Final report: Dynamic peer-to-peer game networks using WebRTC

Jasper Abbink
Karrens Grigorjancs
Joost Verdoorn

30th August 2013
Abstract

This document is the final report on the Bachelor Project conducted by Jasper Abbink, Karens Grigorjancs and Joost Verdoorn on Dynamic peer-to-peer game networks using WebRTC. In this report we detail our findings on the recently developed WebRTC technology. WebRTC enables the creation of web applications built around peer-to-peer technologies by providing the means to directly connect one browser to another. With this project we aimed to facilitate web developers by developing a software library that is a drop-in solution for large-scale peer-to-peer networks. We investigated the scalability of WebRTC networks and attempted to seek the edges of the technology. We evaluated a number of ways by which a browser-based peer-to-peer networks can be deployed, and implemented the ones that best suited our needs. To demonstrate our library we created a small massively-multiplayer arcade game as an entertaining way to display WebRTC’s capabilities.
## Contents

1 Introduction .................................................. 6

2 Assignment .................................................... 7
   2.1 Client .................................................... 7
   2.2 Problem description ...................................... 7
   2.3 Goal ........................................................ 7
      2.3.1 Philosophy ........................................... 7
   2.4 Assignment formulation ................................... 8
   2.5 Deliverables ............................................... 8
   2.6 Risks ....................................................... 9

3 Process ......................................................... 10
   3.1 Methodology ............................................... 10
      3.1.1 Scrum ................................................ 10
      3.1.2 Behaviour-Driven Development ...................... 10
   3.2 Development tools ........................................ 11
      3.2.1 Git .................................................. 11
      3.2.2 Cake ............................................... 11
   3.3 Planning Tools ............................................ 12
      3.3.1 GitHub issues ....................................... 12
      3.3.2 HUbboard ........................................... 12

4 Background information .................................... 14
   4.1 WebRTC ................................................... 14
      4.1.1 Introduction .......................................... 14
      4.1.2 The API ............................................. 14
      4.1.3 The protocols ....................................... 14
      4.1.4 Bootstrapping ...................................... 15
      4.1.5 Streams and channels ............................... 16
   4.2 Peer-to-Peer .............................................. 16
      4.2.1 Introduction ........................................ 16
      4.2.2 Node heterogeneity .................................. 17
      4.2.3 Structuring the network ............................ 18
      4.2.4 Supernode topologies and election approaches .... 18
   4.3 MMOG ...................................................... 19
      4.3.1 Dead Reckoning ..................................... 19

5 Research ....................................................... 21
   5.1 WebRTC ................................................... 21
      5.1.1 API differences ..................................... 21
      5.1.2 Bootstrapping ...................................... 21
      5.1.3 Data channels ...................................... 22
   5.2 Peer-to-Peer .............................................. 22
      5.2.1 Structuring the network ............................ 22
5.2.2 Group based network systems .................................. 23
5.2.3 Location based systems ........................................... 23
5.3 MMOG .................................................................. 24
  5.3.1 Network utilisation ................................................... 25
  5.3.2 Graphics Engine ...................................................... 25

6 Design ..................................................................... 26
  6.1 Node Communication .................................................. 26
    6.1.1 Server technology ................................................... 26
    6.1.2 Mixins ................................................................ 26
    6.1.3 Event-Driven Architecture ....................................... 26
    6.1.4 Controllers ............................................................ 27
    6.1.5 Remotes ................................................................. 27
    6.1.6 Messages ............................................................... 27
    6.1.7 Initialisation ............................................................ 28
  6.2 Network Topology ....................................................... 28
    6.2.1 Server communication ............................................. 29
    6.2.2 Vivaldi ................................................................ 29
    6.2.3 PoPCorn ................................................................. 29
    6.2.4 Quirks ................................................................. 29
    6.2.5 Illustration ............................................................ 30
  6.3 Game .................................................................... 30

7 Implementation details ................................................... 35
  7.1 Technology ............................................................... 35
    7.1.1 WebSockets .......................................................... 35
    7.1.2 WebGL ............................................................... 35
    7.1.3 Canvas ............................................................... 35
    7.1.4 Application Cache ................................................ 35
  7.2 Frameworks .............................................................. 35
    7.2.1 Node.js ............................................................... 36
    7.2.2 RequireJS ............................................................. 36
    7.2.3 Socket.IO ............................................................. 36
    7.2.4 Underscore .......................................................... 36
    7.2.5 Three.js ............................................................. 36
    7.2.6 KineticJS ............................................................. 37
    7.2.7 Adapter.js ............................................................ 37
  7.3 CoffeeScript ............................................................. 37

8 Quality assurance and testing ........................................ 38
  8.1 Frameworks ............................................................. 38
    8.1.1 Jasmine ............................................................. 38
    8.1.2 CoDo ................................................................. 38
  8.2 Tools ................................................................. 38
    8.2.1 Node Inspector ..................................................... 38
    8.2.2 Map ................................................................. 38
8.3 External Tools ......................................................... 38
  8.3.1 JS Coverage .................................................. 38
  8.3.2 cake ............................................................. 39
  8.3.3 CoffeeLint ...................................................... 39
  8.3.4 Sublime .......................................................... 39
8.4 SIG ................................................................. 40

9 Conclusions .......................................................... 41
  9.1 How can one distribute essential network tasks over all peers? ... 41
  9.2 How can the impact of unpredictable circumstances be reduced? 41
  9.3 How can all data reach all peers in a bandwidth-efficient way? ... 42
    9.3.1 Coordinate system ........................................... 42
    9.3.2 Routing ...................................................... 42
    9.3.3 WebRTC ...................................................... 42
    9.3.4 Speed ........................................................ 42

10 Discussion ........................................................... 43
  10.1 Network topology ................................................. 43
    10.1.1 Supernode overlay network ................................ 43
    10.1.2 Routing tables .............................................. 43
    10.1.3 Comparing network topologies ................................ 43
  10.2 Network security ................................................ 43
    10.2.1 Overloading the network or server ......................... 44
    10.2.2 Game integrity .............................................. 44
    10.2.3 Game security .............................................. 44
  10.3 Cross Platform .................................................. 45
  10.4 Peer discovery ................................................... 45
  10.5 WebRTC impact ................................................... 46

11 Evaluation ............................................................. 48
  11.1 Technical challenges ............................................. 48
    11.1.1 Unit Testing ............................................... 48
    11.1.2 Game Testing ............................................... 48
  11.2 Process challenges .............................................. 49
    11.2.1 Scrum ....................................................... 49
    11.2.2 Behaviour driven development .............................. 49

A Glossary ............................................................... 50

B Bibliography .......................................................... 52

C Software Improvement Group Review 1 .................................. 54

D Software Improvement Group Review 2 .................................. 56
1 Introduction

This document is the final rapport of the bachelor end project which is performed under course TI3800 of Delft University of Technology that took place in the summer of 2013 at TNO. The goal of the project is to perform a software product development track in a real company. Over the course of the project we gathered knowledge on WebRTC, node topology and online games. We used that knowledge to do research on these subjects. After the research we delivered a library for WebRTC communication, a structured network topology overlay library and a demonstration in with a browser-based arcade game. This document describes the project form the very beginning by formulating the assignment and ends with our conclusions of the project and discussion on the subject.

We are very proud of the results of this project and are very glad we got the possibility to work on it at TNO this summer. We want to thank all people who contributed to the project:

- **Victor Klos** for being our coach and motivating us to get more out of the project.
- **Eelco Cramer** for sharing his expertise with us and giving an excellent technical feedback.
- **Harrie van de Vlag** for managing us as a group and helping us with Scrum in hard times.
- **Jan Hidders**, our mentor from the university.
- **Felienne Hermans**, our bachelor project coordinator.

Jasper Abbink, Karens Grigorjancs, Joost Verdoorn
2 Assignment

This chapter will start with a short introduction to our client. This is followed by a description of our problem. After that the project’s goal and philosophy about why we chose this subject is explained. Fourth we specify the main research question and explain why we picked it. After that the pre-determined deliverables are discussed and last but not least we compare the previously determined risks to the actual problems we encountered with the planning of this project.

Most of this information can also be found in the original plan of action.

2.1 Client

The client and supervisor of this BSc project is the same organization: TNO. TNO is a non-profit research organization that focuses on applied sciences. More information about TNO can be found at http://www.tno.nl/.

2.2 Problem description

TNO helps their customers understand, use and often develop advancements in technology. One interesting new technology in the field of Media Networking is WebRTC. Currently being standardised by W3C, it allows for serverless browser-to-browser communication. Some even foresee WebRTC to be pivotal in shaping web communication in the next decade.

Having a thorough understanding of a technology encompasses having a feeling for its boundaries and limitations. To that end, a WebRTC project proposal was written in which students were invited to come up with an interesting use case that in some way explores the boundaries of the technology.

2.3 Goal

There are several subjects to think of when trying to reach this goal. You can think about number of simultaneous users, different media sources for streaming, media synchronization, limitation of bandwidth and cross-browser compatibility. In section 2.4 we will chose the subjects for our assignment.

