DELFT UNIVERSITY OF TECHNOLOGY

BACHELOR THESIS

POC High-available Application

Authors:
Marco BOOM
Onne VAN DIJK
Max GROENENBOOM
Joost WOONING

Supervisor:
Georgios GOUSIOS
Coordinator:
Otto VISSER
Client:
Coert BUSIO

June 26, 2017

TU Delft

Preface

This document is the bachelor thesis of Marco Boom, Onne van Dijk, Max Groenenboom and Joost Wooning. On the Delft University of Technology students are tasked with a project that marks the end of their bachelor. This bachelor thesis is our report of that project.

The project was commissioned by ProRail, the rail infrastructure manager of the Netherlands. The goal of the project is to produce a proof of concept, high availability application to prove it is worth investing in high availability and zero downtime to the management of ProRail.

This project would not have been possible without the help of many people. We would like to thank a few people in particular. First of all we would like to thank Georgios Gousios for his general guidance. Secondly we would like to thank Otto Visser for his guidance and specifically the feedback we received halfway the project. Furthermore, we would like to thank Peter Benschop and Coert Busio for their effort to help us get the resources and contacts we needed along with their guidance during the weekly meetings at ProRail. At last, we would like to thank all others at ProRail who gave demonstrations or provided us with the help we needed.

Marco Boom
Onne van Dijk
Max Groenenboom
Joost Wooning
Delft, June 2017.

Abstract

ProRail uses multiple custom built software applications to control the train infrastructure in the Netherlands. At this moment these applications experience downtime when they are updated. This interrupts processes that are related to controlling the train services. ProRail asked us to redesign an existing application in such a way that updates can be applied with zero downtime.

To solve this problem we use redundancy with automatic load balancing. We compared multiple solutions and we chose to use Docker's swarm functionality. We then redesigned the non-critical application Brugkijker, an application to monitor trains approaching a bridge. In the end we could successfully run multiple instances of Brugkijker in Docker and show how the application can be incrementally updated, inducing no downtime for the service as a whole and without losing data coming from upstream. This proposed approach can be used by ProRail to make other non-critical and critical applications highly available.

Contents

Chapter 1

Introduction

This chapter gives a basic understanding on how we got to this project and what this project is exactly. First it briefly explains the history of the company ProRail and why they approached the Delft University of Technology with their project. Furthermore it discusses the details of the bachelor's thesis.

1.1 ProRail

ProRail is the Dutch rail infrastructure manager. The company controls train services, performs the maintenance of the railway infrastructure and the is responsible for safety and security of the railways. It came in existence in 1995 when the 'Nederlandse Spoorwegen' (NS - In English: 'Dutch Railways' - the main Dutch railway corporation) went through a reorganization. Before this time, the NS held a monopoly on the Dutch railways. The reorganization meant to change that. It would not have been possible to remove the NS's exclusive rights on the dutch railways while it still managed the rail infrastructure. This is the reason ProRail became an independent company. The company is currently owned by Railinfratrust B.V., which is owned by the Dutch Government.

As can be expected of a company with the size and responsibility of ProRail, a lot of different kinds of software is used by it. This means lots of systems can break and – depending on the design – take the whole system down. A second problem with high amounts of software is the maintenance. During updates the system will not be usable, and sometimes for prolonged periods.

This is deemed undesirable by ProRail and therefor they have asked the Delft University of Technology to have a team of students prove that the software can be made highly available, even during updates. In other words, ProRail wanted us to (partially) redesign their software and deliver a proof of concept as a demonstration of the possibilities.

1.2 Bachelor's Thesis

The bachelor's thesis is the final project that is carried out at the end of the bachelor Computer Science at the Delft University of Technology. The project is structured and shaped like a real-world project, and is commissioned by a real-world company or organization.

Chapter 2

Summary

ProRail is the Dutch rail infrastructure manager. Its dedicated applications are specially built to perform tasks related to controlling train services. Most of these applications should be highly available, during downtime of critical systems no single train can drive on the Dutch railway network. Therefore, it is hard to update these applications. The goal of this project is to show in a proof of concept that an existing application can be made highly available and updates can be applied without downtime.

For the proof of concept the non-critical application Brugkijker is used. Brugkijker is an application that bridge operators can use to determine when a bridge can be opened. It shows how long it takes for a train to pass the selected bridge, how long it takes before the next train arrives at the bridge and a list of upcoming trains. With this information a bridge operator can plan when to open and when to close a bridge.

We employ different technologies and platforms to recreate Brugkijker and simulate it's place in ProRail's infrastructure, because we don't have access to the existing application. We created a simulation back-end server to replicate data from ProRail's own back-end directly out of log files. We implemented a front-end data server to process this data and serve it to the Brugkijker application. We recreated Brugkijker as a web application that can also run on the server side, in order to always have an instance running and preserve state.

We use Docker to make Brugkijker highly available. Docker is a container platform with built-in load balancing technology. Applications in Docker, like Brugkijker, run in isolated containers. Multiple instances can be run easily at the same time to achieve redundancy. The containers provide all resources that are necessary to run the applications and provide an isolated environment for each copy of the application. In swarm mode Docker enables load balancing and cluster management. Multiple nodes can be added as a worker or manager, which removes single points of failure. Finally, we were able to show that Brugkijker stays available even if a node were to fail, and that updates and rollbacks can be performed without downtime.

Chapter 3

Problem

This chapter describes the main aspects of the problem and our proposed solutions to solve them. Also all the unexpected problems we encountered and our approach to solve them are mentioned.

3.1 Problem Definition

The main problem ProRail has presented us with is: How to make an existing application highly available with as little work as possible. This is complicated by the size of ProRail's systems. With as much software running as they have, it is hard to introduce high availability throughout the entire system.

The actual product ProRail wants us to deliver, is a proof of concept high available application based on one of their existing applications. The application at issue is called 'Brugkijker' (In English: 'Bridge Watcher') application. This application makes it possible for bridge operators to monitor trains that are headed towards their bridges, so they know when it is safe to open a bridge and for how long it can stay open.

Initially ProRail wanted another product (VIEW), but this product proved to be the core of their entire system, making it impossible to rewrite only this part in such a short time span without fully knowing its place in the network. This product is within the inner layer as described in the next section. It was deemed preferable if a program from the outer layer was reimplemented to prove the concept of high availability. Both VIEW, the core of the system, and Brugkijker can be seen in Figure 3.1.

3.2 Problem Analysis

An approach to change the large amount of intertwined software is to gradually make the system more modular. In a modular system it is easier to update a single part of the system without having to restart the entire system. Redundant modules can run in parallel, allowing to update parallel copies of the module. This system in itself can make a huge difference in downtime when a component has to be updated, and can even completely negate downtime if applied well.

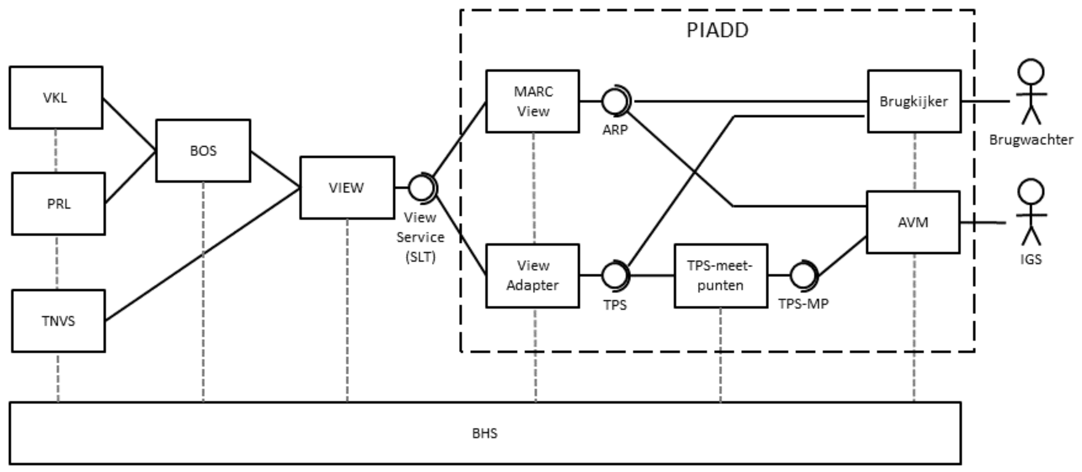


FIGURE 3.1: Overview of Brugkijker's placement within the relevant part of ProRail's IT structure (all parts that Brugkijker directly or indirectly communicates with). Also featured is AVM which is a similar program, but not mentioned any further. The figure also shows how interconnected VIEW is, and how difficult it would have been to reimplement this application.

ProRail's IT infrastructure is ordered in a layered approach. It consists of three different layers, where software in a layer can only receive data from software in lower layers. This way the software in the inner layer can be considered the most critical for other layers. Keeping this in mind it would be easy to modify the software in the outer layer, as no other software can depend on it so it cannot cause a system-wide failure. Software in the inner layers is harder to fix on the other hand, as most of the system or even the entire system can depend on it. This means any small error could cause a system-wide failure.

For the actual application, we considered a web application to be preferable. It would be much easier to maintain and develop, and users would not have to install any additional software on their machines. This actually makes the application somewhat more available to end users, since the update for the end user is a simple and fast page refresh. ProRail originally used a Java applet for Brugkijker, but since Java applets are deprecated recently[1], we decided not to develop on this platform.

The Brugkijker application is rather unique in the sense that it will only be used by a small amount of people, at most a hundred bridge operators in the Netherlands that happen to control one of the 54 bridges over which a railway runs. It is important though that this information is well up to date. If it is not, a bridge might be opened when a train is nearing the bridge causing unnecessary delays for train passengers or ships.

