of the application are needed at all times (this is called Replicas). OpenShift then takes care that the specified replicas of the application are running at all time. If an instance of the application crashes, OpenShift takes care of starting a new instance automatically and also takes care of routing the incoming traffic to the new instance.

5.2 Problem: Maintenance on the running platform

As described before, the platform provides the tools and architecture to support the availability goal of the business critical applications of Swiss Federal Railways. The architecture outline even enables platform server maintenance without application downtime, as the same principles are used. A typical update procedure of the OpenShift platform and its components looks like this:

Server maintenance

- To start maintenance on a specific server, the server is set to maintenance mode. This way no new containers will be started on that server.
- If there are still containers running on the server, they are stopped. OpenShift then takes care to start the containers on another worker server.
- The server is removed from all load balancers.

[7] (Wikipedia, High-Availability, 2017)
[8] Based on 12-Factor Apps: (12-Factor-Apps, 2017).

- The maintenance is done by the operations team.
- The maintenance mode of the server is terminated and new containers are started on the specific server.

Component maintenance

All the OpenShift components are scaled. A new version of the component is installed one instance after the other. For example a version update of the OpenShift configuration database (called etcd [9]) would look like this:

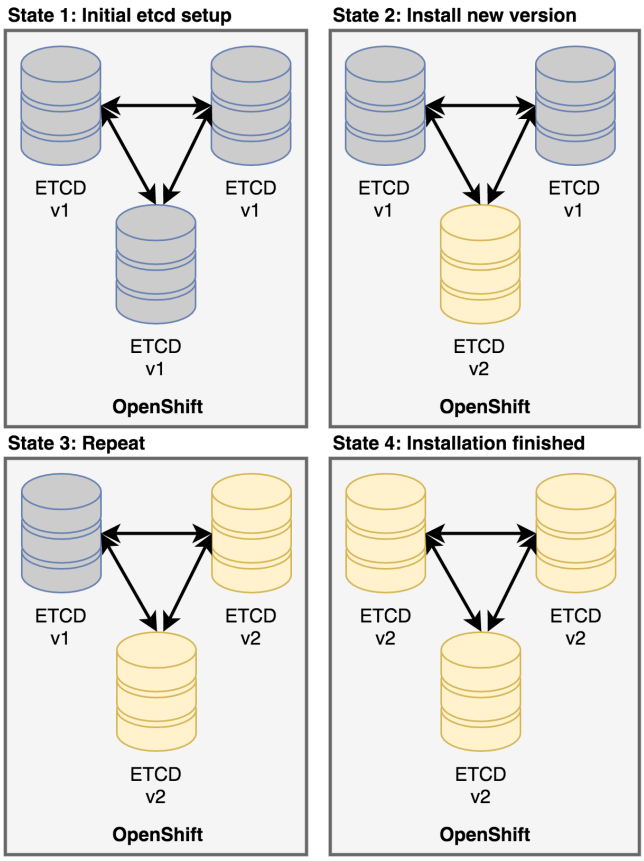


Figure 3: Update process of ETCD.

Basically it is possible to install new versions of OpenShift during full operation of the OpenShift cluster and applications that are running on the cluster. Unfortunately, the experience of the last two years of operating OpenShift showed that **those update were the main cause for incidents and problems**.

To update a OpenShift cluster during full operation introduces a big risk for the availability goal (see “5.1 Preface”) of Swiss Federal Railways. An extract of three experienced situations that caused incidences, which reduced the availability of the OpenShift platform, will be described in the the following subchapters.

[9] (Coreos, 2017)

5.2.1 Problems with the OpenShift platform software

The first experienced incident originated in the platform software. A new version of the OpenShifts components (e.g. master services, worker services, ha-proxy) had software problems (bugs). The new version of the components was installed on the productive cluster and caused incidents. There are several stages of testing before installing a new version to the production cluster, but these issues were not found upfront. The following obstructive conditions were identified afterwards:

- The issues only occurred at a certain load (overall container count, overall traffic per second).
- The issues only occurred after a certain time of heavy operation.
- The issues only occurred after a certain amount of user interaction (e.g. application deployments).
- A combination of the above.

5.2.2 Issues during the rollout

Another experienced incident happened during the rollout. Every operation that changes the platform is fully automated with Ansible [10]. This way the update can be tested on a non-productive environment and then applied the same way on a productive system. Even with that setup in place, there were incidents caused by:

- Errors in the automation that were not found on the non-productive environment.
- Applying the automation to too many servers at once, this way the platform had a resource bottleneck.

[10] (Ansible, 2017)

5.2.3 Update of components without downtime not possible

The last example of experienced incidents concerns the update process. As described in chapter "5.2 Problem: Maintenance on the running platform", updates of OpenShift components are normally realized during full cluster operations. During the update the component works at all time. Swiss Federal Railways experienced a few cases where this update procedure could not be applied. Here a few examples:

Component	Problem & Cause
Update of OpenShifts internal configuration database (called etcd).	The database was updated from one a major version to another major version. The versions were not compatible. Thus it was not possible to update the instances one after another. They had to be restarted all at the same time, which resulted in a **downtime of all OpenShift operations for a few seconds**.
OpenShift communication of all services.	OpenShift uses encrypted communication for all its components. Swiss Federal Railways had to change its whole certificate chain and replace every client certificate in all the components [11]. When a whole certificate chain is changed, all the components have to be restarted the same time. This resulted in a **downtime of all OpenShift operations for a few minutes**.
Introduction of a new component (DNS cache [12]) on the master servers to increase performance of DNS.	A new version of OpenShift brought a new component to the master servers. It was not possible to install this component one master server after another as the installation influenced a global configuration. This configuration had to be applied to all the master servers at the same time. This resulted in a **downtime of all OpenShift operations for a few seconds**.

Table 2: Cases of a cluster version rollout with downtime.

[11] (Wikipedia, Public key infrastructure, 2017)
[12] (thekelleys, 2017)

6 Course of action

An idea of how to resolve the problem, described in chapter “5 Problem description”, was already present at the beginning of this thesis. Out of this idea, the goal for this thesis was formed. The goal is to create an architecture concept, then create a prototype to test the idea. The following table shows the course of action for this thesis:

Step	Description
Preparation	To start out, first the existing code of the OpenShift platform has to be analyzed. The following tasks were planned: - Analyze existing code and object definitions. - Decide how and where OpenShift could be extended.
Concept	In the next step, the architecture concept should be created out of the learning of the preparation step. The following tasks were planned: - Create high level solution approach. - Create architecture & communication design. - Record business & technical requirements.
Implementation	After the concept step, a prototype should be implemented based on the architecture concept. The prototype should fulfil the requirements and is tested using function & load tests. The following steps were planned: - Implement a prototype to proof the idea and concept. - Test functionality of the implementation. - The the implementation with load tests. - Document the implementation & test results.
Conclusion	To finish up, a conclusion of the learnings should be written. Also the initial thesis premise should be validated.

Table 3: Course of action.

6.1 Methodology

To track the defined tasks and steps the scrum [13] methodology was used. The guardian of this thesis was the product owner. The following task board was used to track and report the progress to the product owner and the master thesis expert:

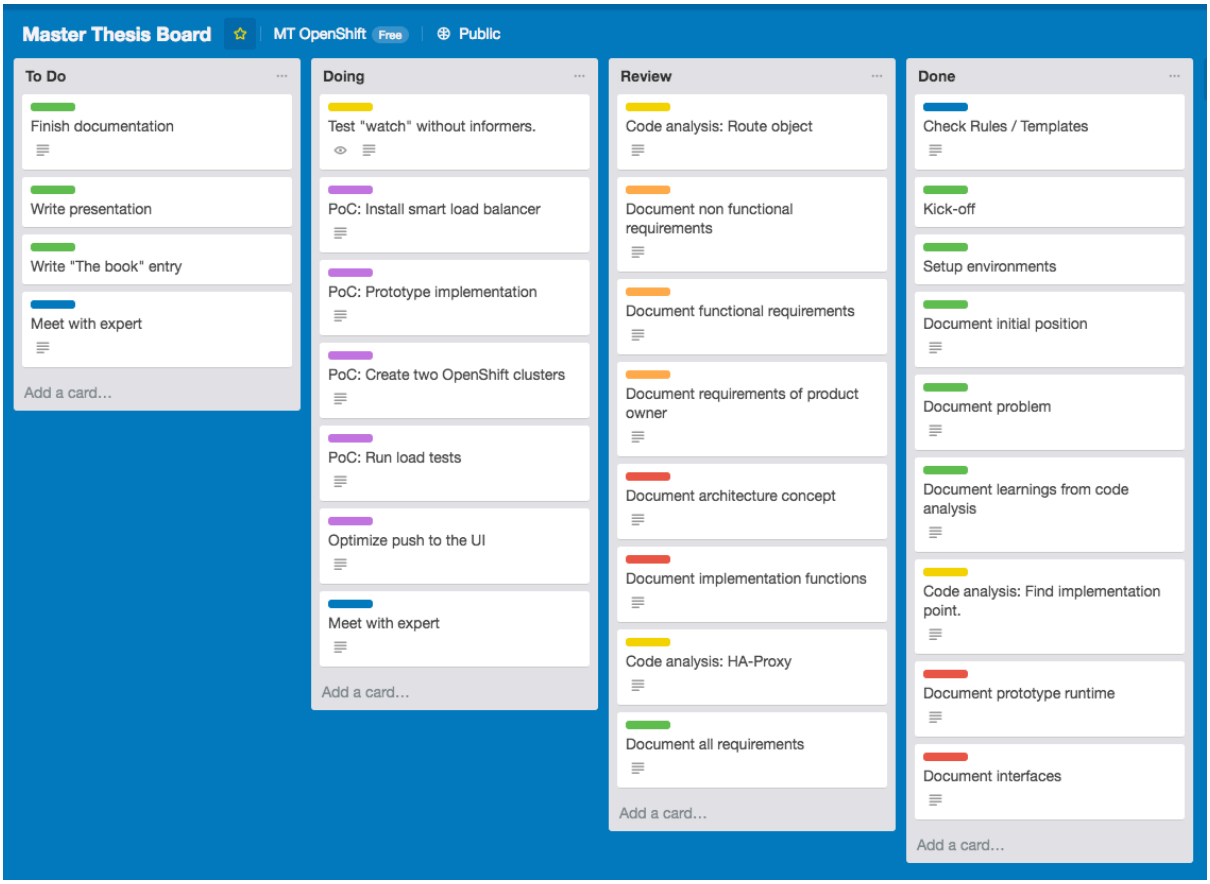


Figure 4: Thesis scrum task-board.

[13] (Scrum.org, 2018)

6.2 Thesis risk management

Before the start of the master thesis the following risks were identified. Those risks were tracked and monitored during the master thesis:

Risk description	Measure
Not enough time to do the full prototype implementation.	Research & evaluation of existing Open-Source implementation that can be reused.
Necessary information are not available.	Prior discussions and clarifications with important stakeholders.
Infrastructure for testing is not available.	Prior discussions and clarifications with important stakeholders.
Timing issues, versioning incompatibilities because OpenShift is constantly updated.	Define a fixed version set of OpenShift to work with. Build the prototype against that version.

Table 4: Risk and measures.

7 Solution

This chapter describes the idea of how to solve the problems described in chapter “5 Problem description”. It contains the solution idea, the architectural concept and the requirements.

7.1 The solution ideas origin

As mentioned in the introduction, an idea of how to maintain a large productive OpenShift cluster nearly risk free, was born upfront at Swiss Federal Railways. This chapter describes where the idea originated.