The second goal is to create a demonstration of our work to easily show others what we have accomplished with WebRTC.

2.3.1 Philosophy

More and more developers turn to the Web as their main platform. It offers a far wider reach than any other platform, and the possibilities it offers increases every day. Aside from that, the last decade has also seen a dramatic increase of bandwidth available to the average consumer. This bandwidth often exceeds the actual use of the average person.

Developers, on the other hand, have to hire expensive servers to serve a very demanding user base, who expect a high quality of service and fast loading times.
Where these server costs can easily be borne by the Googles and Facebooks of this world, it’s a heavy burden on individuals and small businesses, and undoes some of the democratising impact the Internet has had on the world.

In peer-to-peer networks the burden of bandwidth is borne by the individuals that make up the network, and this gives developers the opportunity to create a rich user experience without having the downsides of maintaining expensive servers. With WebRTC, peer-to-peer networks come to the browser, and we want to help developers make the best use of it. We aim to keep server utilisation to a minimum, and to maximise the use of the possibilities WebRTC offers.

Aside from relieving monetary burdens, WebRTC can offer users more privacy by allowing them to interact directly with each other instead of having to trust a third party with their communications. This decentralised nature makes it harder for government institutions to snoop through communications that aren’t intended for their eyes.

2.4 Assignment formulation

After learning about the different aspects of WebRTC, trying out some experiments and encountering our own problems with it, we decided to focus our project on the scalability of WebRTC-networks. We chose this because WebRTC makes it possible to connect multiple users with each other without a central server that relays all traffic. Properly designed a system could then construct a large global peer-to-peer network.

“Distributed networks” is a subject with large amounts of research already done. Most research focuses on specialized clients connecting to such a network. Because WebRTC runs without external software directly in the user’s browser, where computer resources are not easily given to a script requesting it, we thought it would be very interesting to focus on scalability. From here we constructed this main research question: “In what way can one create a dynamic peer-to-peer multiplayer-game-network-topology that scales with the amount of users participating by only using WebRTC?”. Related questions to this main question are:

- How can one distribute essential network tasks over all peers?
- How can the impact of unpredictable circumstances (peers suddenly leaving the network) be reduced?
- How can all data reach all peers in a bandwidth-efficient way?

To cover all these questions we decided to develop a simple Massively Multiplayer Online Game (MMOG) as demonstration of our underlying WebRTC network structure.

2.5 Deliverables

In addition to the deliverables from the University (process documentation, progress-reports, this document) we have two deliverables for TNO.
The first is a JavaScript library that allows anyone to easily create a scalable network with WebRTC. This library will contain all functionality to cover the research questions posed in the previous section.

The second deliverable for TNO is a demonstration that fully uses the created library. We picked a browser-based MMOG which will function as one big integration test for the library.

2.6 Risks

Before we started this project we saw bad time management as the biggest risk. In the original plan of action this was described as possibly not finishing the library or the game. In the end we did make two working parts, but both do not contain all planned features.

Luckily Scrum saved us from our initial fear and made sure that we at least do have two working products now that can easily be extended after the project is over.
3 Process

3.1 Methodology

In this chapter we will explain how we decided to execute this project. We will begin with explaining our software development methodology and after that we will continue with our software development technique.

3.1.1 Scrum

During the study Computer Science, we encountered several software development methodologies. From this we know that agile methodologies like Scrum fit very well in relatively short and small projects with a small team. Scrum uses small development cycles called sprints to stimulate a fast completion of the project. At the beginning of each sprint the team decides which features shall be implemented and at the end of the sprint there has to be a working product. Unimplemented features will remain in the Product Backlog”.

In our team Karen is the Scrum Master. The task of the Scrum Master is to remove all distracting influences for the development team. Our Product Owner is Victor Klos from TNO. He prioritizes our Product Backlog and will ensure our product meets the requirements for TNO. During the holiday of Klos, this task was taken over by HARRIE VAN DE VLAG. EELCO CRAMER from TNO will assist us with technical problems.

The runtime of our project is very limited. Therefore we have sprints of one week, from Thursday to Wednesday. Everyday at 10 o’clock we have a daily standup that takes up 15 minutes at most. In this time we discuss what we did the previous day and what we will do the upcoming day. Every Thursday morning there is a Sprint Planning Meeting to determine which Product Backlog items get included in the upcoming sprint. The Product Owner always tries to attend at least this weekly meeting to properly prioritize the backlog items.

3.1.2 Behaviour-Driven Development

To deliver a fully tested library we decided to use Behaviour-Driven Development (BDD). BDD can be seen as an extension to Test-Driven Development (TDD). The difference is that tests in BDD try to be as close to user-stories as possible. These stories can easily be deduced from the Product Backlog items we use in Scrum. Just like with TDD, tests in BDD are written before the actual unit is being implemented. After writing all (at that moment failing) tests, the developer will try to get test after test to pass by implementing the least amount of code necessary for this.

The tests will not only be used to determine if the software acts as described. Another use is to make sure new code does not break older functionality. As testing framework we use Jasmine, a BDD-test-framework for JavaScript.
3.2 Development tools

3.2.1 Git

As version control system we used git hosted on GitHub with web interface. We decided to use git because of the many advantages above other version control systems like speed and decentralisation. Our project is about distributed systems after all. Git helped us a lot to get more overview of continuously changing files.

Our repository is accessible via https://github.com/joostverdoorn/webrtc/.

3.2.2 Cake

Cake is a tool that executes buildscripts for CoffeeScript. Our so called Cakefile has the following features:

- **cake deploy**

  Automatically downloads the server dependencies from internet using Node Package Manager (NPM), then runs a build command. After this command a game is ready to play.

- **cake build**

  Compiles all *.coffee files and stores them in a the compiled folder. After that it copies all non *.coffee files to the compiled directory as well.

- **cake clean**

  Removes everything but the source files.

- **cake watch**

  Watches the file system and if needed compiles and copies the edited files. This is an continuous process until the user aborts it.

- **cake test**

  Runs all tests for the library and starts a server up with the codecoverage results.

- **cake lint**

  Tests all coffee files on coding style.

- **cake run**

  Starts a server on localhost:8080 which responds to index.html for the game.
3.3 Planning Tools

In this chapter we will explain which tools we used to gain insights about the amount of work to be done. These tools also helps our mentors to keep track of what we are doing.

3.3.1 GitHub issues

As explained in section 3.2.1, we use GitHub for version control. A very handy GitHub feature available is the Issues. A new issue ends up in the backlog. When the issue is chosen for the sprint, it gets a tag of the sprint, or so called milestone. That way we could easily filter the sprint log from all other issues. In figure 1 a sprint log is illustrated.

We also use labelling of the issues. Some issues clearly belong to each other. For example in figure 1, labels "login" and "controllers" are used to get an overview of what kind of work has to be done. We also use some default semantic issues like: bug, enhancement, duplicate, invalid etc. when needed.

To prevent the same issue from being worked on by different persons at once, each issue has to be assigned to someone before someone starts working on it. This also makes it easy to filter all issues assigned to you to keep your personal to do list.

Our GitHub issues overview is accessible via https://github.com/joostverdoorn/webrtc/issues.

3.3.2 HUboard

HUboard is an instant project management tool for GitHub which helps us with an overview of the sprint log. HUboard also makes use of labels, in our case: To do, In progress, Testing and Done. When the status of an issue changes, you can drag it to other column. We also benefit of the prioritization possibilities in the To do category to order the issues the way the Scrum product owner wanted it. See figure 2 for an example of our sprint log.

HUboard is accessible via http://huboard.com/joostverdoorn/webrtc.
Figure 1: GitHub issues

Figure 2: HUboard
4 Background information

4.1 WebRTC

4.1.1 Introduction

WebRTC, or Web Real-Time Communication, is an API definition[2] and a set of protocols that enables the web browser to directly communicate with other web browsers. At the time of writing, the API is still being drafted by the World Wide Web Consortium (W3C), the main standards organisation for the World Wide Web. WebRTC first saw the light of day when Ericsson Labs created a pre-standards concept in January 2011 and has evolved to the point that there are now advanced implementations in several modern web browsers. WebRTC was developed out of a need to allow for richer web applications that simply run in the browser without any third party software. With the advent of WebRTC, a myriad of completely new web applications are being developed that take advantage of the new possibilities it provides, for example BitTorrent clients for file sharing, voice-over-IP applications and customer service applications that allow customer clients to share their screens with customer service representatives. WebRTC is yet another omen of a future where more and more applications move from the desktop to the browser, and where consumers never have to download and install a piece of software on their operating system ever again.

4.1.2 The API

Through the WebRTC API the W3C aims to deliver a simple yet powerful interface to connect browsers through a peer connection, and has been process of drafting the API since 2011. Even though the specifications are still in development, Chrome and Firefox have implementations of WebRTC in their stable releases. However, they both do not yet fully implement the official API and code written for one cannot always run on the other. There is an ongoing effort to make them compatible with each other through a small JavaScript library called adapter.js, which equalises most of the API, although some inconsistencies remain.