[1]JEP 289: Deprecate the Applet API, `http://openjdk.java.net/jeps/289`, consulted on 24-06-2017

3.3 Requirements

This section describes the requirements using the MoSCoW-model. This method specifies for each requirement how important they are: 'Must have', 'Should have', 'Could have' and 'Won't have'. The Must have, Should have and Could have requirements respectively have to be, should be and may be implemented in the final product. Won't have requirements have been considered at the start of the project, but have been deemed unrealistic, unfeasible or unnecessary.

Since the only real requirement from ProRail was that the application must be highly available, we placed all Brugkijker-specific requirements under lower categories than Must have.

Must haves

- Brugkijker must run on a high available server, meaning nodes on a cluster can shut down at any time without causing downtime.
- Brugkijker must automatically reconnect to a the web server if the WebSocket closes.
- Brugkijker must be able to be updated with zero downtime.

Should haves

- Brugkijker should be able to display when trains arrive at a bridge.
- Brugkijker should display for how long a train is on a bridge.

Could haves

- Brugkijker could be able to watch multiple bridges at the same time.
- Brugkijker could also be able to watch random stations instead of only bridges.
- Brugkijker could have the possibility to set alarms for when a train approaches a selected bridge.

Won't haves

- A data simulating and generating back-end is included as a realistic testing environment.

Chapter 4

Background

This chapter identifies fundamental problems that are related to high availability. In section 4.1 we explain what availability and high availability mean and what kind of problems are related to (high) availability. Furthermore, section 4.2 describes relevant information about networking in general.

4.1 Availability

According to Piedad and Hawkins, 2001 availability of a system means that a system is ready for use when the user needs it. They defined three levels of availability:

1. High availability: The system or application is available during specified operating hours with no unplanned outages.

   The system or application has appointed uptimes during which it will be available. Outside these hours the server will or can be down, but within them even errors or other problems should not completely shut the system down.

2. Continuous operations: The system or application is available 24 hours a day, 7 days a week with no scheduled outages.

   The system or application should always be up. Updates should be applied without having to shut the server down, but when errors occur there is no guarantee that the system keeps running.

3. Continuous availability: The system or application is available 24 hours a day, 7 days a week with no planned or unplanned outages.

   The system or application should always be up. Updates should be applied without having to shut the server down, and errors or other problems should also not shut the system down. This means the server ideally is up anytime, even when problems occur.

With these levels of availability come specific methods to maintain or update the applications. For the first level, high availability, maintenance can be planned outside operating hours. This method does not need special attention, as long as the maintenance can be finished outside the operating hours. Otherwise, the maintenance process needs to be redesigned. Possible solutions are: splitting the process in smaller independent parts or using some kind of fallback system.

For maintenance of applications in the second or third level of availability, continuous operations or continuous availability, other solutions are needed. Because these applications needs to be available 24 hours a day, 7 days a week with at least no scheduled outages, it is not possible to schedule maintenance for which these applications will be temporarily unavailable. So, these applications needs to be maintained in such a way that users or other machines can continue to use the application without interruptions. To solve this issue, we need to dive deeper in particular problems that can occur when applications will be designed for the second or the third level of availability. For convenience the term high availability will be used in the rest of the report to indicate continuous availability and not the definition of high availability mentioned above.

4.2 Network communication

Sometimes, it is not possible to do maintenance without shutting down a system. Think for example of replacing hardware. To reach level 2 or 3 of availability, there needs to be a mechanism to 'hand over' tasks to other machines. These handovers should not interrupt the services. In order to understand which problems can occur during these handovers, it is important to understand how systems or applications communicate with each other.

A standard model of network communication is Open System Interconnections Reference Model (Zimmermann, 1980). In the OSI reference model components participating in network communication are divided in seven layers. Basically layers 1, 2 and 3 are dedicated to handle (local) network traffic while layers 4, 5, 6 and 7 are dedicated to sending and receiving data. Therefore, only these last four layers are relevant for continuous operations and continuous availability. Based on these layers we show what kind of problems could occur when distributed computing service are moving to other machines. Although network devices do not need to have an implementation of network communication based on the OSI reference model, the same problems are still relevant.

4.2.1 Layer 4

Layer 4, the Transport layer, is the lowest layer in the OSI reference model that is guaranteed to communicate directly with the source and target machine (Day and Zimmermann, 1983). The lower layers communicate with only intermediary network devices like switches and routers. Layer 4 handles for example connecting and disconnecting with a target device, sending and receiving messages and listening. There are a lot of communication protocols available for this layer. Two well-known protocols are the Transmission Control Protocol or TCP and the User Datagram Protocol or UDP (Fall and Stevens, 2011). The first one is optimized for accurate delivery while the second one is optimized for timely delivery (Postel, 1980).

An interesting aspect about TCP is that TCP provides a data stream service, that means a TCP connection stays open until a server or client decides to close the connection (Postel, 1981). In terms of availability, this could mean, depending on the actual service, that a client needs a TCP connection that stays open during maintenance. This leads to the first problem: handing over data stream connections.

One way to do this is closing the connection on the first device and opening a new connection on the second device. But this method does induce downtime. From the moment that the connection closes until the moment that a new connection is opened, the client cannot communicate with the server, resulting in down time. Other solutions are opening a new connection first before closing the old connection or using a Software Defined Network based method (Binder, Boros, and Kotuliak, 2015).

4.2.2 Layer 5

The next layer in the OSI reference model is layer 5, the Session layer (Zimmermann, 1980). Sessions are managed in this layer. It depends on the transport protocol and the application whether sessions are implemented in this layer, in layer 7 (Application layer) or not at all. However, this choice does not change the concept of sessions. Sessions can be stateful or stateless. For the latter, a connection handover would not cause big problems as long as the device that takes over the connection provides exactly the same service like the device that lost the connection. With stateful sessions the second problem arises: handing over sessions.

Session data could be stored on the server. Shutting down a system could result in a loss of session data. Removing session data could have impact on users, for example in an HTTP web server, users could be immediately 'logged out' if the web server stores information about authentication and authorization of users in session data. Therefore, sessions need to be transferable. A common solution is the use of a memcached session manager (Cukier, 2013). Servers that are responsible for a session update a copy of the session on the memcached session manager. When a server goes down, other servers can use the memcached session manager to fetch sessions that lived on the original server.

4.2.3 Layer 6

The sixth layer in the OSI reference model, the Presentation layer, translates data from packets to data structures and vice versa (Day and Zimmermann, 1983). Also data formatting tasks like encoding/decoding, encryption/decryption and compression/decompression are handled inside this layer. Actually, this layer is used to process data and is not related to sending or receiving data or managing connections. Therefore, there are no problems related to continuous operations or continuous availability in this layer.

4.2.4 Layer 7

Finally, the highest layer in the OSI reference model, the Application layer, contains application specific problems and solutions related to continuous operations and continuous availability (Zimmermann, 1980). Even problems in layer 4 and layer 5 (handing over data stream service connections and stateful sessions) could be solved inside this layer.

We will take a look at the HTTP protocol, the WebSocket protocol, file service, database systems and message passing systems. These services and systems are used by ProRail for monitoring of the Dutch railway network.

HTTP Protocol

The HTTP protocol is a stateless protocol that is build on the TCP protocol (Fielding et al., 1999). Originally, the HTTP protocol opened a TCP connection for every single request and immediately closed the connection after the response was received. In later specifications TCP connections are declared persistent by default. Although, some web servers automatically close the connection after a specified timeout (Apache, 2017a, Microsoft, 2017). For example, in Apache 2.2 and above TCP connections will be closed after a timeout of 5 seconds by default (Apache, 2017a, Apache, 2017b). Although a connection can be persistent, there is no need to hand over the connection. Because the HTTP protocol is stateless, opening a new connection and closing the old connection when the response of the last request sent over the old connection has been returned is a safe way to switch to another server without downtime.

There is a bit more difficulty when web applications use sessions. In this case session data needs to be accessible by the devices that take over the connections, even when the original web server will be turned off. When these devices can access the session data new TCP connections can be opened for each user. After that, the old connection can be closed.

WebSocket Protocol

The WebSocket protocol is built on the TCP protocol and uses HTTP for initialization, once it is initiated it does not depend on the HTTP protocol any more (Pimentel and Nickerson, 2012). The HTTP protocol can be upgraded to the WebSocket protocol (Fette, 2011). This protocol provides a full-duplex connection between server and client which can be used for transmitting messages (Pimentel and Nickerson, 2012). WebSockets can be implemented in a stateless (justin, 2015) or stateful (Pimentel and Nickerson, 2012) way.

While handing over stateless WebSockets could be done by handing over the TCP connection, handing over of stateful WebSockets is harder. A WebSocket server can run a single instance with living objects and variables in the global scope. In comparison with HTTP web servers, the latter can have isolated sessions and session data for each user and cannot have global scope variables and objects. WebSockets however can have a global scope. This could result in a hard problem when WebSocket clients from one WebSocket server have to be distributed to multiple other WebSocket servers.

File service

When applications running in continuous operations or continuous availability mode use files in runtime, those files need to be available too. In a high available platform these files should be available to multiple nodes. Read-only files do not require special attention for this feature, because every node can receive a copy and read the file. Files that need to be modified require some kind of collaborative real-time editing functionality.

Besides the availability of files to multiple nodes, the file service itself should not become a single point of failure. Distributed file systems (Satyanarayanan, 1993) can provide a solution.

Message passing

A message passing system is a distributed system that provides functionalities to pass messages between nodes (Gropp et al., 1996). A common type of middleware that is used to implement message passing is Message-oriented Middleware (MOM) (Curry, 2004). In continuous available environments multiple problems could arise. Message passing systems can have similar problems to those of the WebSocket protocol. For example, a message passing system could be stateful, like a WebSocket server. So, states should be transferable.

Another problem arises when a message passing system uses queues. Messages that are queued for a server that needs to be turned off should be rerouted to other servers. For stateful environments it is important that the right states are ready on the devices that will process the rerouted messages.