The idea is based on OpenShifts application deployment principle. OpenShift uses a principle called rolling update[14] to install a new version of an application. During the update procedure, the application is always available and usable for the clients. The following figure will illustrate the process:

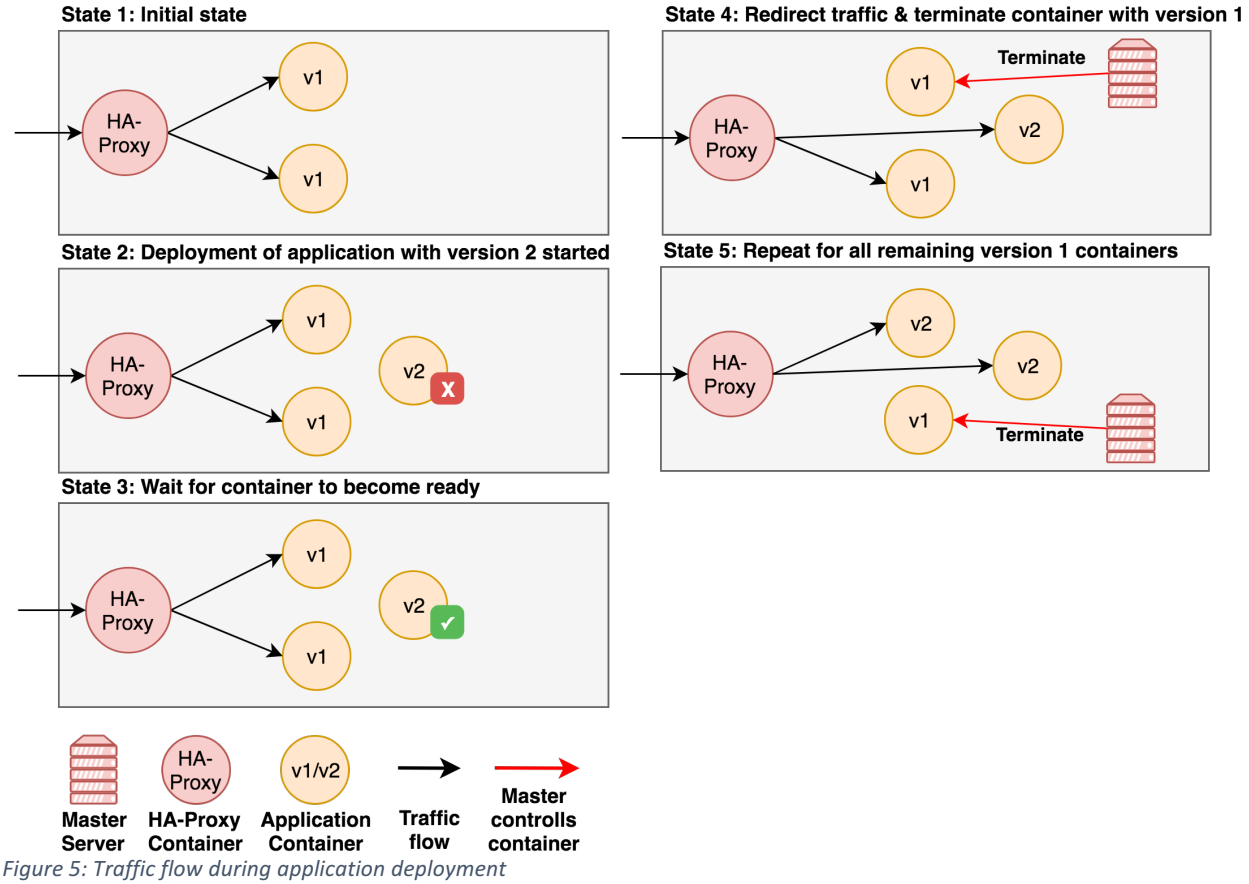


Figure 5: Traffic flow during application deployment

[14] (Kubernetes, 2018)

State	Description
State 1: Initial state	The application A with version v1 is scaled to two container.
State 2: Deployment of application A with version 2 started	A new container with application A with version v2 is started.
State 3: Wait for container to become ready	An application can expose an API to tell OpenShift when it is ready. OpenShift waits for the application to become ready.
State 4: Redirect traffic & terminate container with version 1	After the new container became ready, OpenShift will change the traffic flow to the new application container. After that it will send a termination signal (SIGTERM [15]) to one of the containers with application in version v1. That container can then finish its pending requests and do cleanup tasks.
State 5: Repeat for all remaining version 1 containers	OpenShift will repeat the same process for every container with the old application version.

Table 5: Steps during an application deployment.

With this deployment-procedure OpenShift enables application deployments without downtime. Out of this approach, the **idea was born to use the same procedure to update a OpenShift Cluster**.

[15] (GNU, 2017)

7.2 High level approach

As mentioned in the previous chapter, the idea of updating an OpenShift cluster using a rolling update approach arose from OpenShifts application deployment model. This chapter describes the approach on a high level.

The approach should enable the rollout of all components of a new OpenShift version **without introducing any risk or downtime** to the cluster and its applications. To achieve this goal, the rollout a new OpenShift cluster version should work the same way as a rolling update of applications (see chapter “7.1 The solution ideas origin”).

Today, a new version of OpenShift is installed directly on the existing cluster. This introduces the risks mentioned in chapter “5 Problem description”. With the rolling update approach, a new OpenShift version is installed on a **new and empty OpenShift cluster** and applications are slowly migrated and tested. This way, risks for unplanned downtime are mitigated.

The installation process for a new OpenShift version would look like this:

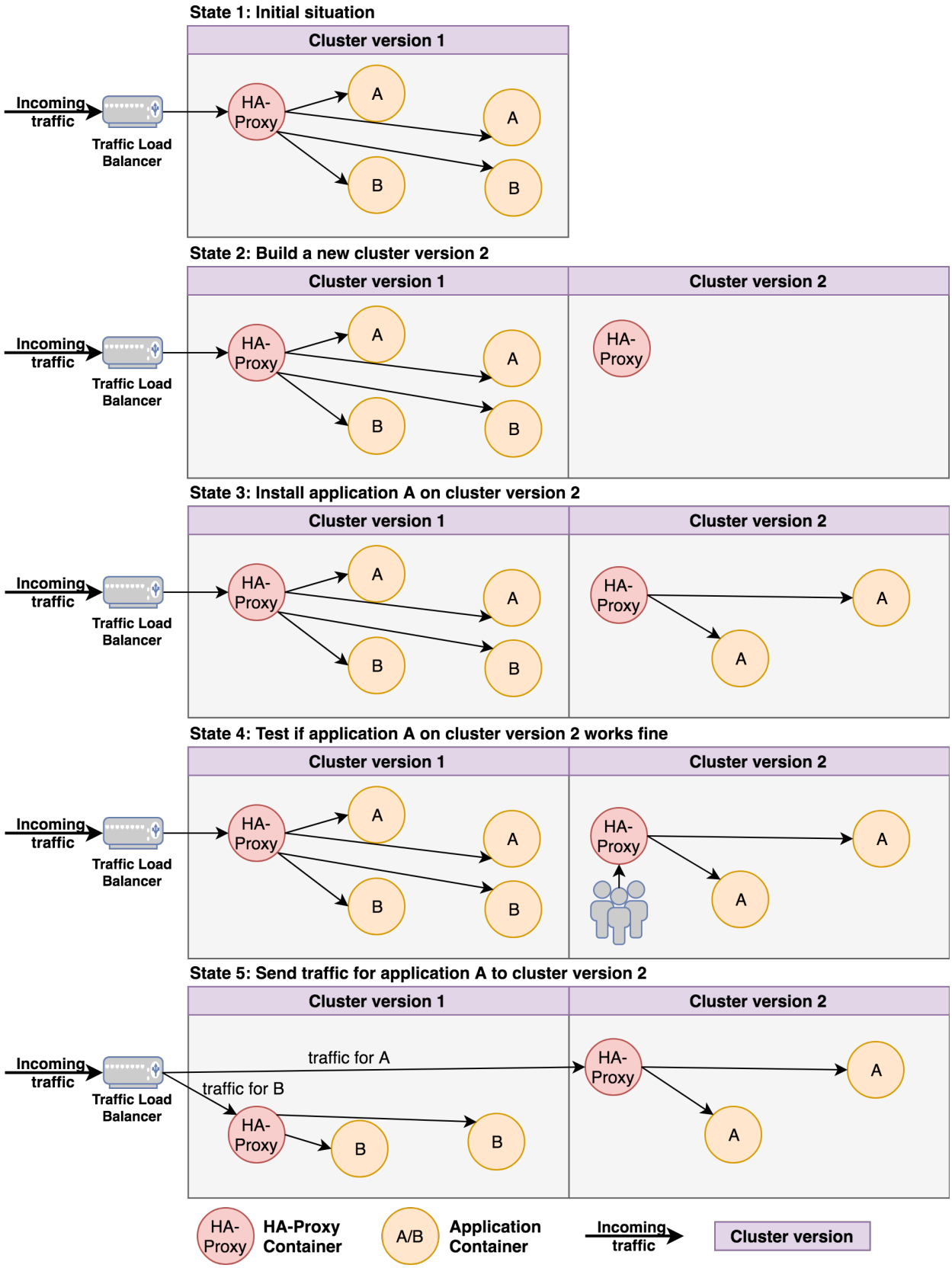


Figure 6: OpenShift rolling update of new cluster version.

State	Description
State 1: Initial satiation	- A OpenShift cluster in version v1 is set up. - Application A & B are deployed to that cluster. - The traffic load balancer sends all traffic to the OpenShift HA-Proxies on cluster v1.
State 2: Build a new cluster version 2	- A new OpenShift cluster with version v2 is set up. - This new cluster has no application installed on it, just the basic OpenShift components like the HA-Proxy are installed.
State 3: Install application A on cluster version 2	- The application A is deployed to the v2 cluster. - Users are still working on Application A on cluster v1.
State 4: Test if application A on cluster version 2 works fine	- IT-Operations and users are testing the application A on cluster v2.
State 5: Send traffic for application A to cluster version 2	- If all tests are successful, the traffic load balancer is configured to send traffic for application A to the v2 cluster. - Users are now working on application A on cluster v2.
Further steps	- The development team now can shut down the application A on cluster v1. - The same process is repeated for every application on cluster v1. - Finally, the cluster v1 can be shut down as well.

Table 6: Steps of a rolling update of a new cluster version.

7.3 The approach in depth

The above idea description shows an approach to update an OpenShift cluster without any risk and downtime. There are a few prerequisites necessary that this approach can work. Most of them are already fulfilled today:

A fast way to create a new OpenShift cluster

During one year of operations a lot of updates are installed on the OpenShift platform. A new major release, which replaces every component in a OpenShift cluster, is installed every three months. If those updates were applied with the new approach, the creation of a new OpenShift cluster cannot take longer than a few hours.

This prerequisite is already fulfilled today, as the creation of an OpenShift cluster is fully automated with Ansible [16]. The setup of a new OpenShift cluster takes only a few hours, including the setup of the servers and other necessary infrastructure.

A fast way to start applications on the empty cluster

The same prerequisite applies to installing applications. If 450 developers have to install their applications four times a year on a new OpenShift cluster, it has to be fully automated and run within a few hours.

This prerequisite is also already fulfilled today. The applications are packaged in the container format. The resulting container image can be started on any OpenShift cluster. Only the necessary OpenShift configuration needs to be installed on the new cluster. At Swiss Federal Railways the necessary configuration is placed in source control (git [17]) and the process of deploying everything to OpenShift is fully automated in a continuous delivery [18] pipeline. Based on that setup, it takes only one script to deploy an application to any OpenShift cluster.

A "smarter" load balancer

The last prerequisite is a traffic load balancer that is aware of both OpenShift clusters. An application can run on multiple OpenShift clusters during the migration phase. The traffic load balancer that is in front of both OpenShift clusters needs to be aware of which application is currently running on which cluster.

Today the traffic load balancers are statically configured. They do not know about applications that are running on an OpenShift cluster. They just forward any request to OpenShift. To have a "smarter" traffic load balancer **is the last missing peace** to enable this solution approach.

The goal for this master thesis is to **create a prototype of this smarter load balancer** and proof the solution idea.

[16] (Ansible, 2017)
[17] (Git, 2017)
[18] (Wikipedia, Continuous Delivery, 2017)

7.4 Requirements

Based on the solution idea (see chapter “7.3 The approach in depth”) a new smart load balancer is needed. This brings a few requirements from different sources. The smart load balancer has to fulfil technical and business requirements.