4.1.3 The protocols

Reliably establishing a peer-to-peer connection between two consumer devices is not a trivial matter. While web servers are usually designed around accepting connections and have their own external IP-address, consumer devices are often protected by firewalls and located behind a Network Address Translation (NAT) service. WebRTC utilizes several technologies to enable peers to connect through environments that limit the connectivity of those peers. The protocols that are used for WebRTC have been developed by the Internet Engineering Task Force (IETF). Most of these protocols have been defined before
WebRTC was developed and have implementations in other areas than just the web browser. We detail the most important protocols below.

- **Interactive Connectivity Establishment (ICE)**, is a technique for NAT-traversal for UDP-based media streams\[8\], although in practice ICE isn't limited to UDP alone and can also be applied to TCP. ICE makes use of the offer/answer model, where party A sends a connection offer to party B including its preferred connection terms and aspects - such as bandwidth and protocol version - and party B responds to party A with the terms it agrees to. ICE generates a multitude of so-called ICE candidates, which define different manners for the other party to establish a connection.

- **Session Description Protocol (SDP)** is a standard created back in 1998 (and updated in 2006\[3\]) to assist in setting up streaming media connections such as Bluetooth headsets streaming audio to and from mobile phones and video conferencing where both video and audio are streamed between a multitude of entities. It’s used by ICE in the offer/answer process, where it encodes connectivity information into a Session Description packet.

- **Session Traversal Utilities for NAT (STUN)** is a technique to discover whether or not the application is located behind a NAT\[9\], and is for this reason heavily used by ICE. STUN does this by calling a so-called STUN-server and query the server to retrieve the IP-address and port of the requesting party. If this IP-address doesn’t match the locally known IP-address (link address), the application is located behind a NAT. The discovered external IP-address and port are then included in the connection offer or answer by encoding them into the Session Description and ICE candidates.

- **Traversal Using Relay NAT (TURN)** is an extension on STUN\[7\]. It allows ICE connections to be relayed by a TURN-server, which means that even when the user is behind an aggressive firewall or problematic NAT, ICE connections can be established by having the TURN-server relay all packets. In contrast to STUN-servers, which are freely provided for public use by a number of entities, TURN-servers are usually not free and have to be self hosted or rented from a third party.

4.1.4 **Bootstrapping**

Before a peer connection can be established, two peers have to exchange connectivity information in the form of session descriptions and ICE candidates. Since there’s no connection between the peers yet, these packets have to be routed through a third party that can communicate with either node, also called a signalling channel. The W3C WebRTC specifications do not specify a way to set up this signalling channel and instead leave it to the developer to coordinate the exchange of session descriptions and ICE candidates.
4.1.5 Streams and channels

Once a connection has been established, exchanging data is trivial. Media streams and data channels for sending raw data can be added quite easily, although the connection terms have to be renegotiated when one adds a stream or channel after the connection has been established. This means that that whole offer/answer process has to be repeated, with the upside that when a data channel is already open the session descriptions don’t have to be routed through the server but can be sent to the peer directly. To avoid renegotiation, one can also add a data channel or media stream to the connection prior to generating the offer, so that these are included in the initial session description.

4.2 Peer-to-Peer

4.2.1 Introduction

A peer-to-peer network is a decentralized distributed network architecture. The nodes in the network can communicate with each other without any interaction with the central server which keeps a track of all nodes. A node usually does not have full knowledge of the network but instead has a limited awareness of the peers it communicates with - its neighbours.

Applications There are countless applications where peer-to-peer technology is used. The most known application is definitely file sharing. The communication between peers does not proceed through the server and therefore peer-to-peer file sharing applications scale better for a huge amount of data sent through the network and benefit of high bandwidth. Some instant messaging software also relies on peer-to-peer networks. Beside saving traffic, instant messaging software also profits from the increased privacy because the messages are not stored on some server on Internet. Peer-to-peer is also used in many other applications for media streaming or content distribution purposes.

Infrastructure The simplest form of a peer to peer network is an unstructured network. Nodes are randomly connected with each other and share data inefficiently. All nodes have the same roles in this network. It is usually better to implement some kind of overlay network to help the nodes to better find each other and transmit data, by creating a structured network. For example the nodes with more bandwidth operate as supernodes and help the other nodes to communicate with each other. Node availability can also be one of the ways to structure the network. There are many implementations of structured networks, discussed later in section 5.2. The most of them are hybrid, which means that a central server is still needed for some functions for example when a node enters network. In Assignment formulation we already define that the type of peer-to-peer application will be a browser based game. Our exploration on peer-to-peer networks will consequently elaborate on our choice.
4.2.2 Node heterogeneity

Our goal is to create a stable and fast performing network with WebRTC technology. It is important to realise that every node which wants to connect to the network is different, and thus will not perform the same way. According to Jan Sacha [11], a node has many characteristics to consider. We will sum them up:

- A node has a session duration defined by the amount of time a node spends in the network or expected to stay in the network. In our case of an arcade action game, a player will join for a period between 1 minute and 1 hour. A small session duration will make the stability of network vulnerable.

- Availability of the node is the next property, defined by a fraction of time a node spends in the network within longer periods of time. This is important for availability of files for file sharing purposes. The files that should be distributed through the network are the network library files which allow the creation of P2P network, and the game files, which allow actually playing the game. Availability plays a role if the game files are being delivered through different peers. Longer availability means more peers in network which means a more stable network and better delivery. However this property is less important.

- A more important characteristic of a node is bandwidth. Our application should be able to send a huge amount of data to all connected nodes. This could be a bottleneck for network-heavy applications. Therefore by structuring a network, this can be an important factor.

- Sending a large amount of data leads to processing a large amount of data. That is why memory, CPU, GPU and disk space can be important; not only for the game experience but also for the network. Also the problems processing a game can lead to slower processing of the network functionality.

- Some of the nodes can be located behind firewalls and might only be able to connect to a limited amount of nodes. It is our goal to ensure that everyone who is able to make a WebRTC connection to at least one of the nodes (and that is not so hard, see section 5.1), can communicate to all others. Therefore, we have to provide a path that allows every node to reach every other node. This connectivity is one of our key properties that allows our players to play the game with each other.

- Jan Sacha also mentions the amount of shared files and the amount of traffic generated by the nodes as properties of the nodes. This is mostly important for file sharing purposes.

According to Sharad Agarwal and Jacob R. Lorch[1], one of the most important properties for online peer-to-peer games is the latency. Reaction speed is essential for the real-time arcade action games. A small delay can be experienced as “laggy” by users. Therefore, latency will be one of the key node properties, we should consider while implementing our network.
4.2.3 Structuring the network

Now that we know that there are differences between nodes, we should pick a network topology which fits the best for our network technology and arcade action game. A lot of research is done in this field, so we decided to check the existent topologies and not to try to come up with our own.


1. Every adaptation of the system should involve a limited number of nodes.
2. Each adaptation should also use limited information to reduce communication between nodes for scalability purposes.
3. The adaptation mechanisms should not break the network if a node functions with a limited or incorrect information. This can be expected in large networks.
4. A network should strive to keep an up-to-date track of all nodes connected to the network to ensure the connectivity of each node is constant.
5. The adaptation mechanisms should dynamically improve the network stability.
6. Different application should be able to be run on the designed overlay network.

These best practices should be taken in mind while researching the different implementation of the overlay networks. Especially guideline 4 is very important for our game application. Each player should always be aware of all other players participating in the game and they should always have a route to communicate with each other with as little as possible delay. This will be our base by investigating the peer-to-peer network topologies.

4.2.4 Supernode topologies and election approaches

A supernode handles data flow and connections for other nodes, it is like parent-child relationship. A supernode also acts as an equal to other supernodes - a sibling relationship. These connected supernodes form a supernode overlay network which handles the core functionality of the network. Network topologies that implement a supernode structure make use of the heterogeneity of the nodes. There is a lot of research done on supernode topologies. The key question in here is: How many supernodes are desired and which nodes are the best candidates?

Because of the enormous amount of supernode algorithms we will split them in different categories. Jan Sacha defines four groups that use a different supernode election mechanisms [11].
1. **Simple approaches** are the first generation algorithms. Here are no supernode election processes described or the election approach is very simple or static. Some algorithms in this group use a central server to assign a supernode. Simple approaches of supernode selection don’t produce an optimal network topology.

2. **Group based** algorithms split a node population in different groups and elect a supernode independently. The grouped are usually split by peer properties like network proximity. Group based algorithms have the advantage that a global supernode election problem can be decomposed in local group-level supernode selection problem. As disadvantage are these groups hard to create and to manage.

3. **Distributed Hash Tables (DHT) based** algorithms make use of the well-know DHT peer-to-peer system. In this class nodes use a DHT overlay to discover which nodes are close to them in a DHT defined space. The advantage of the DHT-based systems is that parent-children clusters can be split and merged very easily. However these algorithms can not select a supernode in real time and distribute the clients to the supernodes. This is an important aspect of our game because of the in 4.2.2 discussed session duration of the nodes.