Database systems

There are a lot of different database systems available. And all try to fulfill a set of requirements like ACID or BASE (Tudorica and Bucur, 2011). According to the CAP theorem it is not possible to fulfill some of these requirements in combination with high availability. It is not possible to choose more than 2 guarantees from the following guarantees: Consistency, Availability, Partition tolerance (Brewer, 2012).

Another problem is fault isolation (Cai and Leung, 2002). A cluster of multiple nodes can split up in independent subclusters. This could be caused for example by a network failure. These subclusters need to decide which subcluster continue to operate. When all subclusters continue to operate the system could develop the split brain syndrome. This means that multiple subclusters claim to represent the whole cluster operating on the same dataset while these subclusters are unaware of each other.

4.2.5 Applying updates

Switchover methods that are applied in case of failing components cannot be straightforwardly used for applying updates. Above problems and solutions are independent for each component in the chain. For example, switching over from one web server to another web server does not necessarily require change of behavior of clients or back-end servers. This is not the case when updates are applied.

Updates could contain changes for one tier or multiple tiers. In the case where just one tier needs to be updated, the update can be applied seamlessly. Otherwise, a mechanism needs to ensure that no compatibility issues can occur. This could be done by making the application backwards compatible, ensuring that all components, including all client instances, update at the same time or supporting multiple concurrent editions. The second solution conflicts with the requirements of high availability. During the update users cannot use the application. The third solution can contain problems related to data storage. For example an updated middle and presentation tier need to communicate with an updated data tier. To prevent downtime a new version of the application could be deployed next to the old version on all tiers. At the data tier a transition of data is needed to go from the old version to the new version. But at the same time, this data needs to be available for the old version.

This problem can be solved by techniques like Edition Based Redefinition (Choi, 2009). This is a technique developed by Oracle. It supports editioning of database objects like functions, procedures, views, etc. Tables cannot be editioned. Users or application can be assigned to the right edition and use the procedures, views and other objects in that edition. This means that, for example, the same view uses different columns in different editions. In addition, Oracle built cross-edition triggers to replicate data between old and new columns. With this method, only data that is subject to changes is replicated.

4.2.6 Load balancing

A load balancer can solve a couple of the problems mentioned above (Kopparapu, 2002). But it could introduce a new single point of failure. Therefore, multiple load balancers are needed to prevent a new single point of failure. At least a failover setup is needed to keep the cluster online. This means that (at least) two load balancers share a heartbeat channel in an active-passive mode. The passive load balancer monitors the active load balancer. When the active load balancer goes down, the passive load balancer becomes active, takes over the ip-address(es) and takes over all tasks of the deactivated load balancer.

4.2.7 Internet connection

The last part of the high availability chain is the network- and internet connection. Even when all components inside a network do not have a single point of failure, an internet connection is single point of failure if it is the only connection that connects the network with the internet. Routers can be set up in a failover mode like load balancers. This setup ensures that when a router goes down another router takes over the connection.

Besides a failing router, the WAN connection can fail too. Therefore, a multiple WAN setup can remove the final single point of failure. In this setup the routers are connected to multiple WAN connections. When one connection fails, the routers can switch over to other active WAN connections.

Similar techniques can be used to create a multi-site failover setup. This kind of setups supports disaster recovery (Chang, 2015). This ensures that when one site is not available another site can takeover the tasks.

Finally, in the DNS multiple IP addresses can be attached to the same fully qualified domain name (Postel, 1994). In this way, another failover mechanism is implemented by using the DNS. Clients that use the DNS automatically search until they found an active IP address (Singh and Schulzrinne, 2007).

Chapter 5

Server platforms

First we describe which platforms were considered for the server side to manage the sessions. The most challenging part of achieving high availability, is at the server side. It could be considered impossible to achieve zero downtime without redundancy at the server side: Crashes and errors are bound to take place. If the server cannot immediately swap to another redundant node and instead has to restart the crashed software, some downtime would be inevitable. This downtime would increase with the complexity and the required start-up time of the server-side software.

When considering redundancy, some new problems appear. First of all, the server has to distribute and manage the work. Secondly, when many redundant servers or nodes are running at the same time, this has to happen efficiently and these nodes have to be managed. Lastly, if a node happens to crash, its sessions have to smoothly continue on other nodes. The last problem is solvable at both the server and the client side. The first two problems, however, can only be solved at the server side. Fortunately, many solutions have been developed that are now being used by many of the largest companies of the world. This section describes the possible solutions we have considered or at least compared, and why we decided to either use or discard them.

5.1 Physical servers

The most obvious way to achieve high availability is to have multiple servers distributed across different geographical locations. This can – in combination with a load balancer – prevent downtime by having another server as backup. This way, when one of the servers unexpectedly shuts down the load balancer can immediately switch to another server, and because the servers are distributed across multiple locations it is very unlikely that multiple servers fail at the same time. This way it is also possible to upgrade a server: A server can be disconnected from the cluster, upgraded without having incoming connections and lastly reconnected to the cluster. During the downtime of this server, the remaining servers in the cluster can take over. When the first server is up and running again the rest of the servers in the cluster can be upgraded one by one.

Having multiple bare-metal servers sounds like the easiest way to have a highly available application, however there are several downsides to this approach.

First, physical servers require a lot of maintenance. Every time some upgrade process is required, every server has to be upgraded by hand, since it is difficult to automate the process of upgrading for physical server. Furthermore, adding additional hardware requires lots of time, new servers must be shipped, installed and placed. Even when simply hiring more servers at a data center, the setup time is still way too much for critical applications.

A second issue is that server purchase and maintenance costs are high. Servers are high level professional devices and therefore pretty expensive. Especially for small scale applications, having the resources of a whole server available is a waste of money and energy.

Another issue is that when multiple applications share a single server it increases security risks. It might for example happen that application A is accessible to users but contains no data, another application B is not accessible by users but does contain sensitive data. If those applications run on a single server it might expose application B its data to users of application A. With a lack of isolation it might also happen that one application causes a fatal error requiring the whole machine to reboot, causing other applications (which might be more critical) to also have downtime.

5.2 Virtual Machines

Virtual machines (VMs) are a way to emulate a computer system on another computer system, which may or may not be the same operating system. VMs have the same capabilities of physical computers and most recent computer systems have some accelerated hardware implementation to optimize the use of VMs. For smaller applications VMs can be hired at data centers at the cost of only a fraction of a whole server.

VMs solve some of the issues which occur when using physical servers to achieve high availability. VMs require less maintenance, most hypervisors (virtual machine monitors) offer an API to automate the upgrading process of the VMs they are managing. Installing additional VMs is easier, they can be set up by duplicating VM images onto other VMs. A schematic overview of a server with VMs can be seen in Figure 5.1.

Full application isolation is also provided by VMs. Since a VM installs a whole new operating system, and all operations of the guest system go though the VM manager, there is no way for a guest system to access the host systems resources.

Because of these advantages VMs have been the default way to achieve isolation and high availability for applications for a long time. However, VMs also have some downsides, VMs require a full operating system to run in the background. This causes that with the use of a VM for each application a large overhead arises, while combining applications on a VM reduces application isolation.

Furthermore, VMs do not solve all high availability issues which occur with physical servers. While boot times are much faster than for physical servers it can still take a few minutes to launch new VM instances, this is still too much for critical systems.

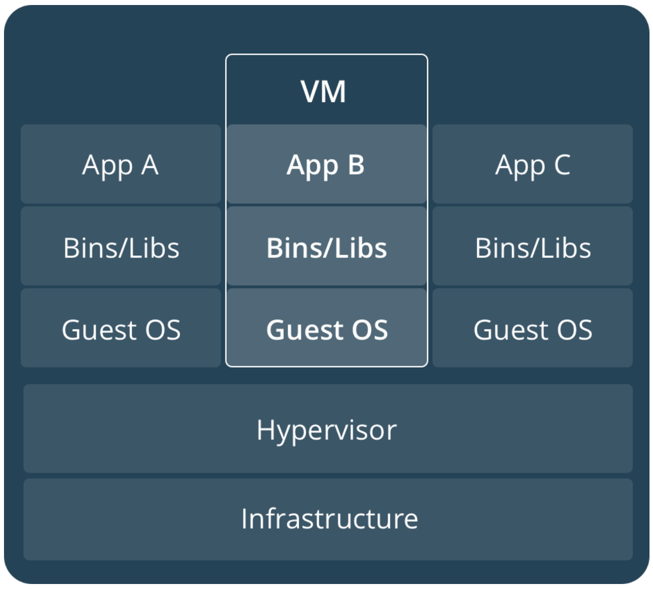


FIGURE 5.1: Overview of a server with a hypervisor and some VMs. As shown, for each VM a guest OS is required. (Source: `https://www.docker.com/what-container`)

5.3 Docker

Docker provides a way to isolate software applications in a container. Containers are often described as lightweight VMs, this is because containers and VMs have two large similarities, both are designed to provide an isolated environment for an application to run and for both this environment is represented as a binary image and can be shared between hosts (*Ebook: Docker for the Virtualization Admin* 2017).

However, there are some fundamental differences between containers and VMs, containers share the resources and infrastructure of the host with each other, thus having low overhead. Because the resources are shared and Docker containers are build from a single image, containers should ideally be stateless and immutable. Having immutable and stateless containers makes it an almost trivial task to duplicate and distribute images, thus making replication and high availability very easy. Figure 5.2 shows a schematic overview of a server running Docker.

Containers have a solution for the problems which occur with physical or virtual machines. Containers are – as mentioned before – highly scalable, they can be launched, shut down, and distributed on the fly in a matter of seconds. With Docker, a swarm of worker nodes (servers) can be created with just a few commands. With this swarm of nodes, no single node is responsible for everything, thus making it no real problem if a node shuts down unexpectedly.

Docker also provides a way to perform rolling upgrades, with these rolling upgrades nodes are upgraded incrementally. An interval time can be specified to make sure there is always a node online, if something goes wrong during the upgrade, a rollback task can be executed.