First, the smart load balancer has to provide the same functionality as the current setup (see chapter “7.5.1 Overview”) to avoid breaking existing functionalities. Then, it has to fulfil new requirements based on the problem solution idea (see chapter “7 Solution”), as the solution idea adds new functionality to the load balancer. The following table lists the requirements:

No.	Description
Requirements based on existing setup [19]:	
1	Handle traffic for all applications on OpenShift with a defined route object.
2	Balance HTTP traffic and provide a HTTP Listener for those requests.
3	Balance HTTPS traffic and provide a HTTPS Listener for those requests.
4	Check the health of HA-Proxies every 5 seconds using a HTTP call.
5	Balance traffic to the HA-Proxy that has the least amount of current connections.
6	The external load balancer overhead has to be less than 10%.
Requirements based on the solution idea:	
7	The smart load balancer must be able to handle multiple OpenShift clusters and their applications.
8	A route can be active on multiple OpenShift clusters. A weight (1-100) can be set to control how many requests are sent to which cluster.
9	The smart load balancer can be reconfigured online (while running without an interruption to traffic balancing).
10	The smart load balancer should provide a user interface to see its current state.

Table 7: Requirements for the smart load balancer.

[19] Requirements from Swiss Federal Railways.

7.5 Solution architecture

Based on the solution idea of the previous chapter, this chapter describes the concrete solution architecture. It visualizes the current setup, the setup with the new smart load balancer and describes two communication flows.

7.5.1 Overview

To give an overview of the differences of the current and the new approach the following figures (Figure 7 and Figure 8) illustrate the setups. Figure 7 illustrates the current setup with the static external load balancers:

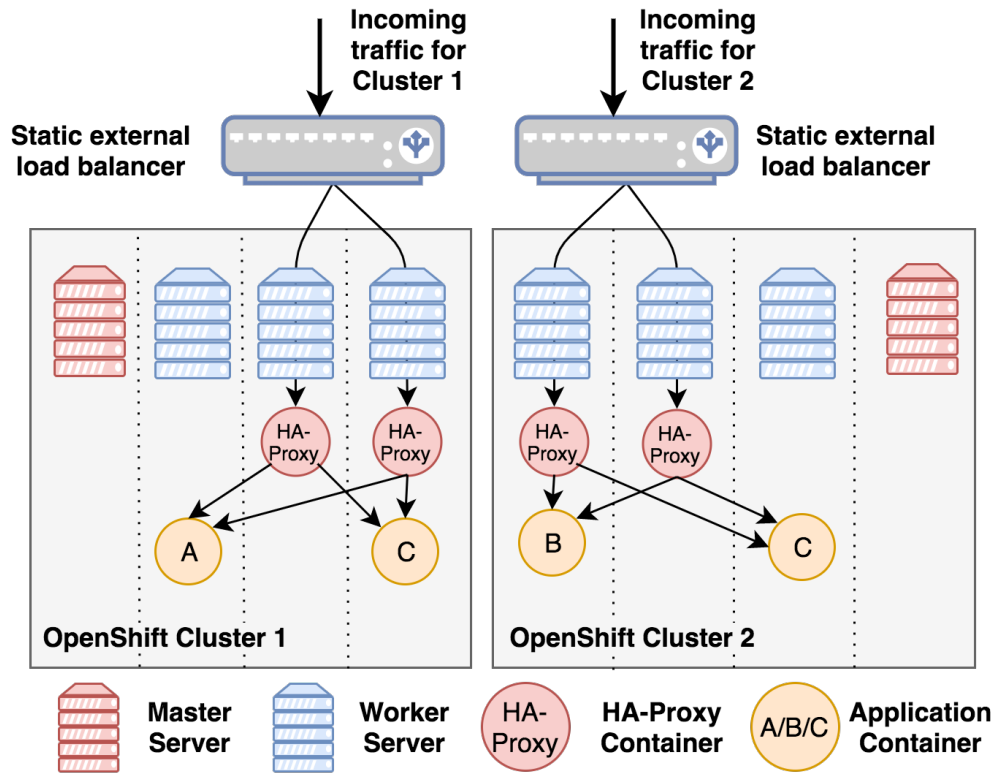


Figure 7: Solution overview with static external load balancer.

Each OpenShift cluster has its own static external load balancer for application traffic. Every load balancer is configured to balance traffic to all the worker servers of the OpenShift cluster. The load balancer uses a health check to see if a HA-Proxy is running on the worker server and if the HA-Proxy is healthy. If a HA-Proxy is running and healthy, the application traffic is balanced to the HA-Proxy which then sends the requests to the corresponding application container.

The application container and the HA-Proxy could run on the same or on a different worker server. The HA-Proxy uses the software defined network to route the request to the worker server where the application container is running.

In the new architecture, that is designed in this thesis, the smart load balancer will replace the static external traffic load balancer. The new setup the looks like this:

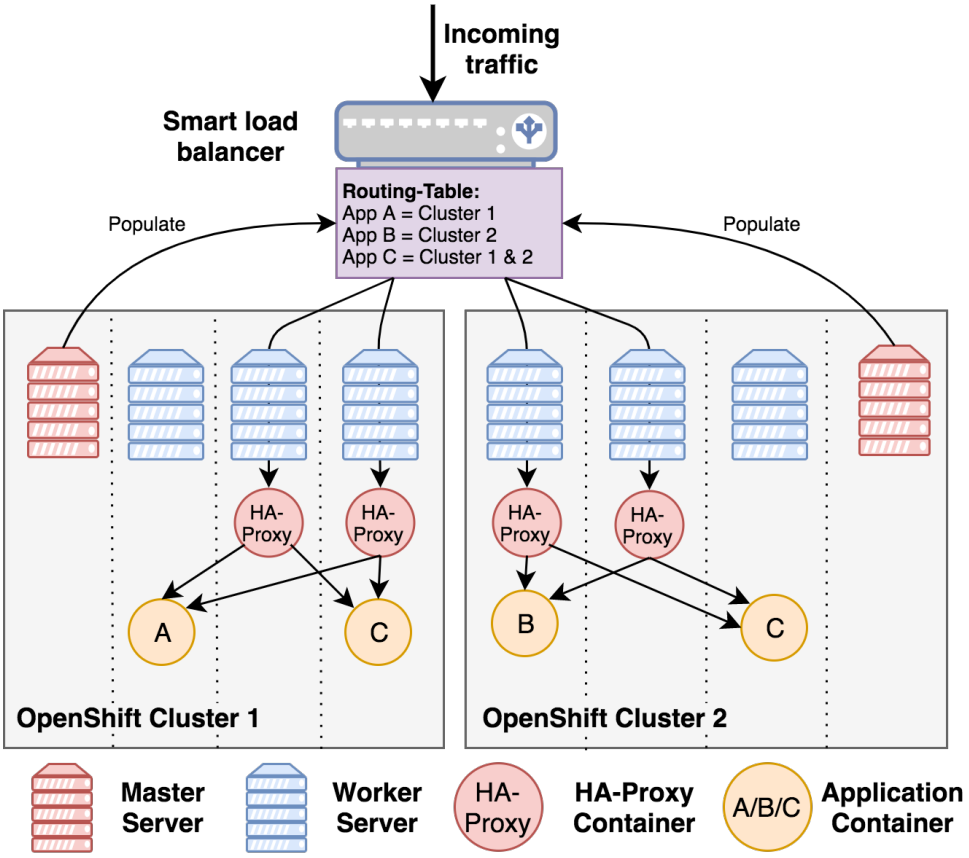


Figure 8: Solution overview with smart load balancer.

The smart load balancer combines the two static load balancers of the setup in Figure 7. Each master server is aware of every application that is running on his OpenShift cluster. This information is continuously populated to the smart load balancer. He maintains a routing table out of this information.

If a new request comes in, the smart load balancer checks its routing table and balances to the corresponding OpenShift cluster. Like the static load balancer, the smart load balancer also uses health checks to examine which worker node has a running and healthy HA-Proxy.

7.5.2 Information in OpenShift

As seen in the previous chapter, the smart load balancer needs information about the OpenShift clusters. That information is defined in a set of objects in the OpenShift cluster. For the smart load balancer, the following objects are relevant:

Name	Description
Pod	A pod [20] specification is a reference to a group of running containers. A pod specification contains information about: - The running container - The worker node that it is running on - Application configuration - Startup parameter - Resources that are requested & used
Route	A route [21] describes the hostname or path to one or more deployed applications. An application is reachable from outside the OpenShift cluster when a route is exposed on the HA-Proxy.

Table 8: OpenShift and kubernetes object types

The external traffic load balancers were configured statically. The smart load balancer has to get its configuration dynamically from the master servers. It needs to know where the HA-Proxies are running. To achieve this, the pod object is used to determine on which working servers a HA-Proxy is deployed. As the HA-Proxy itself is also a container, a pod object is available for every HA-Proxy instance. The pod object contains the IP address [22] of the working server.

The route objects in an OpenShift cluster describes which application is reachable from the outside via the HA-Proxy. Thus the full list of all routes needs to be sent to the smart load balancer to create the routing table.

The pod and route objects are stored in OpenShifts configuration database (ETCD). The master servers are responsible to manage those objects in ETCD. The master servers expose an API for developers to interact with those objects. If a developer wants to use those data, a plugin for OpenShift needs to be created. For the smart load balancer a new OpenShift router plugin [23] needs to be developed. The new plugin will then extract the necessary data from the OpenShift cluster. The setup will look like this:

[20] (Kubernetes, Pod Spec, 2017)
[21] (Redhat, Route Spec, 2017)
[22] (Gartner, IT-Glossary: IP address, 2017)
[23] (Redhat, Router-Plugin, 2017)

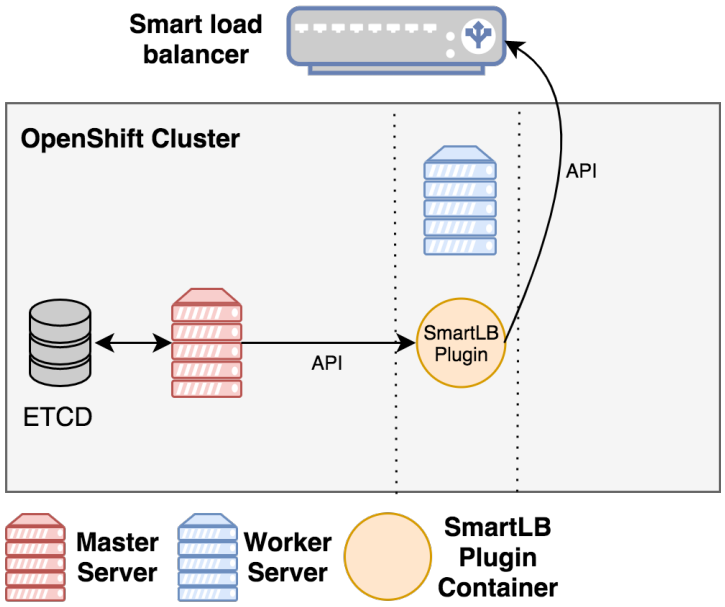


Figure 9: Smart load balancer plugin.

The smart load balancer plugin runs as container on a worker server in the OpenShift platform. It communicates with the master API to watch the **pod** and **route** objects. The master server communicates with the ETCD data store.

On startup, the plugin will get the list of pods and routes of the OpenShift cluster from the master API and forward this information to the smart load balancer. For this purpose, the smart load balancer itself also exposes an API to receive those configuration data and maintains its cluster & routing table based on it.

The plugin also has to get all modifications on the pod and route objects. Developers and the operations team change the pod and route objects all the time. Those modifications must continuously be propagated to the smart load balancer. For that purpose the OpenShift API has a “watch for changes” mode [24]. With this API mode the plugin is continuously informed about every change to an object (created, deleted, modified). The plugin listens for changes on all route objects in the cluster and on all HA-Proxy pod objects. That information is then continuously propagated to the smart load balancer.