4. **Adaptive algorithms** elect supernodes based on pre-defined rules, for example: the maximum number of clients a supernode can have.

We will choose the algorithm the fits the best for our purposes in section 5.2 and will focus on that approach.

### 4.3 MMOG

A massively multiplayer online game (MMOG) is a video game with the ability to support a large amount of players to play the game together at once. As the name states, this has to work over the internet. MMOG’s can choose to have a persistent world that is ever changing or the opposite, a world that is static and immutable.

#### 4.3.1 Dead Reckoning

After developing major parts of the game we noticed that the movement of other players was not going very smooth at all times. This happened due to slight variances in the latency of the network connections between two players. We improved the placement of players on screen by implementing dead reckoning. Dead reckoning tries to calculate the current position of an object by having a previous location, the elapsed time and the speed and direction the object moved since the last location.

As time we take the time that has elapsed since the last position arrived. As speed and direction we take the values from the previous packet. As soon as
we receive a new packet with a real location, we smoothly move the spaceship from the calculated fake position to the actual deterministic location.

Implementing this feature now makes sure that movements look very natural at all times.
5 Research

5.1 WebRTC

As WebRTC is an integral part of the project, a lot of time has gone into reading the official W3C WebRTC specifications and producing prototypes that demonstrate the uses of the technology. In this research phase of the project we’ve experimented with different techniques for establishing connections, methods of sending and receiving data and developing a useful application structure.

5.1.1 API differences

We discovered early on that even though the Firefox and Chrome web browsers are able to communicate to each other through WebRTC media streams, their ways of handling events and setting up connections are incompatible, and their data channels are not able to communicate with one another as well, regardless of the use of the adapter.js library mentioned before. Comparing their respective APIs with the official WebRTC specifications we found that Chrome’s implementation came closest, and keeping in mind that we were merely conducting research on WebRTC and not building an end-user product, we decided to drop support for Firefox. We had our hopes that Firefox would fix their implementation but as of this writing that has yet to happen. We do expect that Mozilla, Firefox’s producer, will eventually bow to the official specifications and that our library will work on Firefox as well.

5.1.2 Bootstrapping

Peer discovery and bootstrapping is an important aspect of any peer-to-peer network and is not a trivial matter. It’s not normally known among peers attempting to enter the network which peers are present and are able to provide an entry point into the network. Even if it is known to a peer whom to connect to, actually establishing a connection to that peer using WebRTC requires some external mechanism (the signalling channel) to coordinate the initial set up. We identify several different manners of going about this bootstrapping process.

- Using a web server, one can use AJAX requests to a web server which in turn routes information from one peer to the other, but this has the disadvantage that any peer has to be aware that the other peer is requesting a connection, or that the use of continuous AJAX polling, continuously requesting the server for an update, is required. The web server can maintain a list of nodes that can be used as entry point and pass this information on to new peers.

- Another way to establish the connection is to have all peers connected to a web server using WebSockets, a technology which provides full continuous communication between the browser and a web server, and is always available and open for sending and receiving data. Session descriptions
can then be easily routed through this web server. This manner would require server technology that can handle persistent WebSocket connections, such as Node.js.

- Apart from that, it would in theory be possible to use a DNS discovery techniques such as Apple’s Bonjour to detect WebRTC nodes on the local network, but this isn’t (yet) possible without the use third party plugins.

- Aside from using a web server or non-existent technology, it would also be possible, although cumbersome, to exchange the information in manually, like copying and pasting the session description or using QR-encoding to encode the session description into an image which can be scanned by another machine.

5.1.3 Data channels

In our prototypes, we found that data channels are by default rate limited to 30 kilobits per second, and if one attempted to send data faster, WebRTC would throw mysterious and indescriptiv error messages. After some research we discovered that the session description generated at initialization can be simply altered to increase the data rate by doing a regular expression replacement of the string flag that indicates the bandwidth, although this method is very unintuitive and feels a bit awkward. For now, however, this seems to be the only way to achieve a useful rate of data flow for the channel.

Another peculiar detail of the data channel implementation in Chrome, and one that makes it appear that data channels are treated as a second-class citizen by WebRTC developers, is that the session description packet to instantiate the WebRTC connection is too large to send through a WebRTC data channel. This has the implication that, without intervening, peers cannot renegotiate their own connections. This we found of course to be unacceptable and led to devising a way to split up packets before transmitting them, and reassembling them on the other side. In hindsight this is a must-have feature regardless of the size of the session description packet, as this means that we can make any packet arbitrarily (though realistically) large without encountering any problems.

5.2 Peer-to-Peer

This chapter describes the first choices we made in our research track and explains the road to the desired algorithm. The chosen algorithm is discussed very briefly and will fully be explained in section 6.2.

5.2.1 Structuring the network

WebRTC is a new technology and first we decided to run some tests of how the WebRTC unstructured overlay network will perform. So if all nodes will be connected to all other nodes and send a significant amount data through. Already after 15 nodes the network started to display connection errors and
sometimes failed to successfully send a message. Of course, we can quickly see that unstructured networks will perform weakly, so we should structure the network.

In section 4.2.4 we discussed different supernode election approaches, at this moment we choose one of the approaches to start our research on the best fitting topology. We choose the group based algorithms as the most suitable for our purposes because of the decentralized approach of supernode selection. However the rule sets of the adaptive algorithms can also be very valuable during the implementation of the group based algorithm.

5.2.2 Group based network systems

We would like to compare a couple of group based systems below. They are divided into three classes by how nodes are organized [11].

1. Location based systems are organised by their physical location, defined for example by communication latency between two nodes.

2. Semantic based systems are organised by their semantics. For example it could be a node position in the game world.

3. Grid based systems are organised by an administrative domain. The large-scale grid consists of small-scale grids what are actually a network themselves.

The choice of structuring the nodes is a tough one. Location based systems are better for optimal communication for all purposes, while semantic based systems can implement the structure for specific purposes. For example in our game, you would not send any data to a player who can not interact with you. We choose to implement a location based structure to keep our library useful for the different purposes. However during the implementation we should make it easy to switch to different organizations.

5.2.3 Location based systems

Now we made a choice which type of algorithm we are going to use, we start to discover the available systems. Below four proposed systems are described.

1. **Crown** [12] organises the network by the prefix of the IP of the user. The users with the same prefix are likely to have low latency and high bandwidth. Nodes in the same group elect a supernode with criteria of high bandwidth, high availability, large computational power and a low load. This approach looks very simple and sorting the nodes by IP can be very unreliable. Also no situation is described when there are too much nodes in the same group.

2. **Peer-to-peer Asymmetric file Sharing System (PASS)** [4] is like the name says designed for sharing files. The approach is similar to Crown by grouping the nodes by latency. An interesting idea here is that PASS
introduces a strong single node that handles all communication between separate groups. Also in PASS, a supernode keeps track of a backup supernode for case it leaves.

3. **PoPCorn** [6] relies upon an external algorithm like Vivaldi to structure the network in an n-dimensional space. First, the network distributes a token to a random node, after which this node negotiates with other token holders in the network. He advertises his token and calculates the combined repulsion force of all other tokens in the network. If the force is lower than a certain threshold for a certain number of time steps, a node may keep the token and becomes a supernode. This approach does not describe how a token generation works in a real time scenario.

4. **Wolf and Merz** [13] heuristic evolutionary algorithm tries to minimise the distance between the supernodes and their clients and the distance between the supernodes. This algorithm uses the local search principle to hopefully find a suitable solution. An interesting fact is that [13] illustrates that location-based supernode selection is a NP-hard problem.

We choose PoPCorn algorithm because the organisation of nodes based on latency aims to reach the fastest communication in the network. The token distribution principle seems reliable to dynamically elect the supernodes dependent on the real-time coordinates. However a lot of implementation details are missing which offers an opportunity to edit PoPCorn to our own needs. For example [6] only describes how the supernode selection works if all nodes are already in the network. In section 6.2 we will describe how we implemented PoPCorn in a continuously changing system.

### 5.3 MMOG

The game is our second deliverable and has the main purpose to demonstrate the functionality of our networking library in a fun way. To demonstrate the library in the best way, the goal is to have the game push the library to its limits, either bandwidth limits, latency limits or scalability limits.

As high raw data throughput usually isn’t an essential element for games, we have two options left: we can try to connect relatively few peers with a very low latency, which is great for a fast-paced action game, or a larger number of peers with a higher latency, which is great for role-playing-games where the action isn’t as quick and a high number of players is essential to the game play.

As testing with a very large number of players can be hard to arrange, and because WebRTC offers low-latency data channels we picked the first option.