Since VMs also have some benefits in contrast to containers, for example the possibility to have different host operating systems, it is also possible to have a docker

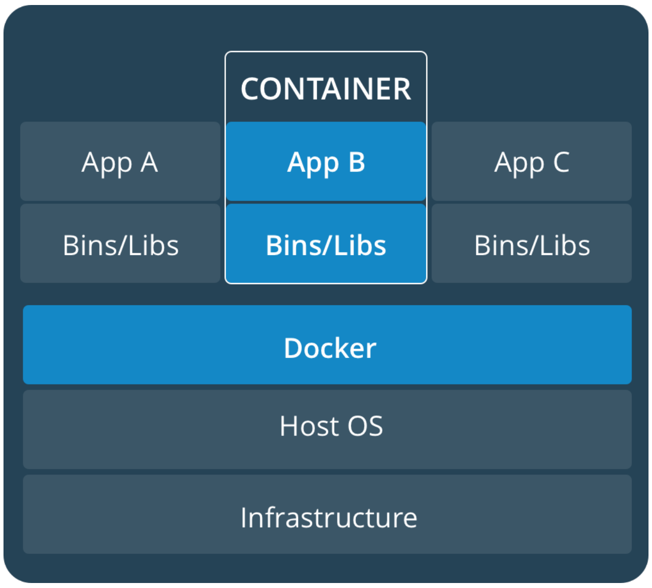


FIGURE 5.2: Overview of how a server with Docker would be configured. Each container has its dependencies and the application itself, the rest of the resources come from the host. (Source: `https://www.docker.com/what-container`)

engine run on a VM. On a VM, Docker has exactly the same possibilities as on a physical machine.

Because deploying and distributing Docker containers is so easy and easily automated, many cloud server providers offer standard Docker swarms which automatically scale (in matters of seconds) according to the current activity. This way no server capacity is wasted and the costs are exactly according to what was used. Another benefit of not wasting server capacity is the environmental impact, which is reduced because more applications can share servers.

5.4 Other solutions

Other available solutions include Kubernetes and Puppet. Kubernetes is a program designed to automate deployment, scaling and operations on application containers across clusters of nodes, this is similar to the Docker swarm features. Kubernetes however does not provide the container software itself, it requires pre-built images made by, for example Docker. Since Kubernetes is much more customizable than Docker swarm its installation process is also much more complicated, for this reason we chose to use Docker swarm instead.

Puppet is a system for describing system configurations. Most of its functionality is already included in Kubernetes and Docker, but Puppet provides some extra features which we didn't require. To reduce the complexity of the system, we preferred a single solution over multiple solutions. This is why we decided not to use Puppet for this project.

We also had to decide on which load balancing software we were going to use. A possibility for this was HAProxy, HAProxy is a widely used load balancer used by many large companies which are specifically beneficial of high availability. Although this seems like a good solution, Docker also provided some basic load balancing software in the Docker Swarm application, we decided it would be easier to use that instead of having to get to understand how HAProxy works.

Chapter 6

Design & Implementation

In this chapter we discuss the design and implementation details of our system and its components. We start with giving an overview of the system as a whole in section 6.1. Then we go into more detail for each component and talk about the frameworks we used to build them and how they work. We discuss the Replay server and Data server together in section 6.2. Then, we talk about the Brugkijker application in section 6.3.

6.1 Overview

We implemented a number of services that are deployed on the Docker Swarm and the user's browser and work together to provide real-time highly available data to the user about the trains approaching the selected bridges. An overview of these services is given in Figure 6.1

6.1.1 Replay server

Because the client couldn't connect us to their data infrastructure in order to provide us with actual real-time data, we were given logfiles instead. These logs contained all the raw data from a particular day that was sent by the VIEW service, which provides a number of ProRails applications with data. The Replay server simulates that day by sending the data from the logfiles at the time that it was logged to our system downstream. It doesn't filter or modify this data at all, this way the main system receives data as if it were actually connected to ProRails infrastructure. Since this Replay server only exists for the purpose of our project we tried to keep the Replay server as simple as possible.

6.1.2 Data server

The first piece of our actual service processes the data sent from upstream. A lot of the data is not used in our application and is filtered out. The rest is processed and formatted for the Brugkijker application to consume. There are multiple instances running of this server in order to achieve the high availability wanted by the client.

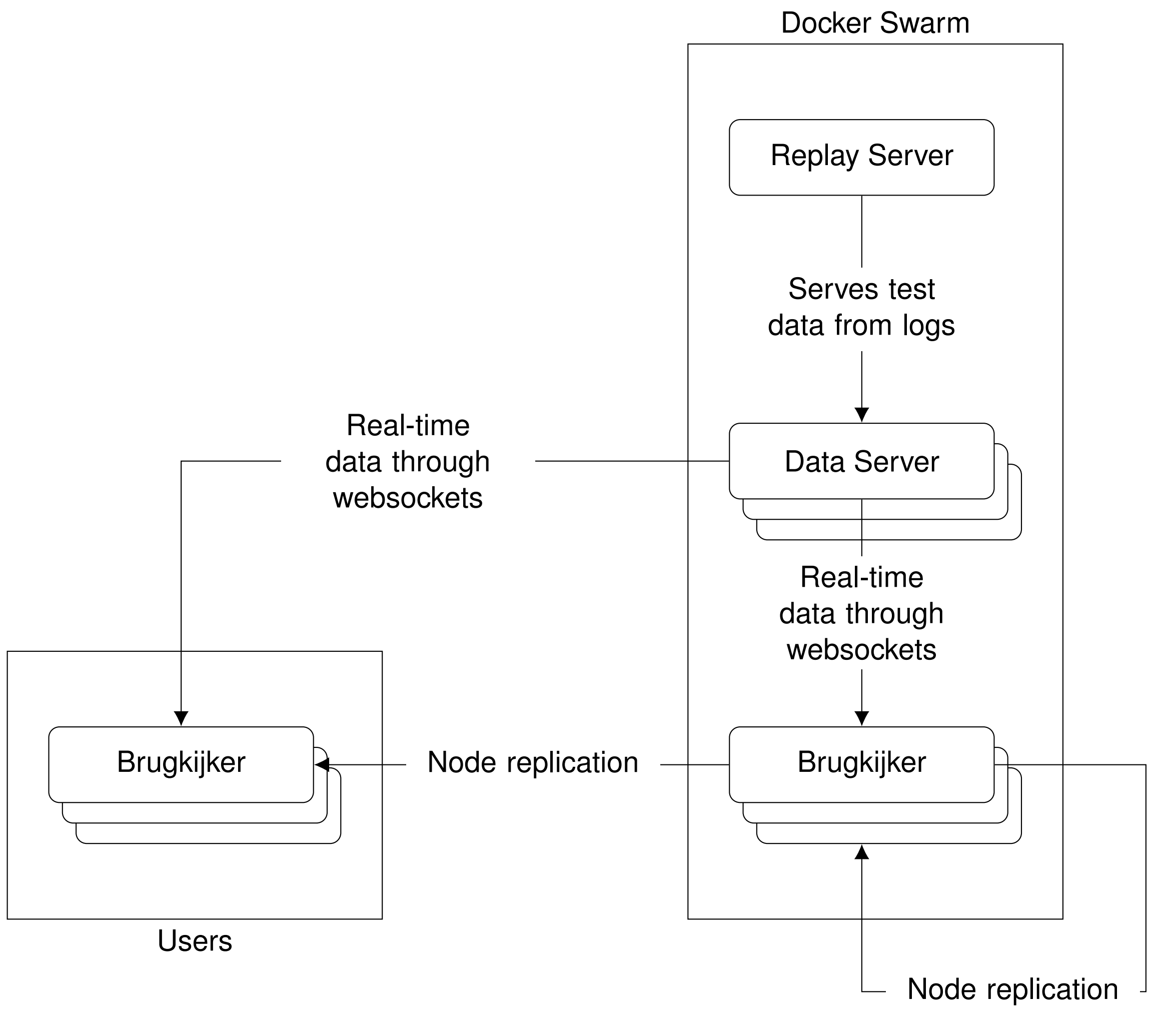


FIGURE 6.1: This diagram shows the interactions between the applications running both in a Docker Swarm and on the user's browsers.

6.1.3 Brugkijker

This application maintains and displays the state of approaching trains based on the data received from the Data Server. It runs on both servers in the swarm and in the user's browsers. Because aforementioned state depends on past data, an instance of it should always be alive so it can be replicated when a new node joins. After this replication the new node will also receive real-time data directly from the Data server. Because state is a pure function of the previous state and new data received, proper handling of this duplication will ensure that state is equal across all nodes and always correct and available as long as there are instances running in the swarm. There are a few big advantages to this approach:

- Maintainability of the system is significantly increased, because code is equal both server and client side.
- Because the internal design of the nodes make state replication incredibly straightforward, distributed highly available data is made easy, while still being as real-time as possible. (See section 6.3)

When running in the browser, Brugkijker will display the state in a human readable fashion.

6.2 Replay and data server

In this section we will discuss the details and specific implementation choices of our back-end servers.

6.2.1 Python

For both the replay and the data server we chose to use Python. We chose for Python instead of Java or C# because of pythons simplicity. Python also has a short learning curve making it easier to use Python for the project members who had no experience in Python yet. We specifically chose for Python 3.x instead of 2.x, because this is the recommendation from Python itself for new projects.

6.2.2 Tornado

We used the Tornado web framework for the data server. Tornado is, besides a web framework, also a library for asynchronous networking. Tornado uses non-blocking i/o operations and it can scale up to tens of thousands of open connections. Due to the fact that our web server only needs to serve static files and needs to be able to keep open WebSockets we chose to use Tornado as our web framework.

6.2.3 Replay server

Since the Replay server is not really part of the actual end application but only serves a purpose to test the real Data server we tried to keep this as simple as possible. Because of this the Replay server does not have a GUI and can only be accessed with the command line.