[24] (Redhat, API Changes, 2017)

7.5.3 The smart load balancer

Before starting the implementation of the smart load balancer, an architecture concept was designed. The following figure shows the internal architecture of the smart load balancer [25]:

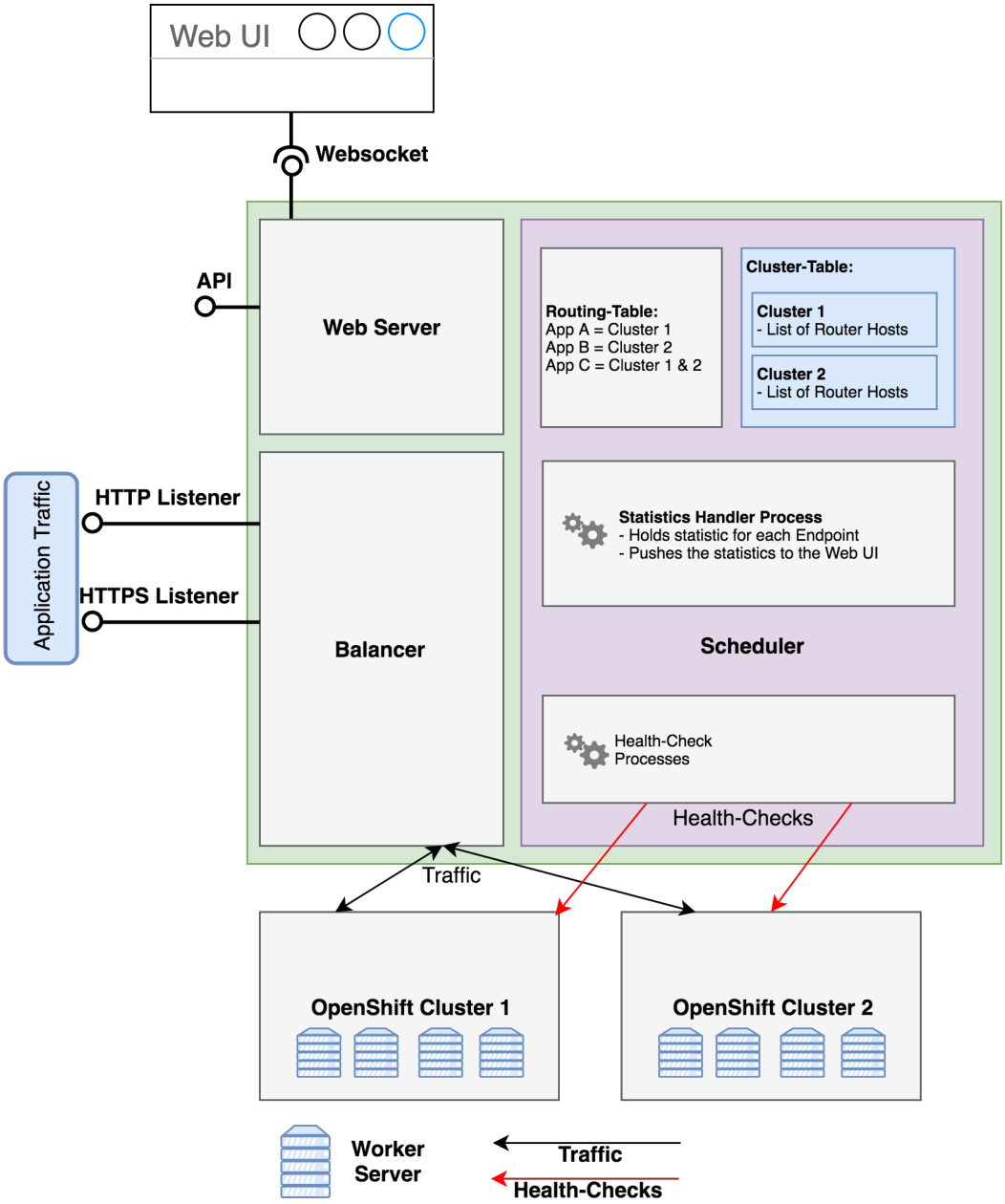


Figure 10: Architecture of the smart load balancer.

[25] Inspired by existing implementations. See "8.1 Preface"

The smart load balancer consists of the following core components:

Name	Description
Web UI	The smart load balancer exposes a Web User Interface that displays status and metrics data of the smart load balancer.
Websocket	The Web UI uses websocket [26] technologies to get the status and metrics data in real time from the smart load balancer backend.
Web Server	The Web Servers exposes an API for the OpenShift Plugin and the websocket endpoint for the Web UI.
Balancer	The Balancer exposes two listeners, one for HTTP and one for HTTPS [27] traffic. Clients connect to those endpoints and are balanced to the OpenShift clusters.
Scheduler	The scheduler is responsible for managing the state of the smart load balancer. The scheduler is also responsible to run health checks against every working server to make sure that the HA-Proxy is running on the server and contains the statistics handler (see description below).
Cluster & Routing table	The routing table holds the OpenShift route objects. The cluster table holds a list of all router hosts (worker servers with HA-Proxy on them).
Statistics Handler Process	The statistics handler keeps track of statistic information for each router host: - Total connections - Currently active connections - Total refused connections - Health state over time These values are pushed to the web UI over websocket. The web UI displays those values as a chart diagrams.

Table 9: Smart load balancer core components.

[26] (Mozilla, 2017)
[27] (Gartner, IT-Glossary: HTTP, 2017)

7.5.4 Communication in detail

Also part of the architecture design are communication flows. They help to understand how multiple components talk to each other. This chapter describes two communication flows. The first flow describes the communication between OpenShift and the smart load balancer. The second flow describes the communication from a client to an application. Each subchapter gives detailed insight in the involved components and their information flow.

7.5.4.1 OpenShift to smart load balancer

The following UML sequence diagram shows how the smart load balancer gets its information from the smart load balancer plugin. It also shows how a change through a developer is propagated back to the smart load balancer:

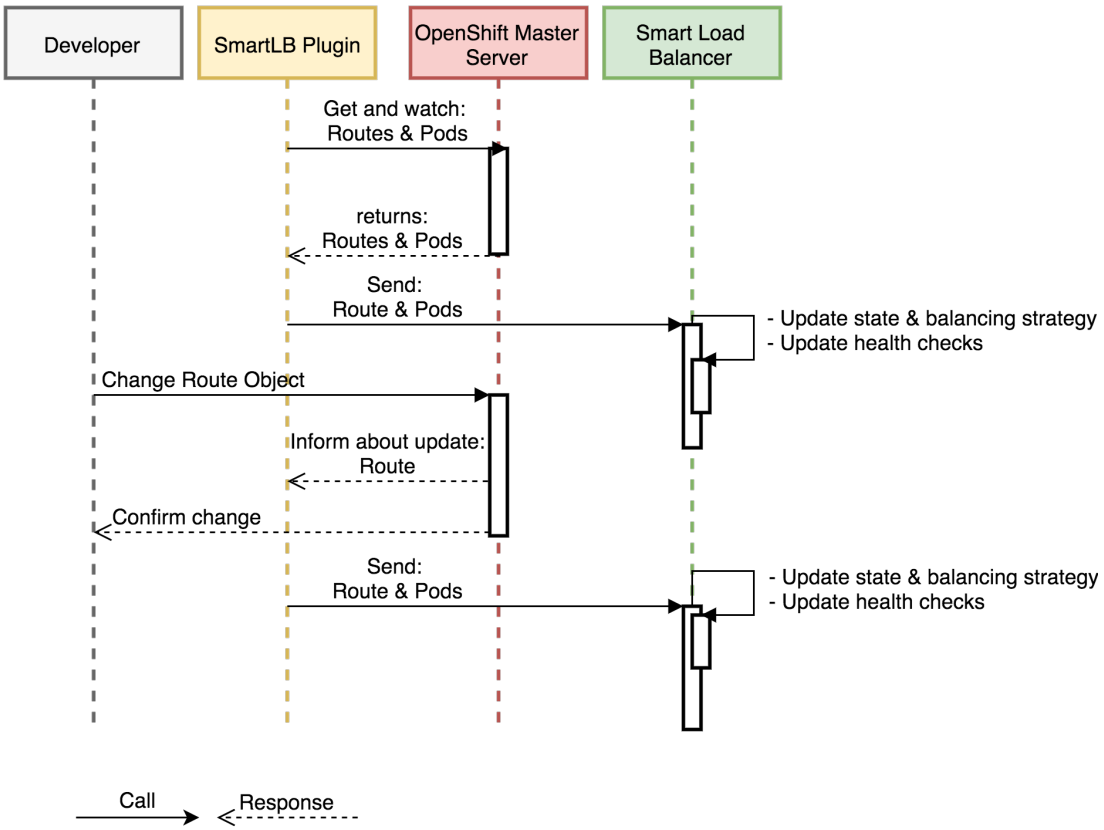


Figure 11: UML sequence diagram: Communication from OpenShift to smart load balancer.

The smart load balancer plugin takes care of watching the resources on the OpenShift API. It initially connects to the API on the OpenShift Master Server and sends the initial objects to the smart load balancer. He updates his own state (routing table, cluster table) with the new data and updates its balancing strategy and health checks. If a developer changes a route object on the OpenShift master, the smart load balancer plugin is informed about the change and propagates it also to the smart load balancer.

7.5.4.2 Client Request via smart load balancer to application

The following UML sequence diagram shows the request flow form a client (typically a user with a computer) to an application on OpenShift:

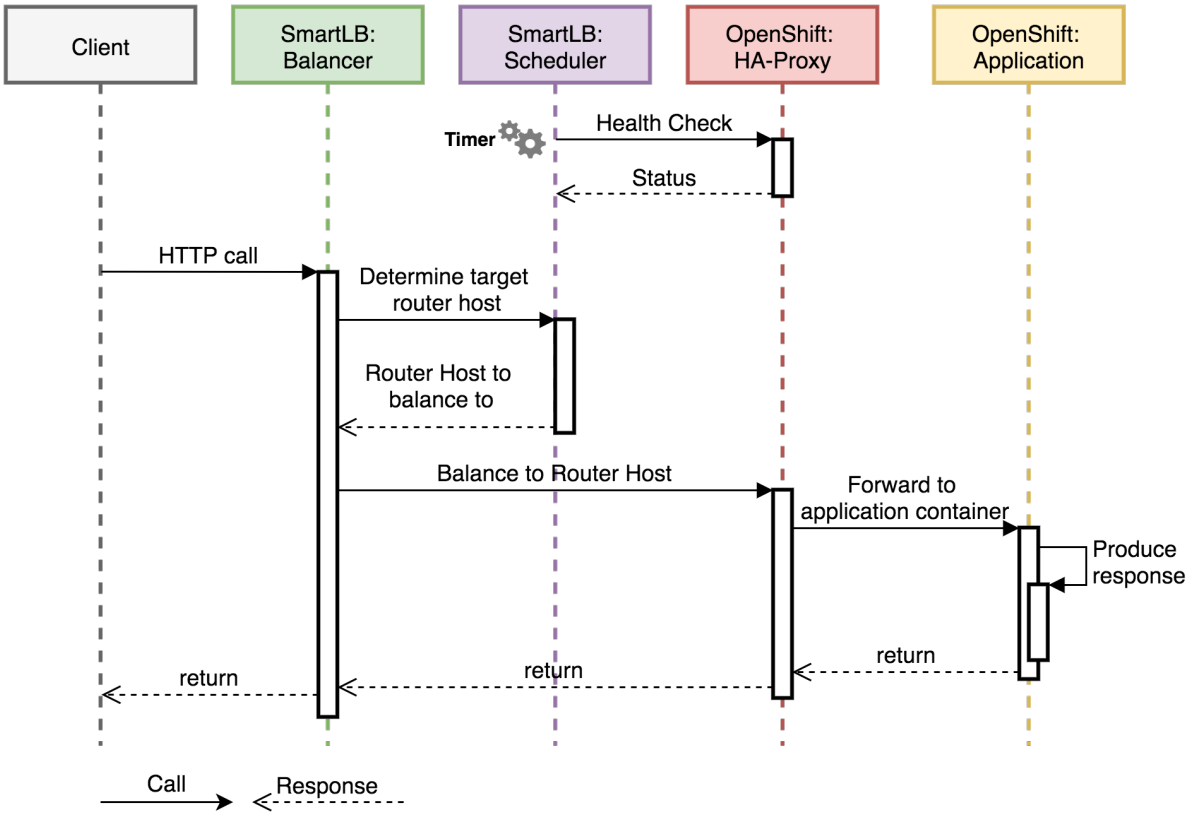


Figure 12: UML sequence diagram: Communication from client request to application.