The game we decided to create is a 3D dog fighting game around a planet with little alien spaceships. Every player controls a single spaceship in 3rd person view. Players can fire small projectiles at other players to damage their spaceship. When the player descends to quickly she will crash into the planet and die. To further demonstrate synchronisation in the network, the game contains a leader board showing all accumulated kills and deaths of active players.
Another reason we went for this concept is because initially we wanted to create an application or game that would make extensive use of orientation sensors in the user’s smartphone. This quickly changed to be a fully networked desktop game but we still liked the idea of being able to control the game with a smartphone. Therefore the ability to fly the spaceship by tilting the phone is included in this game.

5.3.1 Network utilisation

In order to keep the game playable with a decent amount of players (around 50) we need to make sure the game does not use too much bandwidth. Currently the game sends five packets per second about the players location and velocity. Every projectile results in one more packet emitted once. Bandwidth monitoring showed us that for every additional player in the game the total bandwidth usage increased with an average of 5 kilobyte/second (up- and download together). 50 people playing together therefore results in an average bandwidth usage of 250 kilobyte/second, which is by any terms acceptable. A player can fire two projectiles per second at most which results in only a tiny increase in network utilisation.

5.3.2 Graphics Engine

For the graphics of the game we have various options, amongst which HTML5 Canvas, inline SVG and WebGL, although only WebGL offers the raw graphics power we require for a 3D game. As WebGL itself is rather complex, there are various abstraction libraries for WebGL that wrap its functionality into a nice API, but none offer the level of sophistication that Three.js offers. Three.js is a fully-fledged 3D graphics engine that also includes functionality for - amongst others - importing textured models and shaders.
6 Design

6.1 Node Communication

The basis of the networking library is what we call the unstructured network: individual unstructured nodes that have the ability to connect and communicate to each other. The unstructured network - as the name implies - does not organize or structure itself, and requires an additional layer to be fully useful as a peer-to-peer network. It does have a lot of value: the structured network described in the section 6.2 relies heavily on the functionality provided by the unstructured network. Also, individual peer-to-peer connections, such as the connection between the game and the mobile controller, are easily established using unstructured nodes. In this section we will discuss the design and inner workings of the unstructured network.

6.1.1 Server technology

The choice of which technology to use for the bootstrapping process of peers is a relatively easy one. We require the server to know which nodes are present in the network, and ideally without the use of relatively slow databases. Node.js seems to be an obvious choice, where the server thread is persistent and a node list can be maintained in memory instead of on disk. WebSockets seemed to be the best technology for our purposes as it provides a convenient and reliable way for exchanging data, and this choice was backed up by numerous sources on the internet where the use of WebSockets as signalling channel was ubiquitous among WebRTC applications.

6.1.2 Mixins

To achieve multiple inheritance we use mixins, a design pattern common with JavaScript (and therefore CoffeeScript) development. Mixins work by having a class extend the Mixable class, which provides the functionality to include mixins.

6.1.3 Event-Driven Architecture

The networking library makes extensive use of events, for which the functionality is provided by a mixin. Events are used for most of the internal functionality of the nodes, and networking is also entirely event driven. Nodes communicate with each other by emitting events on their remotes, where they are transparently handled. This enables peers to execute complex procedures by triggering a sequence of events on their peers. Node.js is ideal for an event-driven application, as it is built with an event-driven approach in mind.
6.1.4 Controllers

Nodes as well as the server represent a Controller - an interface that provides some common functionality like the relaying of messages or responding to queries. Controllers bind all networking functionality together, and maintain a collection of their connected nodes.

6.1.5 Remotes

All remote communicating entities - a peer, a server or a client - are represented by a Remote object. The Remote base class provides a common transparent API for all communication, regardless of how this communication is implemented in the Remote sub class. Remotes are owned by the controller, as displayed in figure 3. Remotes offer functionality to either emit an event or query their

When a Remote object receives a message, it checks if the message is intended for its controller. If it is, it triggers the event packed in the message. If not, the message is relayed to reach its intended receiver.

Figure 3: Communication between server and nodes

6.1.6 Messages

Events to be sent to a remote are packed into Message objects that, beside the event, contains the event parameters, the sender and intended receiver, the travelled route and the maximum number of hops the message may take. Message objects can be serialized and deserialised for network transfer, and also provide functionality for splitting and reassembling messages that are too large.
6.1.7 Initialisation

As discussed in the section earlier, we route the connection negotiation between two peers via the server using WebSockets. The server, as other controllers, relays any message where the message indicates the controller is not the intended receiver. This has the advantage that the connection negotiation process is transparent and that when there’s already a WebRTC connection between two peers and they have to renegotiate their connection, this connection can be used. Figure 4 demonstrates the initialisation process of the connection between peers Alice and Bob.

![Figure 4: Initialisation of the connection](image)

6.2 Network Topology

In this section the whole network topology that we implemented is being explained. As discussed in section 5.2, we choose PoPCorn as supernode election algorithm. In this section we will explain how we implemented it. PoPCorn uses Vivaldi coordinate system and therefore we will discuss it too. We will discuss it in this section because the implementation of PoPCorn and Vivaldi differs from the implementation discussed in [10, 6].
6.2.1 Server communication

Our philosophy is to minimise the communication to the server. However, this may not be at the expense of the network reliability. Our network uses a server for several purposes, as detailed below.

1. When going to the game URL, a server distributes the needed files which are cached thereafter in the local storage.

2. When entering the network a new node receives a list of existing supernodes to choose from and coordinates the first connection.

3. When a node becomes supernode and vice versa, a message is sent to the server with the mutation information.

4. Once in a while a supernode updates its supernode list from the server to be 100% sure the list is up to date.

6.2.2 Vivaldi

When a node joins the network, it sets its 3-dimensional coordinates to a random vector with values between [-2, 2] to ensure diversity in the coordinates. With an interval of 2 seconds a node asks its peers to for their respective positions and computes the latencies. This information is used to calculate its own position as follows: for each peer, the node tries to set its distance to that peer equal to the latency to that peer, like a spring system. This process happens quickly at first, with a node changing its position very rapidly as it attempts to find its optimal position within the network. Later on it slows down, to avoid oscillations when latency varies [10].

6.2.3 PoPCorn

Now, we have a coordinate system we can start electing supernodes. When a parent can not accept more children it generates a token and distributes it to the random child - the game of tokens can begin. When a node receives a token, it broadcasts the token to all supernodes holding a token. The supernodes save the token information and send their own token back. After a certain threshold, when all tokens assumed to be received a token force is being calculated based on the distances of the other tokens. If the token position is close enough to own position, the node becomes a supernode. Other way all supernodes are being asked to give the closest child to the token which receives a that token - and the game of tokens starts over [6].

6.2.4 Quirks

Vivaldi and PoPCorn describe the basic idea of how the network topology should look like and how it evolves. However not all details are explained in the papers. In this subsection we would like to tell about our additions to the network topology.
Vivaldi performs weak if a node has only one connection to its parent. A stable foundation is needed to find a place in the space. That's why every node has connection to at least three supernodes. Not only for position purposes but also for backup when a parent leaves. The first three nodes which join the network automatically become a supernode. New nodes joining the network receive a list of existing supernodes from the server and choose a random supernode as parent. When a node has settled it gets a recommendation from its parent with the closest supernode. If it is worth it, the node switches from parent. After switching, we check if a node has children then it can lose its supernode status. So is the amount of supernodes dynamically adjusts to the size of the network.

An interesting detail is how the nodes are getting a parent node. Due to continuously changing network where the nodes are changing of the supernode status, asynchronous errors are very easy to get. That is why we use a query functionality to request adoption of a supernode. A supernode can always reject new requests if the situation has changed.

As just said, asynchronous requests make the network vulnerable for errors. We implemented some functionality that can automatically detect and fix errors. Checking for inconsistencies is an important part of the network to ensure the network keeps running. Also before a node leaves the network it informs all connected nodes with its departure.

6.2.5 Illustration

We would like to illustrate our technology with an example shown in figure 5. Legenda is as follows:

- Red bullets illustrate supernodes.
- Green bullets illustrate normal nodes.
- Yellow bullets illustrate normal nodes with a token.
- Violet edges illustrate a parent-child or sibling connection. Random connections to other nodes are not shown.
- Yellow edges illustrate a token force.

6.3 Game

Our game Orbit Impossible is a HTML5 3D dogfighting game. All players start in a small aerial vehicle on the surface of a planet. From here they can fly around the planet and shoot down other players with little bullets. Because we wanted the game to be easily playable for short amount of time without a steep learning curve, we picked the immutable world variant. An immutable world is easy to understand and does not require much explanation to the user. The only interaction a player can have with the world is landing on it, and when approaching with a too large velocity, crashing into it. The other possible interaction is shooting down other players.
To play the game, the user visits a website. While the game is loading in the background and a connection to the network is established, the user gets some basic information about the game. After clicking this screen away an option to pick the controller is presented. Currently two different options are available: Controlling by mouse and keyboard or controlling via an Google Android smartphone (figure 6). In the case of mouse and keyboard, the control mapping is shown (figure 7). When the mobile phone is selected, the user is asked to scan a QR code to connect his or her phone to the game (figure 8). After this is done, a similar control mapping is shown.