The Replay server serves the purpose of sending the Brugkijkers input logs we received to the Brugkijker client application. The log files contain a timestamp for each log rule, with this timestamp we can see when this rule should be send to the Brugkijker application. Because of the design with sending messages to Brugkijker when they should occur we can simulate the real time data stream the Brugkijker application would normally attach to.

By default the Replay server starts at the system time and has a normal speed, this makes the similarity to a real data stream the largest. However, for testing purposes we also included the options to start at a different time or increase the speed of the server. These options can be especially useful for remote bridges where – even during rush hour – only a few trains pass per hour.

For sending the different log files simultaneously to Brugkijker when needed the Replay server uses five threads, one for each of the four log files and one main thread for sending the actual messages to Brugkijker. The four log file threads each watch the timestamps in the file and trigger the main thread to send a message to Brugkijker.

6.2.4 Data server

The Data server connects to the Replay server, calculates how long trains must travel until they arrive at a bridge, and then sends this information to the Brugkijker client application. When the client connects to the Data server, the server creates a subscription, knowing for which bridges information needs to be send to which client.

The input data the Data server receives is the track the train is on, at what position on the track and in what direction. Furthermore, it has a list of service control points (in Dutch: 'dienstregelpunten') it has to pass or did pass already. With this information and the map of all tracks, track connections and possible service control points (see Figure 6.2) we can calculate the distance from the current location of the train to the bridge was selected.

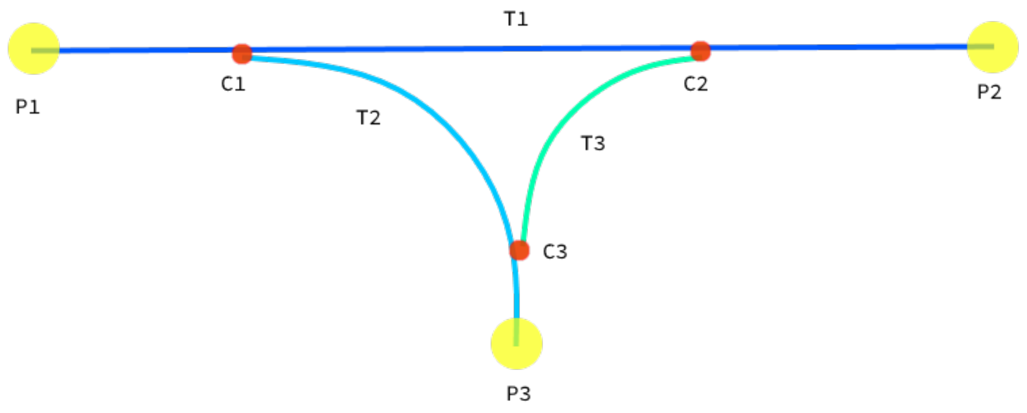


FIGURE 6.2: A typical junction between two tracks, the different tracks (prefixed with T) are connected with the connections (prefixed with C). On the tracks lay numerous service control points (prefixed with P).

In most cases finding the position from one track to a subscribed bridge is quite easy:

- Have start position on some track.
- For each unused connection on the track, check if connected track contains some service control points the train must pass.
- If so, calculate distance from current position to connection and recursively continue calculation until the selected bridge is reached.

To improve efficiency the total map of track is first filtered to tracks which contain service control points which the train must pass.

However, there are some exceptions. For one, in the track map we have there are some tracks that overlap (whole tracks or only beginnings or endings) and having various connections on the overlapping parts. This, in combination with the various service control points only occurring once, and thus only on a single track, makes it unpredictable which track is the shortest (see Figure 6.3). The solution for this was to check for all possible connections and later on make a decision on which route is the shortest.

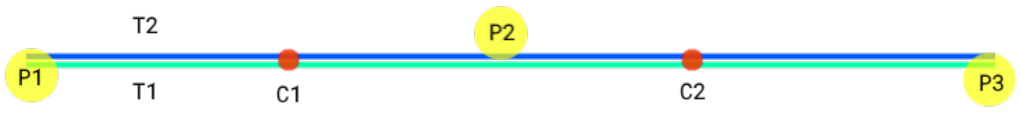


FIGURE 6.3: Figure of two overlaps track T1 and T2. All points only occur on one of two track, P1 and P3 on T1 and P2 on T2. The most efficient way is very obvious: start on T1, pass P1, switch to T2, pass P2, switch to T1 and pass P3. But since other correct routes (T1 -> C2 -> P2 -> C1 -> P3) are also possible, all possibilities have to be checked.

Another exception is where some tracks don't contain any service control points but have to be passed to go from one track to another Figure 6.4. To solve this issue there is an exception if after a track no feasible connection has been found, the program then tries to look for connections to all other track and checks if these tracks have a connection to another usable track.

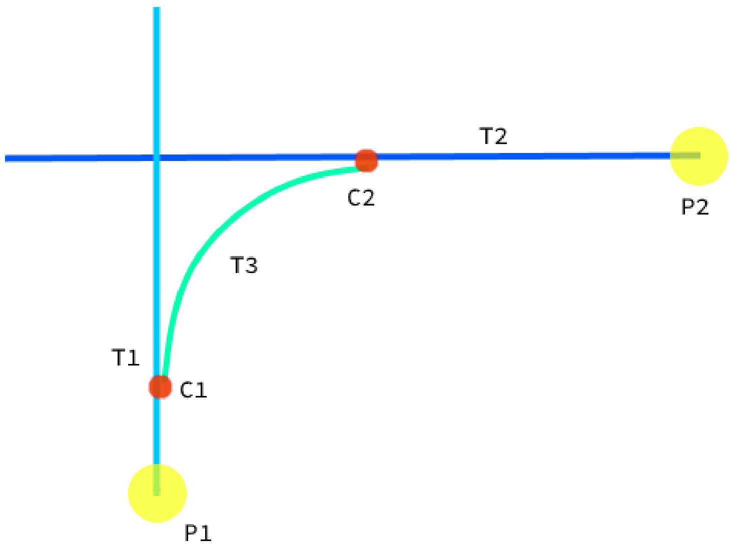


FIGURE 6.4: Traveling from P1 to P2 requires passing T3, which has no service control points on it, and thus is by default not included in the relevant parts of the map.

With the trains passing different points the difference of the distance in combination with the difference in the times of the messages the average speed can be calculated. Then this average speed and the distance to the bridge is sent to the Burgkijker client.

6.3 Brugkijker

6.3.1 TypeScript

Because this application needs to run in a browser it has to be in JavaScript. We implemented it using a superset of JavaScript called TypeScript. TypeScript expands on JavaScript with a number of features and transcompiles to JavaScript. The most notable feature is types: All variables and function are annotated with type definitions. The compiler performs static type checking before transcompilation using these types, helping the developer catch mistakes early on, where before these had to be caught using debugging tools. When plugging this type checking functionality into an IDE, it improves autocompletion and displays type errors directly when writing code. By using TypeScript, some of the advantages of statically typed programming languages can be brought to JavaScript development. However, when using JavaScript libraries not written in TypeScript, which are the vast majority, you are dependent on the community for maintaining type definitions of the library. Also, JavaScript is still a very dynamic language and a lot of libraries use these dynamic properties to such an extent that they are impossible to provide proper type definitions for. In our project, using TypeScript has been a tradeoff between the time saved by the type checker and the time spent finding workarounds for these disadvantages. In the end, there are still a few areas in our code where TypeScript can't properly catch type mistakes.

6.3.2 Node.js

To run the application on the server we use Node.js, a JavaScript run-time environment. The only difference between running the application in the browser and in Node.js is that we need to use different implementations of WebSockets for the data connection and HTTP requests for node replication. Because these things are provided by the browser on the client side, we can't use these in Node.js. Node.js provides its own native HTTP client and we use the popular and well-tested ws library for WebSockets. By using TypeScript interfaces we keep the difference in self-maintained code for both platforms to two very minimal wrappers around these implementations.

We also use Node.js for development tools like the npm package manager and the TypesScript compiler, as well as things like WebPack for making an optimized build of the application to serve to the browser and as a testing platform.

6.3.3 Redux

Redux is an architectural design pattern and accompanying JavaScript library. It is based on a few core concepts:

- One central application state implemented as an immutable data structure.
- Central dispatch of actions.

- One reducer function that is called when an action is dispatched. Takes the state and an action and returns a new state. This function has to be pure (e.g. produce no side effects and always returns the same output with the same input).

UI components can subscribe to state changes and update accordingly. An immutable central state that acts as a single source of truth can have multiple advantages: because data flows only down, it is a very clear and predictable architecture. It also offers easy debugging with time travel, for example: a standard debugger is available for Redux, in which you can see a list of dispatched actions and the changes that they caused in the state. With one click, you revert your entire application to exactly how it was after an action was dispatched. For our system, it makes node replication as described in subsection 6.1.3 very easy. A schematic overview of Redux in our application can be seen in Figure 6.5.