The smart load balancer scheduler checks the health of every HA-Proxy server (called router host) based on a timer event. The HA-Proxy responds with its status.

The other invocation point is, when a client makes a HTTP call to the smart load balancer. The load balancer then determines which router host is a possible target. To determine which host is picked, the smart load balancer uses its routing table and the health status of each HA-Proxy. For more details on this mechanism see “8.4 Balancing strategy”. If the router host is picked, the balancer forwards the request to the specified HA-Proxy server. The HA-Proxy then forwards the request to the corresponding application container. The application is then going to generate a HTTP response, which is sent back to the client.

8 Prototype: Smart load balancer

As described in the introduction, the solution idea of this master thesis should be tested using a prototype. To test the architecture concept and the solution idea, a prototype of a smart load balancer was created. This chapter describes the prototype and the actual implementation. It corresponds to the architecture concept, but adds some details of the actual implementation and mechanisms.

8.1 Preface

To avoid recreating a load balancer from scratch, existing open-source load balancers that were implemented with golang [28] were researched. Golang was chosen as a programming language because OpenShift is implemented in golang as well. The following three implementations were found and assessed:

Name	Assessment summary
gobetween [29]	A feature rich load balancer implementation. It brings: - TCP [30] load balancing with HTTPS handling - Health checks - Different balancing strategies - Management API It is actively maintained and in productive use.
kahlys/tcpproxy [31]	A simple load balancer implementation. It brings: - TCP load balancing - Handling of timeouts
bradfitz/tcpproxy [32]	A simple load balancer implementation. It brings: - TCP load balancing - HTTP Header detection

Table 10: Research results of open source load balancers.

For the smart load balancer, as many parts or modules from those existing implementations were reused. Especially gobetween fulfilled a lot of the given requirements and was thus a good base to start. A few tests with static balancing showed, that gobetween has some modules that are quite resource intensive and are not necessary for the smart load balancer. They were not reused.

[28] (Golang, 2017)
[29] https://github.com/yyyar/gobetween
[30] (Gartner, IT-Glossary: TCP, 2017)
[31] https://github.com/kahlys/tcpproxy
[32] https://github.com/bradfitz/tcpproxy

The other two implementations each brought a unique feature that is necessary for the smart load balancer. Kahlys tcpproxy provides a good handling of timeouts and bradfitz tcpproxy detects HTTP headers. Those features were added to the smart load balancer.

8.2 The smart load balancer in detail

The high level architecture of the smart load balancer looks as described in chapter "7.5.3 The smart load balancer". To give more insights about the effective internal structure of the smart load balancer, the following UML class diagram was created:

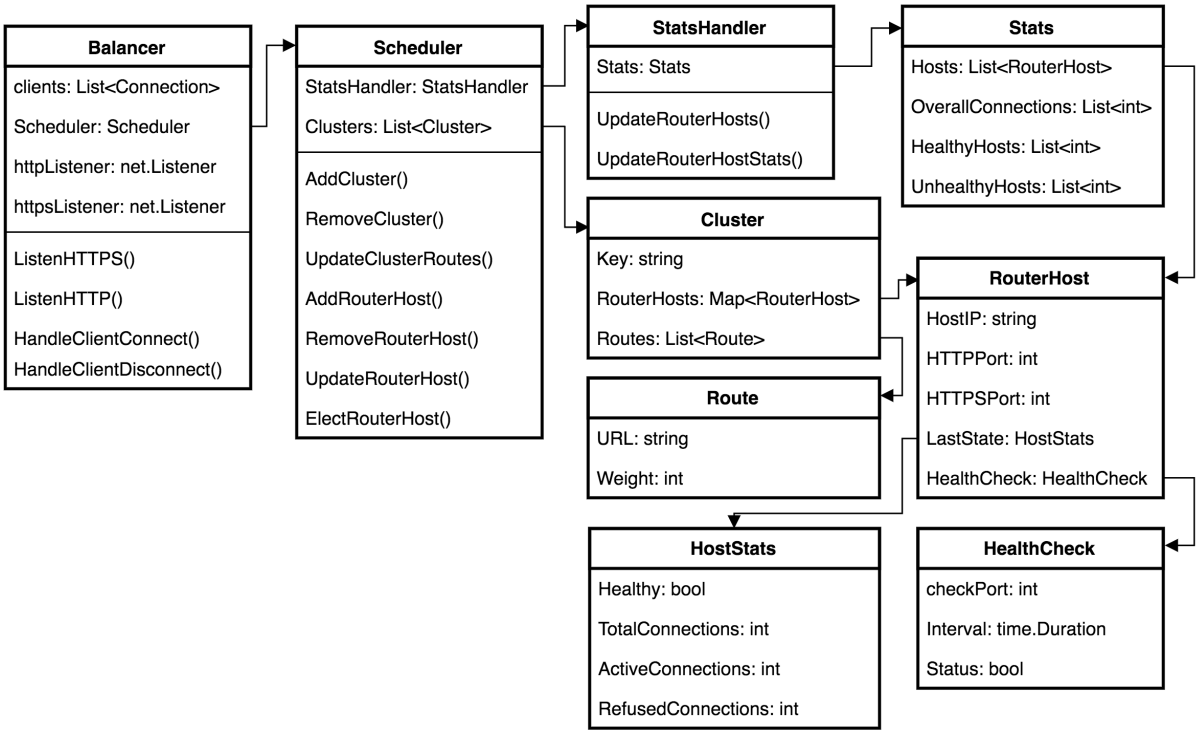


Figure 13: UML class diagram: Smart load balancer. [33]

[33] This figure is simplified.

A requests to the smart load balancer is processes as described in the concept in chapter “7.5.4.2 Client Request via smart load balancer to application”. During the implementation of the prototype **a few steps had to be added**. The following UML sequence diagram shows the extended flow:

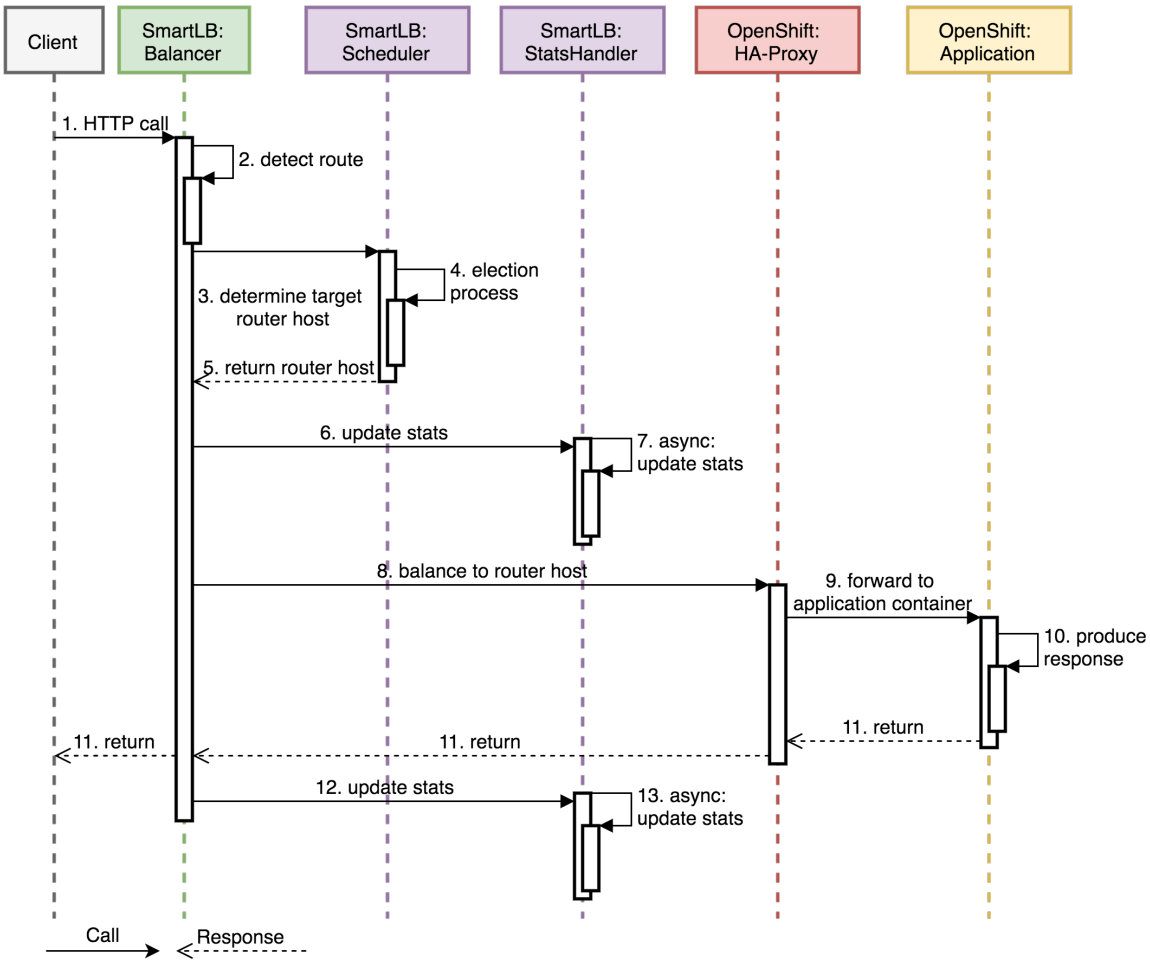


Figure 14: UML sequence diagram: Communication from client trough smart load balancer to application in detail.

No.	Description
1.	A client make a HTTP call to the smart load balancer.
2.	The balancer has to detect to which application the client wants to connect. See “8.3 Route detection” for more details.
3.	The balancer sends an asynchronous request to the scheduler to elect a router host for the detected route.
4.	The scheduler runs an algorithm to elect a router host as balancing target. See “8.4 Balancing strategy” for more details.
5.	The elected router host is returned to the balancer.
6.	The balancer calls an asynchronous update procedure on the StatsHandler to increase the router hosts statistics values (total, active and refused connections).
7.	The StatsHandler updates its internal statistics state.

8.	The balancer opens a connection to the target router host and proxies the request and response streams.
9.	The HA-Proxy forwards the request to the application container.
10.	The application generates a HTTP response.
11.	The response is sent back to the client.
12.	The balancer calls an asynchronous update procedure on the StatsHandler to decrease the active connections on the target router host.
13.	The StatsHandler updates its internal statistics state.

Table 11: Description for figure "Communication from client trough smart load balancer to application in detail".

8.3 Route detection

As described in the previous chapter, the smart load balancer has to figure out which application is called by the client. Depending on the protocol (HTTP or HTTPS) the scheduler has to figure it out differently. The following two subchapters describe how the scheduler detects the called application using the applications route.

8.3.1 HTTP Traffic: Host header

In a HTTP request, the client specifies in a HTTP-Header [34] which application he wants to reach. The header is called 'Host' and looks like this:

```
# Example http request with 'Host' header
$ curl -v www.google.ch
GET / HTTP/1.1
Host: www.google.ch
```

The smart load balancer uses this strategy to detect which application a client requested. The request is enriched with this information and then forwarded to the scheduler to elect a valid router host for this host header.

[34] (Mozilla, MDN: Http Header, 2017)

8.3.2 HTTPS Traffic: SNI

Encrypted traffic introduces an obstacle when detecting the route in the smart load balancer. This chapter explains why and how this issue was solved.