After pressing the boost button, which is used to ascend from the planet, the player spawns in the world and starts to fly upwards (figure 9). The goal of the game is to kill as many opponents as possible while trying not to die yourself. Killing other players is possible by shooting tiny projectiles from a cannon attached to the bottom of the spacecraft (figure 10). When a player dies, scores for the victim and the killer are updated in the scoreboard and the deceased player can respawn by pressing the boost button again (figure 11; the death in that screenshot resulted from a suicide, so no killer is visible).
Figure 6: Controller selection in Orbit Impossible

Figure 7: Mouse/Keyboard control mapping in Orbit Impossible
Figure 8: QR code to connect a mobile phone to Orbit Impossible

Figure 9: A newly spawned player
Figure 10: A player firing projectiles

Figure 11: The leaderboard
7 Implementation details

7.1 Technology

7.1.1 WebSockets

WebSockets allow a client-server application to create a full-duplex communication channel on a single TCP connection. This helps us to get near-realtime communication between our central server and all other peers and it is used to exchange the data needed to bootstrap the WebRTC connections between peers.

7.1.2 WebGL

WebGL (Web Graphics Library) is a relatively new technology that allows developers to utilise OpenGL-like 2D and 3D graphics programming directly in the browser. Most modern browsers implement WebGL and allow it to execute on the GPU, making it very fast. WebGL is used in the game through Three.js, which will be described in section 7.2.5.

7.1.3 Canvas

Our interface of the mobile controller is fully designed in canvas. Canvas is used to draw 2D graphics on the fly and is supported by all modern mobile browsers. A canvas element is defined in HTML and is editable through JavaScript and in our case CoffeeScript. A lot of 2D drawing functions can be used.

7.1.4 Application Cache

Our network library and game don’t use any dynamic server side components. Therefore caching everything is a feasible option. Unfortunately browsers may decide to re-download files even though they are still in the cache.

Luckily the HTML5 Application Cache offers the ability to permanently cache a website on the users computer and load the website even if the internet is not available (yet). Obviously, connecting to the network will not work without any internet connection, but even with an active connection loading the website will go much faster.

The only thing required to get this working was adding a new attribute to the HTML-tag in the game and creating an Appcache-manifest marking the big files like textures and object-meshes to be cached.

7.2 Frameworks

In this project we used several tools and frameworks that are essential for our network library and game to work.
7.2.1 Node.js

Wikipedia currently has a good description of what Node.js is: “Node.js is a software platform that is used to build scalable network (especially server-side) applications. Node.js utilizes JavaScript as its scripting language, and achieves high throughput via non-blocking I/O and a single-threaded event loop.”

In our project this platform is used on the central server in the network library. It is used to deliver the client-side library code, the actual application (in our case a game) and kick start the WebRTC communication between new peers.

7.2.2 RequireJS

RequireJS is a module loader for JavaScript. It gives developers the ability to load other classes in an easy to read and write way.

In our project it is used in every piece of code we made. RequireJS works in both the clients browser and also the Node.js server so sharing code was easy to do thanks to RequireJS.

7.2.3 Socket.IO

Socket.IO is a library that allows developers to easily set up real-time two-way communication channels between browsers and a Node.js server. It tries to use WebSockets for this, but several fallbacks are included to make Socket.IO work in nearly every browser.

Since our project is limited to Google Chrome 28, WebSockets are available and the fallback methods are not necessary. We use Socket.IO to send all WebRTC initialization information from and to peers that have not yet entered the network.

7.2.4 Underscore

Underscore is a library for JavaScript that adds many useful functions to extend the usefulness of Arrays and Objects in JavaScript. Our project does not heavily rely on this functionality but it makes some tasks easier to do.

7.2.5 Three.js

Three.js is the only well documented and full featured JavaScript 3D library that renders 3D objects with WebGL. The library implements all we needed for the game design and we even used it for visualisation of the nodes (8.2.1) and visualisation of the game world (8.2.2).

---

7.2.6 KineticJS

KineticJS is a HTML5 canvas JavaScript Framework that makes it easy to render canvas elements and also implements multi touch event for mobile browsers. Using an external library helped us a lot while designing the mobile controller.

7.2.7 Adapter.js

Adapter.js is a tiny library that suppose to take care of the differences in the WebRTC implementation in Firefox and Chrome. For example Chrome implements “webkitRTCPeerConnection” while Firefox implements “mozRTCPeerConnection”. Unfortunately not only the names differ, the implementation of WebRTC is also different. Firefox does not implement the needed event handlers specified by W3C to set up the connection and implements an old specification from 2011.

7.3 CoffeeScript

CoffeeScript is a programming language and it compiles own source to JavaScript. source so it can can be executed in browsers. CoffeeScript has many advantages above JavaScript. We are very happy we made a decision to use CoffeeScript and would like to explain the advantages we loved the most.

- Cleaner syntax. We have been paranoia about the quality, readability and the beauty of the code. CoffeeScript syntax helped us to keep the code clean.

- Fat Arrow (=>) allows the access to the coffee objects context inside the callback function.

- Object Orientation with “class” keywords. JavaScrip self uses prototype approach for object orientation and does not support classes.

We are planning to release our library and the good part is that the library can be used in all JavaScript projects because the compiled version of the library is in JavaScript.
8 Quality assurance and testing

During the development of our library and demonstrational game we used several frameworks and tools to assure the quality of the written code and to test it properly. We also had access to the expertise of the Software Improvement Group (SIG) to review our code.

8.1 Frameworks

The used frameworks for testing are essentially Jasmine for Behaviour-Driven Development and CoDo for automated code documentation generation.

8.1.1 Jasmine

Jasmine is a Behaviour-Driven Development (BDD) testing framework for JavaScript. It allows the developer to easily convert user-stories to proper BDD test-cases. These test cases can then be run from a command-line interface to create an easy to read overview of failing and passing tests.

This helped us a lot in debugging functionality that broke older functionality.

8.1.2 CoDo

Good code needs to be documented well. CoDo specifies a way to write comments in CoffeeScript code that allows CoDo to easily extract this meta-information about the code and automatically create documentation.

8.2 Tools

8.2.1 Node Inspector

We made our own Node Inspector in WebGL that shows all nodes in a 3-dimensional space. This tool has been very handy during debugging Vivaldi and PoPCorn. figure 5 illustrates an example network configuration.

8.2.2 Map

Another tool we developed helps us in determining the locations of all players in the game world. It is comparable to the Node Inspector but instead of showing all nodes it shows the planet and as green dots around it all players. A screenshot of this can be seen in figure 12.

8.3 External Tools

8.3.1 JS Coverage

JS Coverage is a framework that "instrument" JavaScript code to include counting executions of lines of code before Jasmine runs the test. After running the
test, JS Coverage will then read out how often a specific line of code was executed and creates easy to read reports about this.

Unfortunately JS Coverage only reports line coverage, which of course does not say a lot about code. A better framework would’ve been Istanbul which features a lot more (for example branch coverage). Unfortunately it turned out to be a lot of work to get Istanbul to work with our CoffeeScript code. Therefore work on this was abandoned to not spend more time on getting the testing framework to work than to actually develop the library and game.

8.3.2 cake

With cake it is possible to create buildscripts that run on every computer independent from settings and operating system. This helped us a lot in building the project on regular basis and automatically run all tests to ensure the codes quality.

8.3.3 CoffeeLint

CoffeeLint checks the code to ensure a coherent style of coding. This helps reading the code because it is ensured that code-aspects like whitespacing are the same in every file.

8.3.4 Sublime

The text-editor Sublime helped us slightly by keeping the code clean by automatically removing unnecessary whitespace at line endings and having good
8.4 SIG

The Software Improvement Group evaluates the quality of code by automatic and manual analysis. For this project we are required to send our code to SIG twice. Once halfway of the project and the second time at the end of the project. After sending it in the first time we are expected to read the comments carefully and adjust the code accordingly so the second feedback points out less issues. Our first feedback of the SIG can be found in Appendix C.

After this first review we followed up on most of the suggestions given by SIG. We did move the game to its own directory so files of the library and the game are no longer in the same directory. We also refactored long functions to consist of multiple smaller functions. Additionally we added method and class documentation to now have 100% of the codebase (excluding tests) textually annotated. We did not remove the code duplication between the network library and the game because eventually these two components will be split up in two repositories where the game can only use the public interface of the library and the library should never even know about the game. Therefore the shared code in these two deliverables cannot be placed in a common directory.

The second review by SIG can be found in Appendix D.
9 Conclusions

In Assignment formulation we defined our research question: “In what way can one create a dynamic peer-to-peer multiplayer-game-network-topology that scales with the amount of users participating by only using WebRTC?”. We answer this question through answering the sub-questions.