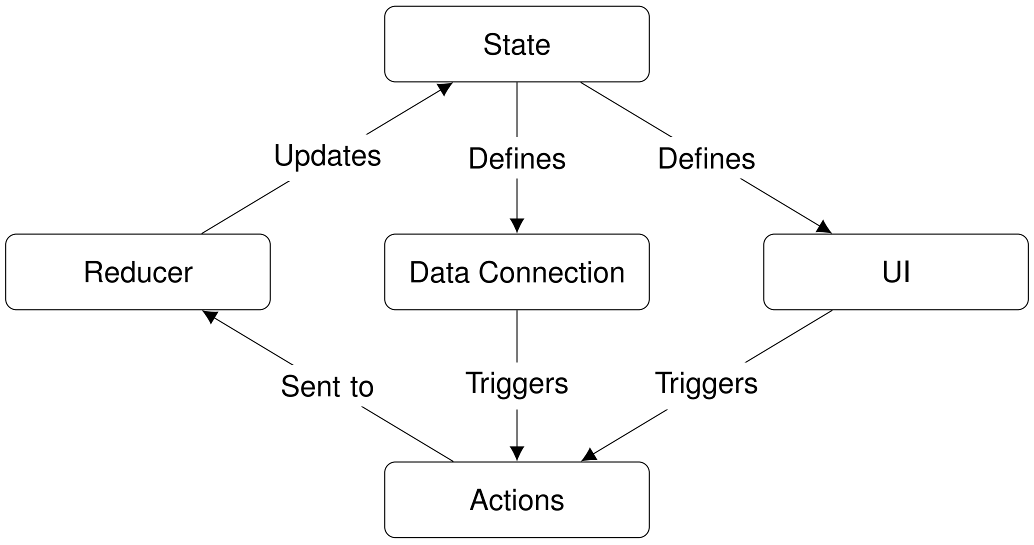


FIGURE 6.5: Overview of core components within the Redux architecture of the Brugkijker application. Note the addition of the Data Connection, it listens to the state to maintain subscriptions on the selected bridges and dispatches the train data coming in as actions.

We will now discuss each part in more detail:

State

Schematically, the application state is structured as shown in Figure 6.6.

In reality, times are implemented as a special object with Moment.js and maps are implemented as immutable data structures with Immutable.js. The list of trains are stored as immutable sorted maps based on B-trees to keep them sorted on arrival time. Redux states should be flat structures and though it might seem that the trains being nested underneath the bridges breaks this rule, the amount of bridges in the Netherlands is fairly static and thus the list of bridges is fixed. So in essence this structure is flat. Adding a bridge warrants an application update (which can be achieved with zero downtime of course).

```
{
  time: "2017-05-25T12:16:38.000Z",
  version: 1,
  lastUpdated: "2017-05-25T12:16:34.285Z",
  trainsByBridge: {
    Spbr: {
      2644: {
        id: "2644",
        startTime: "2017-05-25T12:18:26.359Z",
        endTime: "2017-05-25T12:18:36.359Z"
      },
      1844: {
        id: "1844",
        startTime: "2017-05-25T12:22:49.285Z",
        endTime: "2017-05-25T12:22:59.285Z"
      },
      ....,
      ....
    },
    ....,
    ....,
    Vtbr: {
      3051: {
        id: "3051",
        startTime: "2017-05-25T12:17:22.322Z",
        endTime: "2017-05-25T12:17:32.322Z"
      }
    }
  },
  selectedBridges: [Spbr, Vtbr, Hrm]
}
```

FIGURE 6.6: Datamodel of the state

Actions & Reducer

An overview of actions can be seen in figure Table 6.1. The reducer is straightforward and does nothing more than create a new state based on the actions described there.

Data Connection

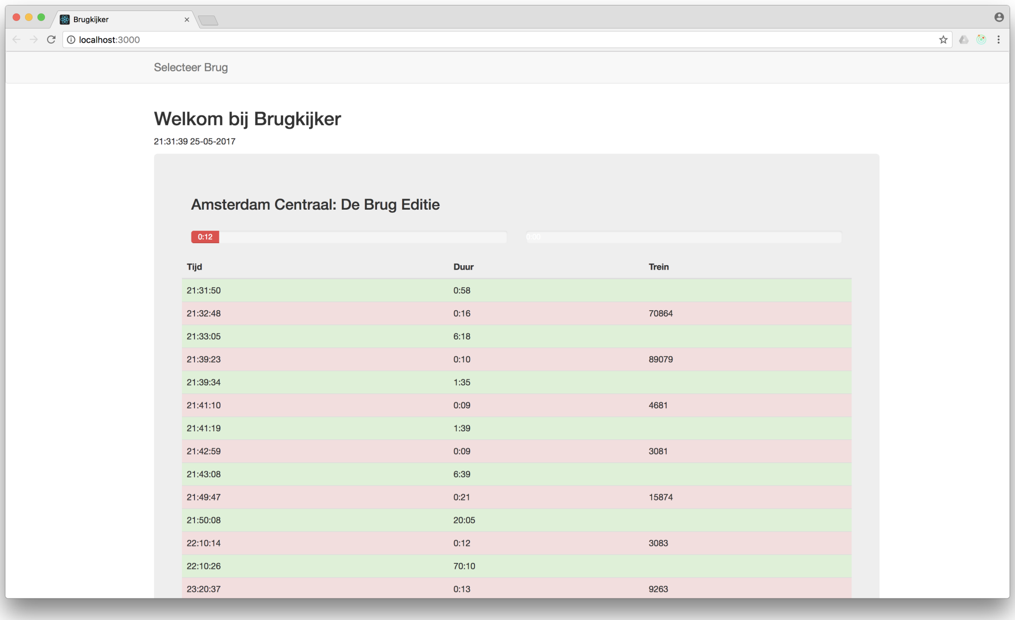
The Data Connection in the diagram is implemented as a class that incorporates a WebSocket connection and an HTTP client. It is responsible for state synchronization over HTTP on startup (e.g. node replication) and whenever the socket connection is broken. After synchronization it automatically opens the socket (again). It listens to state changes and subscribes to data over the socket according to the selectedBridges list. It supplies data received over the socket to the reducer as actions.

SET_TIME	Called every second to update the time. Can also be called manually for testing purposes.
SET_TRAIN	Called by the data connection to update a train's expected arrival/departure time.
TRAIN_PASSED	To delete a train from the state. It is called when a signal is received through the data connection, confirming the train has passed.
SELECT_BRIDGE	To add a bridge to the list of selected bridges.
SET_BRIDGE	To update the entire train list of a bridge. This is used during node replication and status syncing after a connection has been lost.

TABLE 6.1: Overview of all actions used by Brugkijker

UI

The UI displays the state as a simple overview for the user. It shows two decreasing timer bars of which only one is active at any given time, the red one counts down to the moment the train passes whenever a train is on the bridge and the green one counts down to the arrival of the next train whenever there are no trains on the bridge. Underneath these timers a list is displayed of all incoming trains and the gaps in between along with the duration of the event and a colour code that matches the corresponding timer. See Figure 6.7. When the user clicks 'Select Bridge' a list is displayed and additional bridges can be added (this is done by dispatching an action once clicked). If multiple bridges are selected, multiple panels including timers and list are displayed simultaneously, one for each bridge. For more details about the implementation of the UI, see the next section.



Welkom bij Brugkijker

21:31:39 25-05-2017

Amsterdam Centraal: De Brug Editie

Tijd	Duur	Trein
21:31:50	0:58	
21:32:48	0:16	70864
21:33:05	6:18	
21:39:23	0:10	89079
21:39:34	1:35	
21:41:10	0:09	4681
21:41:19	1:39	
21:42:59	0:09	3081
21:43:08	6:39	
21:49:47	0:21	15874
21:50:08	20:05	
22:10:14	0:12	3083
22:10:26	70:10	
23:20:37	0:13	9263

FIGURE 6.7: A screenshot of the Brugkijker application running in a browser.

6.3.4 React

React is a JavaScript library for building user interfaces. Redux was originally designed for use with React and is a very good match. React user interfaces consist of nested components that only update when their properties change. By connecting a React component's properties to only the part of the application state that affects it, unnecessary updates and renders are prevented. Memoized selectors can provide another layer in between, in case the component should only update if a derived value calculated from the state changes. Selectors subscribe to parts of the state that affect them just like components can do directly and update their calculated value every time these parts change. Selectors can also subscribe to each other[1]. React components themselves provide another level of optimization: they maintain a virtual DOM tree and only update the actual DOM tree of the browser where there are changes. This reduces drawing time, an example of this layered approach is given in Figure 6.8.

6.3.5 Node Replication

To ensure synchronized state across the different nodes, we devised the following protocol. When a node initializes a selected bridge, it first subscribes to that bridge over the WebSocket. Action dispatch is withheld until after the next step. The node will then make an HTTP request to a node running on the swarm. Nodes on the swarm listen on an HTTP endpoint and serve their state on request. Once the new node receives that state, the withheld actions created will be dispatched on the received state and the new node is now in sync with the other nodes. This is true because of the following reasons:

- Because WebSockets run over connection-oriented TCP, all data received by the data server after the connection is established is guaranteed to reach the node at some point as long as the connection is not broken. If the connection is broken, the whole process is repeated again.
- Because a WebSocket connection is established first, the delay will cause the received state to contain no actions that are not already received over the WebSocket in the very vast majority of the time. In the rare case a delayed packet to the node being replicated causes the lastUpdated timestamp in the received state to be lower than the timestamp of the first action received over the socket, the received state is discarded and the process is repeated again.
- The node being replicated from can never be out of sync itself, because all nodes in the swarm will only open their HTTP endpoint once they completed the synchronization process and close them if their WebSocket connection is ever broken. Also the lastUpdated timestamp safeguards against this.

To keep the nodes synchronized during an incremental update an additions to the protocol is used:

[1]Selectors don't actually subscribe to state changes, components subscribe to state changes and call on selectors that will then either update or provide their memoized value based on equality checks. This way, selectors won't recalculate if no components are using them. For our discussion however, it is easier to think of it as subscribing.

- Every version a new port will be used for the state endpoint. This will make sure a node will only try to retrieve a state from a node of the same version, e.g. one that is already updated.

In both the case of initializing the swarm for the first time and updating the nodes, there is the problem of the first node being initialized/updated not being able to make a connection to an HTTP endpoint for node replication. Because of this, if there is no endpoint listening it will assume it is the first node and will start dispatching actions on an empty state, or a state of an old version in the case of an update. This is somewhat problematic, see chapter 10.