A secure route in OpenShift is reached with TLS [35] encryption and a HTTPS endpoint. There are a few different modes where the encryption ends (also called: where TLS is terminated). For the smart load balancer, they are all the same:

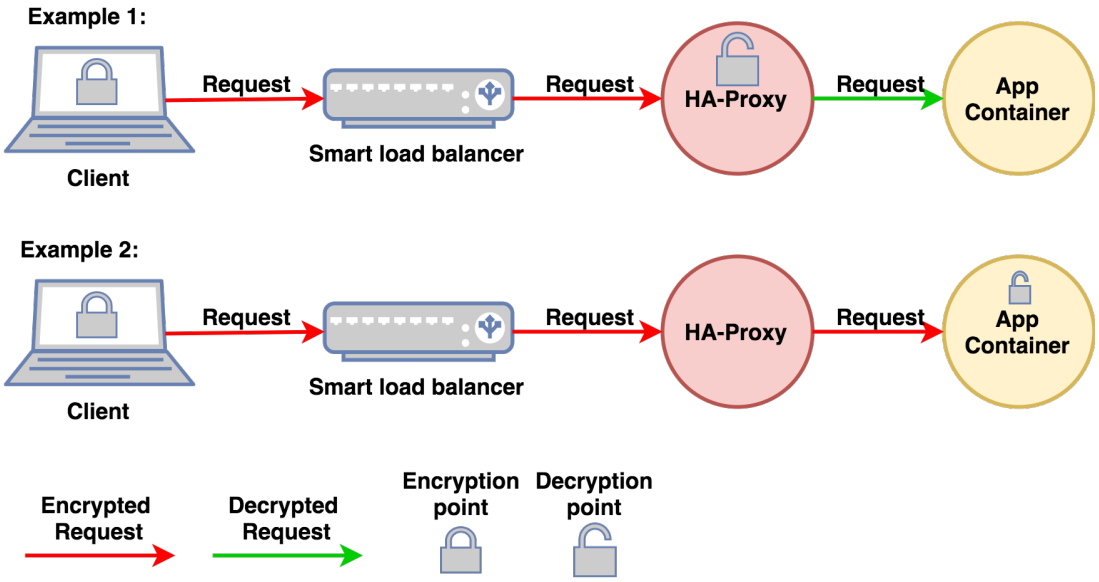


Figure 15: HTTPS routes and the smart load balancer.

The figure above shows two variants of TLS encrypted HTTPS requests. The first is terminated at the HA-Proxy, then sent to the application unencrypted. The second example shows traffic that stays encrypted until the request reaches the application.

For the smart load balancer all the secured traffic **is not readable**. Thus, he is unable to read the "Host" header from the request.

A possible solution to detect which application the client wants to reach is server name indication (SNI) [36]. SNI adds the "Host" information in a readable part of the otherwise encrypted request. This way the smart load balancer is able to detect which application the client wants to reach, without decrypting the HTTPS request. The request is then enriched with this information and forwarded to the scheduler to elect a valid router host.

[35] (Mozilla, MDN: TLS, 2017)
[36] (TheSSLStore, 2017)

8.4 Balancing strategy

The smart load balancer uses a custom balancing strategy. The following figure describes this balancing strategy with an example request. A client requests an answer of an application with the host “app1.ch”. The application is installed on two OpenShift clusters. The smart load balancer uses its routing- and cluster-table to elect a router host:

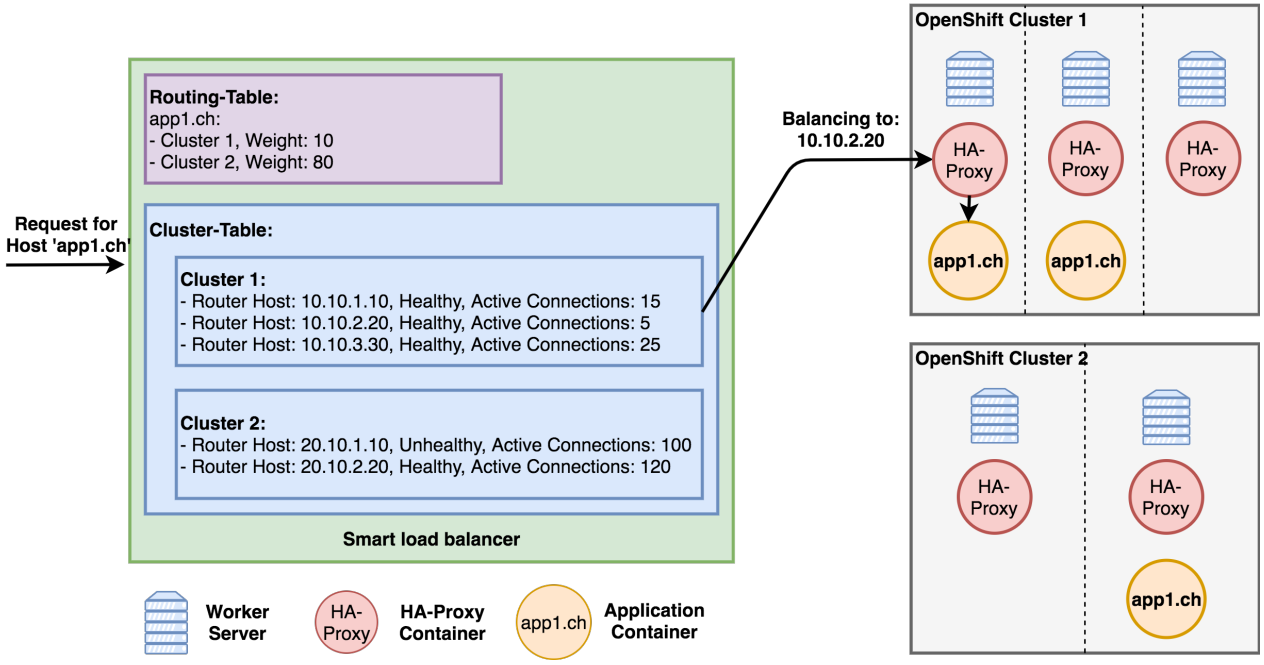


Figure 16: Example of balancing strategy with a sample request.

First the smart load balancer detects the route according to “8.3 Route detection”. In this example the client sends a request with a Host-Header of “app1.ch”. Then the smart load balancer elects to which router host the request should be sent. This works in two steps:

1. The smart load balancer checks its routing table for the detected route. In this case, the entry points to both OpenShift cluster 1 and 2. Both clusters are weighted. Cluster 1 has a weight of 10, cluster 2 has a weight of 80. This means, that out of nine requests one is sent to cluster 1 and eight are sent to cluster 2. In this example, cluster 1 is selected to balance to.
2. In the next step, the smart load balancer checks which router hosts are available on the selected cluster. Only router hosts that are healthy are considered. Each router host has a statistic field called “active connections”. The field shows how many connections are currently open to the specific router host. The smart load balancer sends the request to the router host that has the least active connections.

8.5 Challenge: High concurrency

8.5.1 Two problems

During the implementation and early testing of the smart load balancer prototype **the most challenging obstacle was high concurrency**. On a production OpenShift environment a lot of parallel requests have to be processed. At the beginning, all tasks in the smart load balancer were ran sequentially (after each other). This introduced the following problems:

Problems	Description
Statistics information	The smart load balancer track statistics about: - Active connections - Refused connections - Total connections Those values are changed with every client that connects or disconnects. Also those statistics are saved for every router host. To update the statistics information for every request imposes a bottleneck to the smart load balancer. If the statistics update is to slow, requests would have to wait for the completion.
Constant state change from different sources	The smart load balancer state (routing & cluster table) are constantly updated from multiple sources: - From OpenShift via the smart load balancer plugin. Developers are modifying routes. Operators are modifying HA-Proxies. - From health check results. A HA-Proxy could become healthy or unhealthy. Those changes influence the balancing strategy for the smart load balancer. During the state update clients concurrently connect and disconnect. This also imposes a bottleneck and possible inconsistent state on the smart load balancer.

8.5.2 The solution

To solve both problems of the previous chapter, the following techniques were applied:

Parallel processing with goroutines:

Golang provides a model to execute processes in parallel called goroutines [37]. On the smart load balancer, the following processes run in parallel:

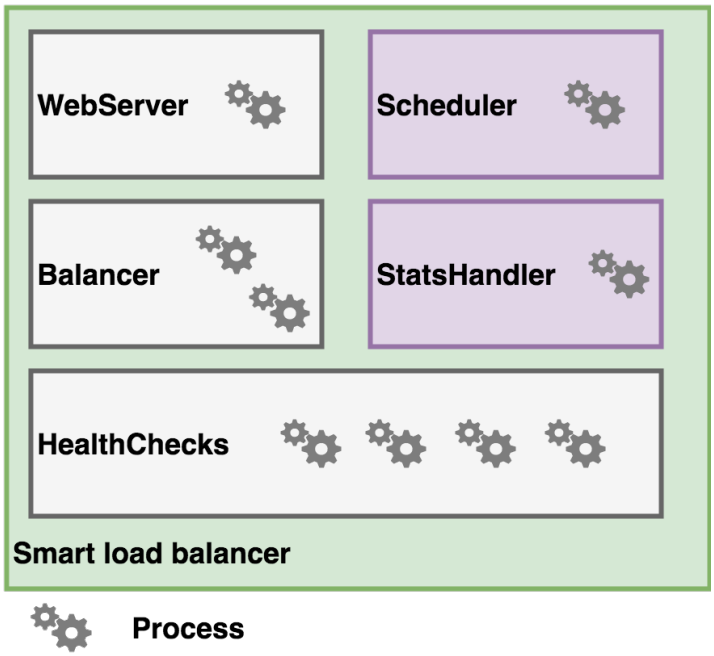


Figure 17: Processes in smart load balancer.

In the smart load balancer every component has its own process. The balancer has two processes, one for the HTTP-Listener and one for HTTPS-Listener. Every router host has its own process that checks for the health of his HA-Proxy.

[37] (GoByExample, Goroutines, 2018)

Communication with channels:
As those processes are all running in parallel, they need a way to communicate with each other. Golang provides a model called channel [38]. The processes communicate with each other using those channels:

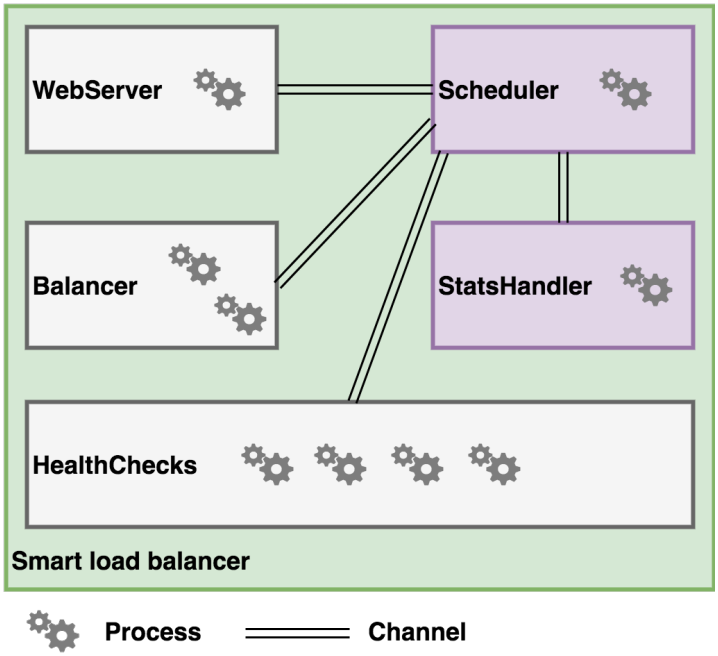


Figure 18: Channels in the smart load balancer.

Locking state where necessary:
When multiple processes access or modify on the same data in parallel race conditions [39] can occur. To avoid this, the data that will be changed has to be locked before modification. The smart load balancer uses locking on its router and cluster table to avoid inconsistent data.

With those three techniques it was possible to process the requests on the smart load balancer in high concurrency. Each process is decoupled and runs in parallel. The processes communicate via channels with each other. Some of those channels even work asynchronously, like the statistics update. A request can be processed and the statistics are updated later when the StatsHandler has time to do it. Furthermore, locking is used where data consistency is necessary.

[38] (GoByExample, Channels, 2018)
[39] (SearchStorage, 2018)

8.6 Web User Interface

To see what is happening on the smart load balancer a Web User Interface was created. The user interface displays information about the smart load balancer and the router hosts. It is attached to the smart load balancer and updates its information in real time. The user interface looks like this:

Figure 19: Screenshot of smart load balancer UI.

The user interface of the smart load balancer gives information about its overall state and about each configured router host. The screenshot shows an example with two router hosts. Every diagram shows the state over time (e.g. each second for the last 120 seconds).