9.1 How can one distribute essential network tasks over all peers?

In Superno de topologies and election approaches we identified the tasks that have to be performed to maintain the network ensuring the network connectivity and handling data flow for unconnected nodes. We saw that we can assign supernodes to carry out these tasks. They ensure the network connectivity by maintaining a connection to all other supernodes, where it is the child node’s own responsibility to maintain a connection to a supernode, and they route data flow between unconnected nodes when a full connection between them is not deemed necessary or when they attempt to establish this connection. To make sure supernodes are elected when necessary and to spread them out evenly throughout the network, we identified PoPCorn as a suitable technique to accomplish this.

9.2 How can the impact of unpredictable circumstances be reduced?

Asynchronous requests make a dynamically changing network vulnerable for errors. We attempted to ensure the network stability by different methods:

Prevention methods:

- Every change in the state of the node is propagated to the involved nodes. For example: getting a token or becoming a supernode. A supernode state is also propagated to the server.

- Once in a while each supernode get a list of all supernodes from the server and ensures it is connected to all other supernodes.

- Once in a while a not supernode gets a list with supernodes from its parent to ensure it is always connected to three supernodes.

- When a node leaves the network, it handles its leave by sending a message to all neighbours and in case of supernode choose a new parent for its children.

Recovery methods:

- Once in a while all broken relationships are fixed with the new information from server.
Once in a while all, eventually unused, token information about other nodes is deleted.

9.3 How can all data reach all peers in a bandwidth-efficient way?

There are two aspects to discuss about bandwidth and latency efficiency: making a graph of nodes and finding a route between the nodes. Also the transmitting technology itself is discussed.

9.3.1 Coordinate system

As in 6.2.2 discussed we use Vivaldi to organise our network by latency. Here, the nodes that are close to each other in a n-dimensional space are most likely to connect to each other. Vivaldi ensures that all edges in the node graph represent the current latency between the peers.

9.3.2 Routing

After the network is structured we can send messages through the network. As in section 9.1 we will use supernodes for this purpose. The supernodes have an important role to provide for communication between all nodes. At this moment if a child node wants to send a message to some other node, it sends it to its parent. If the intended receiver is not a child of the same parent, the parent relays the packet to all its siblings to reach the intended receiver. All supernodes are connected with each other what makes the network somewhat less scalable.

Scalability is currently still an issue. This is a direct result of all supernodes having to stay connected to all other supernodes. Eventually some of these connections have to be dropped to allow an expansion of the entire network. More about this future process is explained in section 10.1.2.

9.3.3 WebRTC

The underlying WebRTC technology is also a factor that is important for data transmission. WebRTC offers a reliability flag. Unfortunately Chrome only supports unreliable WebRTC data channels. We hope that it will be implemented in the near future.

9.3.4 Speed

The network library hardly has any overhead for the network at all and almost all traffic results from data the application using the library sends. In the case of ORBIT IMPOSSIBLE, which is not optimised at all, this still results in quite a pleasant player experience with only 250 kilobyte/second bandwidth usage with 50 active players.
10 Discussion

10.1 Network topology
At this moment we have implemented an advanced network topology and we are still improving it. In this section, we would like to discuss a couple of possible improvements.

10.1.1 Supernode overlay network
section 9.3.2 already describes the routing method. At this moment is far from ideal. A lot of research is done in field of managing supernode connections. Adaptive algorithms discussed in 4.2.4 offer a solution to connect the supernodes if the latency is smaller than a threshold. More research is needed to determine if the network stays reliable after this change.

10.1.2 Routing tables
At this moment all supernodes receive a message when a message has to be delivered. It should be a good idea to remember the route after the first time the message is delivered. This is one of the examples of the optimizations we can implement for routing of messages.

10.1.3 Comparing network topologies
In the beginning of the project we made a decision to implement PoPCorn as a network topology for the game. We are sure that PoPCorn is a suitable choice that with our additions fully fits our needs. If we had more time we would like to look at adaptive algorithms for supernode selection and compare it with our current PoPCorn implementation.

It is also interesting to use the game coordinates instead Vivaldi coordinates for organisation of the network. The functionality is already available but unfortunately we had no time any more to run the proper tests.

10.2 Network security
During this project the main goal was to create a scalable distributed network using WebRTC. As requirement we posed that we would be in a secured research environment without unknown threats from the outside and therefore securing the network against malicious users was not required.

To use the developed network in a production environment, security is an issue though.

In this chapter we will go into possibilities to make the entire system more secure. We begin with preventing the network to collapse after being flooded. Then we continue with the game’s state integrity after malformed packets arrive or get lost. Last we will discuss problems with security of the game with the current network implementation.
10.2.1 Overloading the network or server

Currently all nodes in the network connect to a central server. This is done to track all nodes so new nodes can easily find an entry point to the network. The WebRTC connection initialization is also sent via this central server. Once the first WebRTC connection is established, this connection will get used to establish further connections with nodes in the network.

Currently the central server, nor nodes in the network are prepared for a user requesting uncountable connections and the server and all nodes will happily reply to all requests. In a final product this should be prevented.

One option would be to limit the amount of connections a single user can open to the central server. When the central server detects many incoming connections from the same computer, it can just block this user for a certain time. This will reduce the serverload significantly.

When a computer is already connected to the network it could decide to flood the network. This can be countered by letting every node check if a user did not send too many messages in the recent time. If the amount of messages reaches a certain threshold, the node could decide to not relay a single packet from the spammer and disconnect the connection. Eventually all sane nodes will block the spamming node because they all use the same rules. This results in the malfunctioning or malicious spamming node to be split from the network.

10.2.2 Game integrity

Another problem is the integrity of the games state. In the current implementation the game will broadcast the information about its own player (location and speed) five times a second to all other players. This information is picked up and treated as if it must be the truth.

We cannot expect this package to be correct though, because WebRTC has two different options to create a data channel: a reliable variant and an unreliable variant. Unfortunately Google Chrome currently only supports the unreliable variant. A way to do this better is to only send updates when it is absolutely mandatory. Combined with only sending state changes instead of full updates, transmitting these new packets via a reliable data channel, should reduce the bandwidth substantially and reduces the risk of out-of-sync players.

10.2.3 Game security

Apart from the ability to easily overload the network there are more issues regarding the security of the network and the game. The biggest problem we are facing is the total lack of sanity checks on received packets. First of all we expect all packets to be in a specific format and if a packet arrives that does not fit in this format, the resulting behaviour will be unpredictable. Depending on where the packet contains malformed information, the entire game might stop working, the connection to the network drops or nothing bad happens at all.

Another flaw arising from the lack of sanity checks is the ability to inject arbitrary packets about other players into the network, allowing a user to tele-
port himself, but if done right also others to a desired place. Since the most difficult task in the game is actually hitting other players, the ability to do harm is limited with teleporting. A bigger problem is the ability to create an uncountable amount of projectiles flying in all directions and automatically remove all projectiles by other players. Doing so is easier than teleporting another player and renders the game unplayable for everyone.

To counter these issues, the networking library has to first check if a packet fits the defined format. It also has to include a reliable way of finding out who really sent a specific packet. If it turns out that the claimed sender and actual sender don’t match up, it should just get ignored.

The game then does not receive corrupt or wrong data but the problem of projectile manipulation still exists. To counter this, the game has to tie each packet that modifies an entity to the owner of said entity. If someone else tries to modify an entity that he or she does not own, it should just get ignored. To prevent players from creating more bullets than allowed or teleporting themselves, all players should check if all other players obey the game rules. If this is not the case (this could also happen if two users play on two different versions) the game should drop the connection to the malfunctioning node because otherwise these two clients would just ignore each others packets which makes the connection between them useless.

10.3 Cross Platform

Currently our library only works with Google Chrome 28 and above. This limitation has to do with the lack of implementations of WebRTC in most other browsers. The only other browser with WebRTC support that is commonly used is Mozilla Firefox. Unfortunately this browser does not implement all callbacks as the W3C defined them in the API. Luckily it is only a matter of time until more browsers start implementing WebRTC and Mozilla Firefox switching to the official API definition and this problem should fix itself.

In our research we neglected Opera because of its tiny amount of the marketshare (1.15% worldwide and even less in the Netherlands; see fig 13 for reference).

10.4 Peer discovery

In our network topology implementation we used a central server to help a new incoming node to connect to the rest of the network. The server provides a new node with a list of the available nodes and supports the WebRTC connection process. At this moment it is unfortunately not possible to find other nodes online without a central server but it is possible to find the nodes in your local network. In our game we use a mobile device as a controller for the game. The server is still helping to set up this connection. It should be a great improvement to get the server out of this process.

There is hope: Zero-configuration networking (zeroconf). There are numerous implementations of this protocol like Apple’s Bonjour and Avahi for
Linux. Microsoft has also an implementation of this protocol. Unfortunately these technologies are still not supported by Chrome. It should be nice to implement it in the future.

### 10.5 WebRTC impact

During this project we have gained a lot of knowledge on WebRTC and researched it on several fronts. For the duration of the project and even before, we followed the latest developments of it. Three months ago, there were several demo’s of WebRTC whereof the most were the video conference applications. At this moment the internet is rich with tutorials, demo’s, libraries and different types of applications you can use WebRTC for. Take for example a CDN service built with WebRTC. A website can delegate the distribution of website files to the visitors of the websites by practically just adding a JavaScript file to the page.