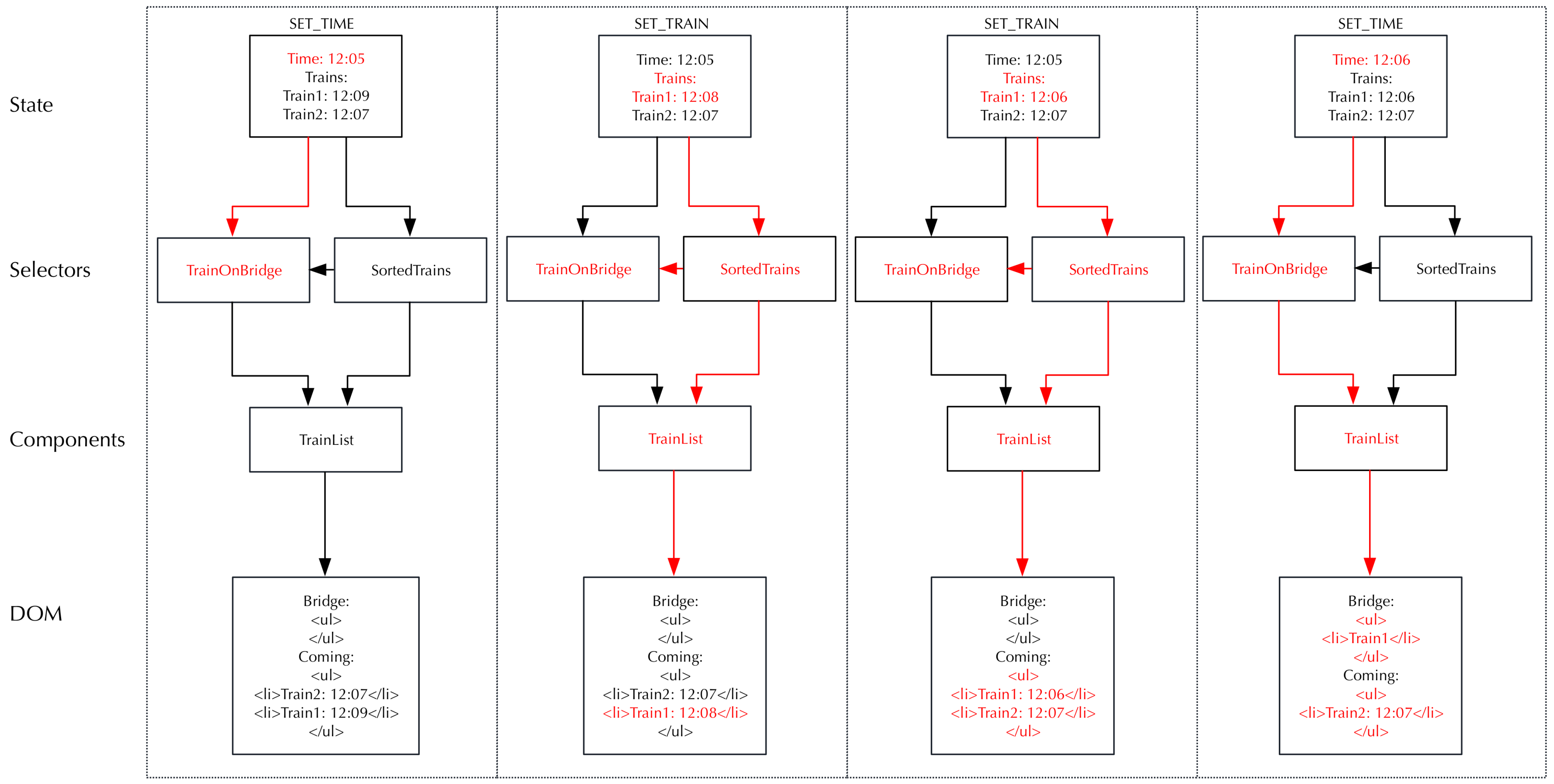


FIGURE 6.8: Here we can see the effect of four consecutive actions on the state, the selectors, the components and the DOM tree. This represents our application in an earlier state of production, where we didn't use an immutable B-tree yet for the trains and we had to resort every time information about a train changed. Red text signifies the parts of the application that are updated through the 'update path' marked by red arrows. After the first action is dispatched, the time does not equal the previous time and so the TrainOnBridge selector recalculates based on the new time and the list of sorted trains from the SortedTrains selector, that is still memoized because the trains haven't changed. (Note that, although state is immutable and a new state object is created every dispatch, this is done through shallow copy, meaning the new state can reference parts of the old state as long as nothing in those parts has changed. This makes copying faster and equality checks possible so SortedTrains knows it won't have to recalculate.) The result of the selector is still *false*, so the TrainList component won't update because of another equality check. Next, a change in Trains will cause SortedTrains to resort. This will cause the TrainList component to update (and TrainOnBridge). The React library will then only update the train entry in the list that has changed. Another change in Trains will cause the order to differ. Now, at the end of the update path, the entire list will update. Last, another change in time will cause TrainOnBridge to update again and this time its value will change. The component updates to reflect Train1's arrival at the bridge.

Chapter 7

Process

7.1 Scrum

During our project we did not use Scrum, this is because Scrum assumes sprints of about a week, and since we only had a few weeks to finish our development process we figured it wouldn't be worth the effort. Another benefit of using Scrum is that it is easier to deal with changing requirements, but since our client's main goal was not the Brugkijker application, the requirements of this application could not really change due to the client.

7.2 Git

For this project we used Git as a version control system, mainly because we all had experience with this system so it would be easy to set it up and start right away. We used feature branches to commit to and tried to keep commits as small as possible and with clear commit messages. Since we all had experience with Git this did not cause any problems.

We used a private GitHub repository as host for sharing our repository. On GitHub we restricted access to the master branch to make sure all changes are made with pull requests. Before pull request could be merged, at least one person had to approve the changes. Because we didn't have a lot of time for the actual implementation we were not able to setup a decent continuous integration platform.

7.3 SIG review

For our first SIG review we did not deliver a lot of code since we had less than two weeks for this review. However, we did supply some code and we have gotten a review of that. SIG rated our code four out of five stars. The only point of improvement being 'unit complexity'. We didn't score a maximum for unit complexity because the replay server had some methods which were considered too complex according by SIG. We fixed most of the specific issues SIG mentioned about complexity. However, we did not agree about some issues and splitting up the method would decrease readability so we kept those few methods intact.

Furthermore, SIG mentioned that the presence of some testing code was promising, but we should extend the amount of test code. We also had to keep up writing testing code when new features were developed or the functionality of methods would change.

Chapter 8

Discussion

This chapter discusses our experiences and the issues we encountered during our project.

8.1 Process issues

8.1.1 Start

We started on the first day of the project with a meeting at ProRail, in this meeting some explanation was given of the application we had to develop (VIEW) and a day later we received some documentation of the program ProRail had already build before. However, this documentation was hard to grasp and after some email contact we still did not understand exactly what the application should be doing and what our goal was.

During the second week it was still unclear what the exact subject of the project was. The general project description gave a brief explanation of that the application had to be highly available and we received some explanation of the program, but no technical requirements were provided. This made it that we effectively could not perform any work until our second meeting with ProRail, unfortunately the earliest possible date for this was – due to incompatible agendas – at the end of week 3.

During this meeting in week 3 we met with an internship mentor at ProRail who was able to meet with us more often and thus we planned weekly meetings at ProRail. Some days later we received some XML files we could work with and we finally could set up some basics of the program. Furthermore, we discussed what programming languages and what frameworks we were going to use.

8.1.2 Program switch

During the planned meeting with ProRail in mid week 4 we met with the process manager of the VIEW application. He knew what the main requirement of the project was, high availability, and wondered why we were working with VIEW. A much simpler application was used in the IT structure of ProRail and keeping in mind that we were almost halfway the project we decided to switch to the simpler application: Brugkijker. A few days later we received some documentation on the system so we could start reading into it.

Unfortunately, we still had to wait until the start of week 6 before we received the input logs without documentation. From this moment on we could actually start writing the application, with the first (postponed) SIG deadline only ten days later.

8.1.3 Overall communication

The amount of management layers within ProRail made that requests to get information or data from sources which needed approval from upper layers was quite difficult. Sometimes we had to wait over a week to get an answer or receive data, we sometimes didn't get an answer at all.

8.1.4 Collaboration

Because we did not use Scrum in our project it was not always clear what issues needed to be covered and what people were working on. Even though we could probably only have two or three sprints, we probably would have benefited from a decent overview of the issues and a clearer view on what people were working on.

Since the group size for this project was relatively small the overall collaboration was fine, we met nearly every day during this project and this ensured that a short meeting was almost always possible on short notice. Questions or feedback about a pull request or someones work was, because we worked in the same room most of the time, easy to do.

8.1.5 Continuous integration / testing

For this project we tried to use unit testing, but we did not have enough time to write a decent amount of unit tests. This does mean that we do not know if our product really works exactly as expected. We however did do a lot of non-automated end-to-end tests, so we have at least some certainty that the product works.

At the beginning of the project we also discussed to have some sort of continuous integration tools, like automated testing tools for pull requests and automatic container building tools when the master branch is pushed to. However, we also didn't have the time to properly set this up.

8.2 Design issues

8.2.1 High availability

None of us had any experience of using container applications and thus we had to learn how this worked from the very start. When we finally had some know-how with Docker we found out that sharing the application images in a swarm wasn't as easy as it looked at a first glimpse. We experienced some networking issues, some of these because we couldn't get the swarm running correctly on the TU Delft network and some because we simply didn't have enough skills with networking in containers and VMs (which we, at first, tried to use for testing local swarms on a single computer).

8.2.2 Brugkijker

As can be read on the previous section we had to wait a lot on data or other resources, because of this we sometimes had to work just with what we had. This caused that instead of asking for the correct data we just designed some kind of workaround which wasn't always a good solution.

8.3 Ethical issues

In our project we did not encounter many real ethical issues. A big part of the economy is dependent on railways and shipping though. Not taking proper care to make our application work and be available can cause unnecessary delays to happen when bridges are opened or closed at the wrong time. This can cause financial loss and other problems for businesses and individuals.