No.	Label	Description
1	Overall connections	Displays all client connections over time to the smart load balancer.
2	Healthy hosts / Unhealthy hosts	Displays the amount of router hosts over time and their health status.
3	-	A row with three diagrams for each router host is displayed.
3a	Total Conn: router host name	Total connections of the router host.
3b	Active Conn: router host name	Currently active connections of the router host.
3c	Refused Conn: router host name	Refused connections of the router host.

Table 12: Smart load balancer UI description.

9 Findings

The goal of this master thesis was to test and proof the solution idea using a prototype. The implemented prototype was then installed on a testing environment of Swiss Federal Railways. This chapter describes the tests that were run against this setup.

9.1 General tests

First the general functionality was tested to make sure the smart load balancer fulfils the given requirements (see chapter “7.4 Requirements”). To test the requirement fulfilment, the following setup was created:

- Two OpenShift clusters with 10 application & routes each.
- Installed smart load balancer plugin on both clusters.
- Installed smart load balancer.
- Open smart load balancer web interface in a browser.

Requirement	Test case	Result
Handle traffic for all applications on OpenShift with a defined route object.	Call every route and check response.	Ok
Balance HTTP traffic and provide a HTTP Listener for those requests.	Call a HTTP route and check response.	Ok
Balance HTTPS traffic and provide a HTTPS Listener for those requests.	Call a HTTPS route and check response.	Ok
Check the health of HA-Proxies every 5 seconds using a HTTP call.	Stop a running HA-proxy. It should become unhealthy on the user interface within 5 seconds.	Ok
Balance traffic to the HA-Proxy that has the least amount of current connections.	Make 10 concurrent calls, check which HA-Proxy has the least current connections. Make another call and verify that the HA-Proxy with the least current connections was selected.	Ok
The external load balancer overhead has to be less than 10%.	See “9.2 Load tests”.	Ok
The smart load balancer must be able to handle multiple OpenShift clusters and their applications.	Tested trough other test cases.	Ok

A route can be active on multiple OpenShift clusters. A weight (1-100) can be set to control how many requests are sent to which cluster.	Add weights to both OpenShift clusters and check if balancing is correct with 1’000 test requests.	Ok
The smart load balancer can be reconfigured online (while running without an interruption to balancing).	Change a route while calling the route at the same time.	Ok
The smart load balancer should provide a user interface to see its current state.	Tested trough other test cases.	Ok

Table 13: General tests and results.

All the general function tests were successful. The smart load balancer **does fulfil** the given requirements.

9.2 Load tests

After the general function tests, the smart load balancer was tested during heavy load. One of the goals of this test was to see how much overhead in percent the smart load balancer adds to a request. The same tests were run against the smart load balancer and the current static load balancers to have a meaningful comparison. The following subchapters describe the setup and the test results.

9.2.1 Setup

To run the load tests the testing environment of Swiss Federal Railways was used. Two OpenShift clusters with two worker nodes and one master server were set up. In front of every cluster a static load balancer was set up. Also the smart load balancer was installed in front of both OpenShift clusters. The load test servers then sent traffic through those load balancers.

The following figure illustrates the setup:

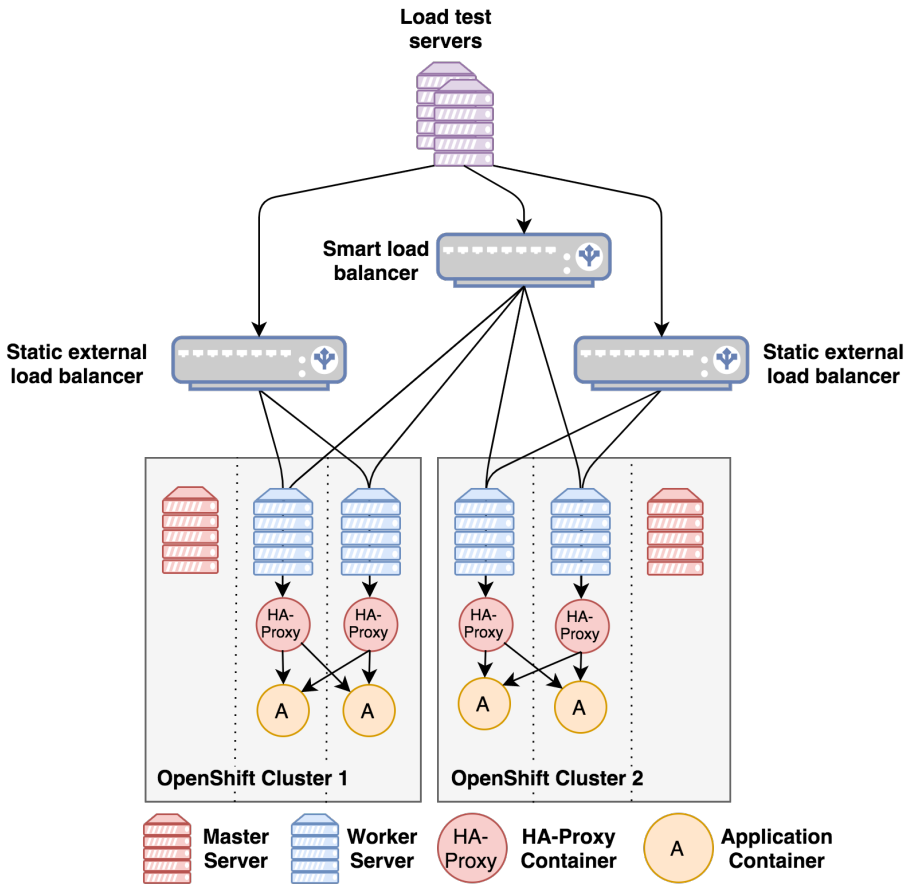


Figure 20: Load test setup.

To simulate load that is comparable with real application traffic, the load tests are divided in two test modes:

Unique connections

In the first mode the load test servers simulate clients that are unique. **Each client creates only one request and then closes the connection**. Each requests opens a new connection. This test simulates unique clients like website or mobile application users.

Reuse connections

In the second mode the clients create multiple requests to the application while using the same connection. **The connection is opened, all the requests are sent sequentially, then the connection is closed**. This scenario simulates server clients like other applications. If a server consumes an application, it would pool its connections and reuse the connection for multiple requests.

9.2.2 Results

This chapter contains the results of the load tests that were run against the smart load balancer.

Unique connections

The following table displays the results of the load test using unique connections for each request. In this tests mode, each client created a new connection for every http request.

Test scenario	Load balancer	Result
Mode: Unique connections **Requests**: 1'000 per client **Concurrency**: 1 request	Static load balancer	Total time: 2.09s **Requests/s: 477** Latency: 2.1ms
	Smart load balancer	Total time: 2.19s **Requests/s: 466** Latency: 2.2ms
Mode: Unique connections **Requests:** 10'000 per client **Concurrency:** 100 requests	Static load balancer	Total time: 3.71s **Requests/s: 2'694** Latency: 36.8ms
	Smart load balancer	Total time: 3.71s **Requests/s: 2'689** Latency: 36.8ms
Mode: Unique connections **Requests:** 100'000 per client **Concurrency:** 150	Static load balancer	Total time: 33.55s **Requests/s: 2'980** Latency: 58.2ms
	Smart load balancer	Total time: 41.0s **Requests/s: 2'437** Latency: 61.4ms

Table 14: Results of "unique connections" load test.

The smart load balancer is slightly slower than the static load balancer in each test scenario. In the scenario with the maximum amount of concurrent requests, the latency of the smart load balancers **starts to increase noticeably**. This behavior can be explained, as the smart load balancer has to maintain a lot of internal state which requires locking & asynchronous processing (see chapter "8.5 Challenge: High concurrency").

Reuse connections

To see how the smart load balancer performs with reused connections, another set of load tests was run. In this test mode the clients make as many requests as possible in a given timeframe. The following table shows the results of those tests:

Test scenario	Load balancer	Result
Mode: Reuse connections **Duration**: 2 minutes **Concurrency**: 100 request	Static load balancer	Total requests: 530'600 **Requests/s: 4'418** Latency: 24.01ms
	Smart load balancer	Total requests: 1'229'354 **Requests/s: 10'243** Latency: 9.8ms
Mode: Reuse connections **Duration**: 5 minutes **Concurrency**: 500 request	Static load balancer	Total requests: 1'212'999 **Requests/s: 4'042** Latency: 124ms
	Smart load balancer	Total requests: 2'924'215 **Requests/s: 9'745** Latency: 51ms

Table 15: Results of "reuse connections" load test.

In those load test scenarios, the smart load balancer **outperforms** the static load balancers massively. As the smart load balancer has to change its internal state less often (because connections stay open), the processing performance is increased.

9.2.3 Conclusion of load test

The load test showed that the smart load balancer performance is highly influenced by its internal state management. If clients do not reuse connections, the smart load balancer has to update its routing table on every new client connection (see chapter "8.4 Balancing strategy"). As a result, the performance of the smart load balancer is worse than the performance of the static load balancer. The more concurrent unique connections are sent, the more the performance difference grows.

If clients reuse connections, the smart load balancer does not need to update its routing table and fields like the active connections. This improves the performance massively. In this case the smart load balancer outperforms the static load balancer.

With this load tests, the fulfilment of the following requirement should be tested:

- The external load balancer overhead has to be less than 10%.

The load tests showed that this requirement **cannot be judged with "ok" or "not ok"**. The overhead of the smart load balancer depends on the given workload and the client's behavior. The maximum of 10% overhead could be achieved if the workload is well known and the smart load balancer is scaled accordingly (create multiple instances of the smart load balancer).

10 Conclusion

This thesis started with the premise that maintaining a large, productive OpenShift cluster can and should be nearly risk and downtime free. OpenShifts architecture supports this premise, as every component of the OpenShift cluster is scaled and redundant. Thus, every component normally can be updated during full operations. Multiple years of operating an OpenShift cluster at Swiss Federal Railways showed that there were situations where it was not possible to update OpenShifts components without a downtime or the update introduced a big availability risk.

This thesis started with the idea to use multiple OpenShift clusters to solve those problems. The idea was inspired by OpenShifts rolling update approach, that enables application deployments during full operations. The same principle could be used to update OpenShift clusters without downtime and risk. To achieve this, only one peace was missing: a "smarter way" to balance the incoming traffic to multiple OpenShift clusters.

To create a prototype of this "smarter" load balancer and to test the above mentioned idea was the goal of this master thesis. The prototype was created and tested using general functionality tests and load intensive tests. The prototype did fulfil most of the requirements. One requirement could not be measured the way the requirement was formulated. The requirement specified how much overhead in percent the smart load balancer could have. The results of the load tests showed, that the overhead of the smart load balancer depends on the given workload and the client's behavior. Thus it was not possible to judge the requirement with "ok" or "not ok". The smart load balancer can fulfil the given overhead requirements if it is adapted to a well-known workload.

Based on those results, the smart load balancer prototype proves that the **thesis premise of a risk and downtime free OpenShift cluster maintenance is satisfied**. It is possible to minimize the risk during OpenShift cluster maintenance, using multiple OpenShift clusters with the the rolling update approach combined with a smart load balancer.

10.1 Lessons learned

During this master thesis I learned a lot about the internals of load balancers. Especially about optimizing performance of a golang based application using messaging, state management and locking. Also the golang tools for analyzing performance bottlenecks and optimizable code parts were new to me. Lastly, the server name indication (SNI) was a new concept that I learned about during this thesis.

Using the prototype of the smart load balancer showed that the internal state management has a massive impact on the performance. The internal state management can only be optimized up to a certain level. At some point, locking the internal state is always necessary and introduces a bottleneck.

10.2 Outlook

The prototype showed that the basic idea of balancing to multiple OpenShift clusters is practicable.

Redhat

The idea to have a rolling update for OpenShift clusters, including a smart load balancer or something similar, will further be discussed with the product vendor Redhat. They will be informed about the results of this thesis. Redhat could integrate such a solution directly into OpenShift.

Swiss Federal Railways

Alternatively, Swiss Federal Railways could build the solution itself. The prototype of the smart load balancer is not production ready. There are better optimized software load balancers out in the field, but they do not know about OpenShift and its concepts. The results of this master thesis could be adapted to such a software load balancer, such as nginx [40] or any load balancer of a public cloud vendor. Swiss Federal Railways will possibly follow up with vendors of those load balancers on this approach. Then vendors could integrate the smart load balancer techniques into their products and Swiss Federal Railways would install them in their OpenShift setup.

[40] (Nginx, 2018)

11 Glossary and references

11.1 Source Code

The source code of this master thesis is located at:

Description	Source
Source Code of the load balancer	https://github.com/ReToCode/openshift-cross-cluster-loadbalancer
Source Code with the changes made to OpenShift	https://github.com/ReToCode/origin/tree/smart-cluster-lb

The smart load balancer is based on those repositories and libraries:

Description	Source
OpenShift Origin	https://github.com/openshift/origin
gobetween	https://github.com/yyyar/gobetween
kahlys/tcpproxy	https://github.com/kahlys/tcpproxy
bradfitz/tcpproxy	https://github.com/bradfitz/tcpproxy
Gin Server Library	https://github.com/gin-gonic/gin
Logrus: Logging Library	https://github.com/sirupsen/logrus
Gorilla Websockets	https://github.com/gorilla/websocket
VueJS: Frontend Library	https://vuejs.org/
ChartJS Library	http://vue-chartjs.org/#/

11.2 Glossary

Term	Description
12-Factor-apps	A list of twelve points how to build reliable applications for the cloud. (12-Factor-Apps, 2017)
Ansible	An automation software. (Ansible, 2017)
API	Application programming interface. This is used to control an application programmatically. (Gartner, IT-Glossary: API, 2017)
Certificate	A digital certificate to establish trusted connections. Part of the public key infrastructure. (SearchSecurity, 2017)
Channel	A construct in golang to communicate between goroutines.
Cluster	In the context of this thesis: a set of servers for one OpenShift installation.
Cluster-Table	A table of all OpenShift clusters in the smart load balancer.
Container	A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. (Docker, 2017)
Continuous Delivery	The automation of the build, test and deployment pipeline in software engineering. (Fowler, 2017)
Deployment	Software deployment, a process to rollout software or infrastructure to different stages. (Gartner, IT-Glossary: Deployment, 2017)
DNS	Domain Name System. (TechTerms, 2017)
DNSmasq	A DNS caching software. (TheKelleys, 2017)
Docker Image	An image for the container runtime. Is based on a build. Contains the application and can be started on a worker server.
ETCD	A distributed reliable key-value store. Used as OpenShifts configuration database. (Coreos, 2017)
GIT	Source code management software. (Git, 2017)
Golang	A programming language from Google. (Golang, 2017)
Goroutines	A construct in golang to create parallel running processes.
GUI	Graphical User Interface. (Gartner, IT-Glossary: GUI, 2017)
HA-Proxy	High-Availability-Proxy is a software that runs as a load balancer on OpenShift.
HTTP	Hyper Text Transfer Protocol. The protocol for web requests. (Gartner, IT-Glossary: HTTP, 2017).
HTTP-Header	A list of fields in an HTTP Request where client and server can specify details about the request/response.

HTTP-Listener, HTTPS-Listener	A Listener (see Listener) for HTTP or HTTPS traffic.
HTTPS	Like HTTP but with encryption. See HTTP.
IP-Address	An internet protocol address to identify a server. (Gartner, IT-Glossary: IP address, 2017).
Kubernetes	Container orchestration platform. (Kubernetes, Kubernetes, 2017)
Listener	A programmatic construct that opens a port on a server and “listens” for connections.
Load balancer	A load balancer is a software or hardware that distributes incoming traffic to multiple targets to ensure that a target is always available for a client. (Gartner, IT-Glossary: Load-Balancing, 2017)
Master load balancer	A load balancer in front of OpenShift master servers. Balances to the API of those master servers.
Master server	A master server in OpenShift is responsible to orchestrate the cluster and manage state in ETCD.
Nginx	A load balancer software. (Nginx, 2018)
OpenShift	Container orchestration platform. (Redhat, OpenShift, 2017)
PaaS	Platform as a service. (Gartner, IT-Glossary: PaaS, 2017)
Pod	A kubernetes pod specification is a reference to a group of running containers. (Kubernetes, Pod Spec, 2017)
Public key infrastructure	See certificate. (SearchSecurity, 2017; DigitalOcean, 2017)
Race condition	When multiple parallel processes want to access or modify the same information race conditions occur. If this happens, the data that is accessed by those processes can be corrupted. (SearchStorage, 2018)
Replica	A kubernetes specification of how many instances of an application should be running.
Request	A request (in this case web-request) form a client to a server.
Rolling Update	A way to deploy a new version of a software along with the old version while users can use the application during the whole process. (Kubernetes, 2018)
Route	A route describes the hostname or path to one or more deployed applications. An application is reachable from outside the OpenShift cluster when a route is exposed on the HA-Proxy. (Redhat, Route Spec, 2017)
Router host	Invented name for this thesis. It is a worker server where a HA-Proxy is running.
Routing-Table	A table of OpenShift route objects in the smart load balancer.

Scheduler	Scheduler component of the smart load balancer. See 7.5.3 The smart load balancer.
Scrum	A software engineering methodology. (Scrum.org, 2018)
SDN	Software Defined Network. (Gartner, IT-Glossary: SDN, 2017)
SIGTERM	A unix termination signal. (GNU, 2017)
SLA	Service Level Agreement.
Smart load balancer, SmartLB	Invented name of the load balancer that is built in this thesis. See 7.5.3 The smart load balancer.
SNI	Server Name Indication, a specification on how to detect which application a client wants to reach. See also (TheSSLStore, 2017).
Stateless	A stateless application has no request or client state on the server side. Each request is handled uniquely without a session or anything similar.
Statistics Handler, StatsHandler	Statistics Handler component of the smart load balancer. See 7.5.3 The smart load balancer.
Swiss Federal Railways	The company that operates the national public transportation of Switzerland.
TCP	Transfer Control Protocol, a communication protocol. (SearchNetworking, 2018)
TLS	Transport Level Security. A security protocol to encrypt communication. (Mozilla, MDN: TLS, 2017)
Traffic load balancer	A load balancer in front of all applications that are running on OpenShift. Balances the application traffic to the HA-Proxy.
Web UI	See GUI.
Websocket	A technology to keep open TCP sockets starting out of a HTTP request.
Worker server	A worker server in OpenShift hosts application containers.

11.3 Table of figures

11.4 Table of tables

11.5 References

12-Factor-Apps. (2017, November 13). *12-Factor-Apps*. Retrieved November 13, 2017, from 12-Factor-Apps: https://12factor.net/

Ansible. (2017, November 13). *Ansible: Automation for everyone*. Retrieved November 13, 2017, from https://www.ansible.com/

Coreos. (2017, November 13). *ETCD*. Retrieved from https://github.com/coreos/etcd

DigitalOcean. (2017, November 4). *High-Availability*. Retrieved November 13, 2017, from https://www.digitalocean.com/community/tutorials/what-is-high-availability

Docker. (2017, November 13). *What is a Container*, 1. (Docker) Retrieved November 13, 2017, from Docker: https://www.docker.com/what-container

Fowler, M. (2017, September 12). *Continuous Delivery*. Retrieved November 13, 2017, from https://martinfowler.com/bliki/ContinuousDelivery.html

Gartner. (2017, November 13). *IT-Glossary: API*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/application-program/

Gartner. (2017, November 13). *IT-Glossary: Deployment*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/deployment/

Gartner. (2017, November 13). *IT-Glossary: GUI*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/gui-graphical-user-interface/

Gartner. (2017, Dezember 12). *IT-Glossary: HTTP*. Retrieved Dezember 12, 2017, from https://www.gartner.com/it-glossary/http-2-0/

Gartner. (2017, Dezember 12). *IT-Glossary: IP address*. Retrieved Dezember 12, 2017, from https://www.gartner.com/it-glossary/ip-address-internet-protocol-address

Gartner. (2017, November 13). *IT-Glossary: Load-Balancing*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/load-balancing/

Gartner. (2017, November 13). *IT-Glossary: PaaS*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/platform-as-a-service-paas/

Gartner. (2017, November 13). *IT-Glossary: Scalability*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/scalability/

Gartner. (2017, November 13). *IT-Glossary: SDN*. Retrieved November 13, 2017, from https://www.gartner.com/it-glossary/software-defined-networks/

Gartner. (2017, Dezember 19). *IT-Glossary: TCP*. Retrieved Dezember 19, 2017, from https://www.gartner.com/it-glossary/tcpip-transmission-control-protocolinternet-protocol/

Git. (2017, November 13). *Source-Code-Management*. Retrieved November 13, 2017, from https://git-scm.com/

GNU. (2017, November 13). *Termination Signals*. Retrieved November 13, 2017, from https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html

GoByExample. (2018, January 8). *Channels*. Retrieved January 8, 2018, from https://gobyexample.com/channels

GoByExample. (2018, January 8). *Goroutines*. Retrieved January 8, 2018, from https://gobyexample.com/goroutines

Golang. (2017, Dezember 19). *Programming Language*. Retrieved Dezember 19], 2017, from https://golang.org/

Kubernetes. (2017, November 13). *Kubernetes*. Retrieved November 13, 2017, from https://kubernetes.io/

Kubernetes. (2017, Dezember 12). *Pod Spec*. Retrieved Dezember 12, 2017, from https://kubernetes.io/docs/concepts/workloads/pods/pod/

Kubernetes. (2018, January 11). *Rolling Update*. Retrieved January 11, 2018, from https://kubernetes.io/docs/tutorials/kubernetes-basics/update-intro/

Mozilla. (2017, Dezember 19). *MDN: Http Header*. Retrieved Dezember 19, 2017, from https://developer.mozilla.org/en-US/docs/Glossary/Header

Mozilla. (2017, Dezember 19). *MDN: TLS*. Retrieved Dezember 19, 2017, from https://developer.mozilla.org/en-US/docs/Web/Security/Transport_Layer_Security

Mozilla. (2017, Dezember 12). *Web Sockets*. Retrieved Dezember 12, 2017, from https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Nginx. (2018, January 22). *Nginx*. Retrieved January 22, 2018, from https://nginx.org/en/

Redhat. (2017, Dezember 12). *API Changes*. Retrieved Dezember 12, 2017, from https://docs.openshift.com/container-platform/3.6/rest_api/index.html#rest-api-websockets

Redhat. (2017, November 13). *OpenShift*. Retrieved November 13, 2017, from https://www.openshift.com/

Redhat. (2017, Dezember 12). *Route Spec*. Retrieved Dezember 12, 2017, from https://docs.openshift.com/container-platform/3.6/rest_api/openshift_v1.html#v1-routespec

Redhat. (2017, Dezember 12). *Router-Plugin*. Retrieved Dezember 12, 2017, from https://docs.openshift.com/container-platform/3.6/architecture/networking/haproxy-router.html

Scrum.org. (2018, January 22). *What is scrum*. Retrieved January 22, 2018, from https://www.scrum.org/resources/what-is-scrum

SearchNetworking. (2018, January 7). *TCP*. Retrieved January 7, 2018, from http://www.searchnetworking.de/definition/TCP-IP-Transmission-Control-Protocol-Internet-Protocol

SearchSecurity. (2017, October 11). *Public key infrastructure*. Retrieved November 13, 2017, from http://www.searchsecurity.de/definition/PKI-Public-Key-Infrastruktur

SearchStorage. (2018, January 8). *RaceCondition*. Retrieved January 8, 2018, from http://www.searchstorage.de/definition/Race-Condition

TechTerms. (2017, November 13). *Domain Name System*. Retrieved November 13, 2017, from https://techterms.com/definition/dns

TheKelleys. (2017, November 13). *DNSmasq*. Retrieved November 13, 2017, from http://www.thekelleys.org.uk/dnsmasq/doc.html

TheSSLStore. (2017, November 17). *What is SNI*. Retrieved Dezember 19, 2017, from https://www.thesslstore.com/blog/what-is-sni/