Chapter 9

Conclusion

According to ProRail our main goal was to make an application which is highly available, meaning the application won't experience downtime on updates or unexpected server shutdowns. A secondary goal was to make an application called Brugkijker which shows bridge operators how long it takes until a train arrives at a certain bridge, and for how long this train will be on the bridge. For this conclusion we will discuss both goals separately.

9.1 High availability

We chose to work with Docker for this project, Docker is a container application which can also do container orchestration with Docker Swarm. We found that with this application it is quite easy to set up a swarm with multiple machines and to get an application running in this program. Most time we spent with Docker was finding out the very basics of containerization and sharing built application images. We had to spend some time with networking issues as well, which normally shouldn't cause a lot of problems.

In week 9 of the project we had given a demonstration at ProRail for the IT management which originally came up with this project and they seemed happy with the results. We were able to show that random nodes could shut down at any given time while the application kept being reachable. We also showed that with this same setup, it is possible to do a rolling upgrade of an application without requiring any downtime.

9.2 Brugkijker

For the Brugkijker application itself we encountered more problems. At first we had some trouble getting the right resources and data needed to implement this application, this caused that we only had 4 weeks left to actually develop the application. This turned out to be enough time to get the basics of the application up and running but definitely not all 'Could have' requirements have been implemented.

Chapter 10

Recommendations

Brugkijker needs to make an estimation of when trains arrive at the bridge based on the data it receives. This includes speed and distance to the bridge. In its current form, the application just makes a simple calculation based on these values, without accounting for acceleration. This causes the estimated arrival time to wildly fluctuate. A more sensible prediction should be made using some form of polynomial extrapolation, also making use of the train's maximum speed.

The protocol described in subsection 6.3.5 that we use for state synchronization across nodes still has a problem. It assumes that whenever there is no endpoint listening, there is no state to synchronize with. This might not actually be true in all cases. The solution would be to have a special type of container of which only one is deployed on the swarm, that does exactly the same as the other Brugkijker nodes, except that it is the only one that can make this assumption. This container should then be deployed first and guides the initialization or update.

Also, the node replication protocol is very specific for this type of application where data flows only down, but is still dependent on previously received data. It aims to provide a way to keep the application as real-time as possible, being notified of data changes directly, while still preserving synchronization. We also use it because it allowed us to reuse a lot of our code on the server side. We had already implemented the client-side and needed state synchronization fast. It has the downside of not being thoroughly tested and for a lot of different kind of applications more well tested solutions like distributed databases would be preferable. However, it is an interesting method and more research and testing could be worthwhile.

Bibliography

Apache (2017a). *Apache Core Features*. URL: https://httpd.apache.org/docs/2.2/mod/core.html#keepalivetimeout (visited on 06/25/2017).

— (2017b). *Apache Core Features*. URL: https://httpd.apache.org/docs/2.4/mod/core.html#keepalivetimeout (visited on 06/25/2017).

Binder, Andrej, Tomas Boros, and Ivan Kotuliak (2015). "A SDN Based Method of TCP Connection Handover". In: *Information and Communication Technology*. Springer, pp. 13–19.

Brewer, Eric (2012). "CAP twelve years later: How the" rules" have changed". In: *Computer* 45.2, pp. 23–29.

Cai, J and Simon Leung (2002). "Building highly available database servers using Oracle real application clusters". In: *Redwood Shores, CA: Oracle Corporation*.

Chang, Victor (2015). "Towards a Big Data system disaster recovery in a Private Cloud". In: *Ad Hoc Networks* 35, pp. 65–82.

Choi, Alan (2009). "Online application upgrade using edition-based redefinition". In: *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*. ACM, p. 4.

Cukier, Daniel (2013). "DevOps patterns to scale web applications using cloud services". In: *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*. ACM, pp. 143–152.

Curry, Edward (2004). "Message-oriented middleware". In: *Middleware for communications*, pp. 1–28.

Day, John D and Hubert Zimmermann (1983). "The OSI reference model". In: *Proceedings of the IEEE* 71.12, pp. 1334–1340.

Ebook: Docker for the Virtualization Admin (2017). URL: https://goto.docker.com/docker-for-the-virtualization-admin.html (visited on 06/25/2017).

Fall, Kevin R and W Richard Stevens (2011). *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley.

Fette, Ian (2011). "The websocket protocol". In:

Fielding, Roy et al. (1999). *Hypertext transfer protocol–HTTP/1.1*. Tech. rep.

Gropp, William et al. (1996). "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6, pp. 789–828.

justin (2015). *Stateless WebSockets with Express and Pushpin*. URL: http://blog.fanout.io/2015/03/09/stateless-websockets-with-express-and-pushpin/ (visited on 06/25/2017).

Kopparapu, Chandra (2002). *Load balancing servers, firewalls, and caches*. John Wiley & Sons.

Microsoft (2017). *Default Limits for Web Sites <limits>*. URL: https://www.iis.net/configreference/system.applicationhost/sites/sitedefaults/limits (visited on 06/25/2017).

Piedad, Floyd and Michael Hawkins (2001). *High availability: design, techniques, and processes*. Prentice Hall Professional.

Pimentel, Victoria and Bradford G Nickerson (2012). "Communicating and displaying real-time data with WebSocket". In: *IEEE Internet Computing* 16.4, pp. 45–53.

Postel, Jon (1980). *User datagram protocol*. Tech. rep.

— (1981). "Transmission control protocol". In:

— (1994). "Domain name system structure and delegation". In:

Satyanarayanan, Mahadev (1993). "Distributed file systems". In: *Distributed Systems. Addison-Wesley and ACM Press*, 821, pp. 145–154.

Singh, Kundan and Henning Schulzrinne (2007). "Failover, load sharing and server architecture in SIP telephony". In: *Computer Communications* 30.5, pp. 927–942.

Tudorica, Bogdan George and Cristian Bucur (2011). "A comparison between several NoSQL databases with comments and notes". In: *Roedunet International Conference (RoEduNet), 2011 10th*. IEEE, pp. 1–5.

Zimmermann, Hubert (1980). "OSI reference model–The ISO model of architecture for open systems interconnection". In: *IEEE Transactions on communications* 28.4, pp. 425–432.

Appendix A

Project description

Als beheerorganisatie moeten wij regelmatig updates(patches) uitvoeren om onze infrastructuur up2date te houden. Wat wij nu zien is dat niet applicaties tegen eenvoudige schakelmomenten kunnen en voor de gebruiker last oplevert. We willen in een POC op basis van een bestaande applicatie laten zien dat met moderne technieken schakelmomenten geen impact hebben voor de gebruiker. En dat applicaties ook snel eenvoudig e bouwen zijn.

Appendix B

SIG Review

De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Complexity.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt.

In jullie project is Replay.process_cli_input een goede kandidaat om nog wat verder te vereenvoudigen. Er zijn verschillende mogelijkheden om dit te bereiken. De makkelijkste is om het parsen van de parameters uit te splitsen naar een aparte methode. Met andere woorden: je scheidt de inhoud van de for-loop van de for-loop zelf. Een andere manier is om de detectie van de "speed" parameter naar een aparte methode te verplaatsen. Jullie kijken nu of de snelheid -2, -4, of -8 is, maar deze aanpak wordt moeilijk leesbaar als je straks 5+ verschillende snelheden ondersteunt.

De constructor van Replay bevat ook complexiteit, maar dan op een iets andere manier. Jullie doen nu steeds hetzelfde, eerst kijken of een parameter beschikbaar is, en zo ja, die toekennen aan een veld. Je zou dit ook anders kunnen opschrijven om zo de hoeveelheid herhaalde code te beperken.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt. Op dit moment is de hoeveelheid tests nog vrij beperkt in verhouding tot de hoeveelheid productiecode.

Over het algemeen scoort de code dus bovengemiddeld, dus bovenstaande aanbevelingen zijn voornamelijk kleine puntjes om een nog hogere score te bereiken tijdens de rest van de ontwikkelfase.

Infosheet

General Information

Title of the project: POC High-available Application
Name of the client organization: ProRail
Date of the final presentation: 3 July 2017

Description

ProRail controls train services, performs the maintenance of the railway infrastructure and is responsible for safety and security. Information Technology (IT) has an important role in performing these tasks. Dedicated software applications form the IT infrastructure of ProRail. Most of these applications should be highly available, during downtime of critical systems no single train can drive on the Dutch railway network. Therefore, it is hard to update these applications.

Challenge: The client wants to be able to update their critical systems without downtime. We had to explore solutions for high availability and create a proof of concept, highly available version of one of their applications.

Research: We researched different solutions and subjects regarding high availability including redundancy, incremental updates and load balancing.

Process: Because of problems during the start of the project, we had to develop in significantly less time. Because of this, we did not have time to implement standardized process methodologies.

Product: We developed a version of ProRail's Brugkijker application that can be deployed on a Docker Swarm. The application shows incoming trains for a selected bridge so time-frames during which the bridge can be opened can be checked for. Updates to the application can be incrementally rolled out without losing data and without downtime.

Outlook: The methods used for the Brugkijker product can be used for bigger, more critical systems. We have demonstrated the possibilities, which was our assignment. Because the Brugkijker is in an outer layer of the client's infrastructure, although developed in isolation, it can also be relatively easy deployed to replace the current version for real.

Team

Marco Boom
Interests: Web application programming, high availability platform design
Contributions: Web server architecture, client-server communication protocol

Onne van Dijk
Interests: Software architecture, programming language design
Contributions: Client side/Node.js development

Max Groenenboom
Interests: (Software) efficiency, client-side programming
Contributions: Client side programming, LaTeX structure & layout

Joost Wooning
Interests: Software architecture, server-side programming
Contributions: Replay server, server-side calculations

All members contributed to the final report and presentation.

Client

Coert Busio
ProRail

Coach

Georgios Gousios
Web Information Systems

Contact: onnevandijk@icloud.com
The final report for this project can be found at: `http://repository.tudelft.nl`