We totally believe that WebRTC will make in the rapidly evolving Web society. The success of WebRTC benefits a lot from the success of JavaScript web-applications that run JavaScript both client and serverside. A four year old Node.js server started the revolution.

The battle for Internet browser market share is still not ended. Every month some new features are being introduced to the world. As result a modern browser can take over the roles of all other desktop application. We believe that
WebRTC will play an important role in this take-over. In Applications discussed application of peer-to-peer can now easily evolve to the browser. Combined with the rest of technologies the browser already offers and the ease of programming of the web application, it should be a matter of time these new applications will attract users.

Unfortunately at this moment Internet Explorer does not support WebRTC and as discussed earlier not all browsers support the newest implementation of WebRTC. This will definitely slow the adoption of WebRTC by developers. At some point users will desire the upcoming browser applications based on WebRTC and that will the moment all browser producers will implement the technology. It will not happen at once but like gradually like the adoption of HTML5. Lets hope this adoption takes big steps.
11 Evaluation

During our bachelor project we have been involved in a software development project completely from the beginning. In this chapter we would like to reflect on our experiences during this project.

11.1 Technical challenges

11.1.1 Unit Testing

After setting up our development environment we noticed that testing in this environment wasn’t as easy as expected. Our combination of having shared code between the server and the clients, that code being written in CoffeeScript and included via Require.js turned out to be something most standard testing frameworks for JavaScript were not able to accomplish.

We solved this by daisychaining several processes. First we found a modified CoffeeScript to JavaScript compiler that also adds code-coverage information to the code as explained in 8.3.1. After this we would tell Jasmine to execute the tests as JavaScript and use Require.js and at the same time keep track of the global code-coverage array JS_coverage created. After this was done, JSCoverage would generate the report about the code-coverage and Jasmine would create the report about failing and succeeding tests.

As pointed out in 8.3.1 as well, we wanted more than just code-coverage. We found a tool called Istanbul that could also generate statement, branch and function coverage but it turned out to be a lot of work to get this working with CoffeeScript because in order to instrument the code it used a JavaScript parser which we had to replace with a CoffeeScript parser with the same functionality. Technically this is possible but would require too much time to do during our project.

11.1.2 Game Testing

Testing the game was a challenge as well. We did make this easier for ourselves by creating bots that join the game and randomly fly around and fire projectiles. These bots help a lot for looking network errors and performance problems but running multiple bots (10+) on the same computer is hardly doable because of the high amount of calculations that need to be done. This can be solved by sharing parts of the bots memory with other bots so calculations for collision detections is only done once per frame instead of once per frame per bot. To overcome this problem we added more computers to simulate bots because adding the shared state would cost too much time to be useful.
11.2 Process challenges

11.2.1 Scrum

The most of us haven’t experienced scrum in business environment. After some reading in and assigning roles to ourselves, we were ready to go. What we did not expect in the first couple of weeks is that gaining more insight in WebRTC technology or gaining more insight in node topology does not fit well in scrum philosophy because the amount of time the task is going to last is hard to predict. That is why we started with scrum in week 3 when we fully specified what we are really going to build and deliver.

An interesting detail is also that because of the vacation period, we have had a second scrum product owner who first had to get in to the project. It succeeded very well and he brought new insights with him. It was good experience because falling out of the scrum team also happens in real work environment.

11.2.2 Behaviour driven development

In our plan of action we described that we will maintain the behaviour driven development discussed earlier in section 3.1.2. In hindsight it turned out to be a bad decision. BDD can only work if a clear product requirements are defined in the beginning. We built a prototype which later became an end product. After the prototype was ready and the end goal was clear, we decided to test the whole existing application and try to continuously achieve the 100% code coverage.
A Glossary

AJAX
Asynchronous JavaScript And XML is a group of techniques that allow for asynchronous applications on the web, and is usually used for asynchronous data retrieval from a server.

API
An Application Programming Interface specifies a way for components to interact with other components.

DNS
Domain Name System is a naming system for devices in a network.

W3C
The World Wide Web Consortium is a standards organisation tasked with developing and maintaining Web standards, such as HTML5 or CSS3.

ICE
Interactive Connectivity Establishment is a technique for NAT-traversal for UDP-based media streams.

IETF
The Internet Engineering Task Force is a standards organisation tasked with developing and maintaining Internet standards and protocols.

Latency
Also known as ping or two-way round-trip-time, latency is the time it takes for a packet to reach a remote party plus the time it takes for an acknowledgement to arrive.

Library
A drop-in piece of software with some specific functionality accessible to developers.

NAT
Network Address Translation is a technique that maps internal network addresses to external network addresses.
**Overlay networks**

An overlay network is a network built on top of another network, usually using the old network to implement new functionality.

**P2P**

Peer-to-peer is a method of connecting two entities directly, without intervention of a server.

**SDP**

Session Description Protocol is a standard to assist in setting up streaming media connections.

**TCP**

Transmission Control Protocol is a protocol for sending data reliably across a network.

**STUN**

Session Traversal Utilities for NAT is a technique to discover whether or not the application is located behind a NAT and to discover the external network address of the application.

**TURN**

Traversal Using Relay NAT is an extension on STUN to allow ICE connections to be relayed by a TURN-server.

**UDP**

User Datagram Protocol is a protocol for sending data unreliably but quickly across a network.
B Bibliography

References


C Software Improvement Group Review 1

The following text is the raw Dutch feedback received by Eric Bouwers (e.bouwers@sig.eu) from the Software Improvement Group on August 15, 2013.

Vanwege de gebruikte technologie, Coffeescript, is het op dit moment heel lastig om een score voor onderhoudbaarheid uit te rekenen volgens ons standaardmodel. Een combinatie van grove statische analyse en manuele analyse wijst echter uit dat de codebase relatief klein is. Daarnaast maakt de aanwezigheid van tests het makkelijker om aanpassingen te doen zonder de huidige functionaliteit te breken.

De keuze voor het gebruik van de Coffeescript technologie als basis voor dit project is waarschijnlijk onderbouwd in de documentatie. Het gebruik van deze technologie is op zichzelf geen minpunt, zolang er rekening mee is gehouden dat door de benodigde voorvijven minder makkelijk nieuwe ontwikkelers gevonden kunnen worden.

Wat wel opvalt is het hoge percentage duplicatie (16%). Alhoewel de documentatie aangeeft dat er zowel een library als een spel gemaakt wordt is het uit de directory-structuur niet meteen duidelijk welke code bij de library en welke code bij het spel hoort. Daarnaast is het mij niet duidelijk waarom er code gedupliceerd moet worden, kan er geen 'common' library gemaakt worden welke door zowel de te ontwikkelen library als het spel gebruikt kan worden? Het is aan te raden deze keuze nogmaals te overwegen.

Daarnaast zijn er een aantal methodes die aan de lange kant zijn. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvouder te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'update'-method in 'entity.player.coffee', zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaar-regels zoals bijvoorbeeld '# Add tilt forces' en '# Add thrust straight downward from the player. If the player’s boosting,' zijn een goede indicatie dat er een autonom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Het ander code-niveau punt heeft betrekking op het functiecommentaar. In een deel van de gevallen bevat dit commentaar niet meer informatie dan al gegeven wordt door de naam van de methode. Het is aan te raden dit type commentaar uit te breiden of te verwijderen om zo de codebase overzichtelijker te maken.

Als laatste de opmerking dat niet alle code meegestuurd lijkt te zijn. Op verschillende plekken worden JavaScript files uit een 'meshes' directory geladen welke niet aanwezig is in de aanlevering. Mocht deze geschreven zijn voor dit project dan ontvangen wij deze code ook graag in de tweede aanlevering.

Over het algemeen lijkt de onderhoudbaarheid van de code bovengemiddeld te zijn, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelafase. 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.
D Software Improvement Group Review 2

The following text is the raw Dutch feedback received by Dennis Bijlsma (dbijlsma@sig.eu) from the Software Improvement Group on August 27, 2013.

In de tweede upload zien we dat het codevolume met ongeveer 20% is gegroeid. Een score voor onderhoudbaarheid kunnen we helaas niet berekenen, omdat onze tooling hiervoor CoffeeScript (nog) niet kan analyseren.

Het eerste dat opvalt is dat jullie de directorystructuur hebben aangepast. De library is nu duidelijker gescheiden van de code die de library gebruikt, waardoor de directorystructuur nu overeen komt met de door jullie bedachte architectuur.

Een aantal lange methodes is inmiddels opgesplitst, wat de leesbaarheid en testbaarheid ten goede komt. Er zijn nog steeds een aantal methodes die aan de lange kant zijn (bijvoorbeeld Entity.Player.onLoaded, node.queryTo, de constructor in server.coffee). Probeer de meerderheid van de methodes 20 regels of kleiner te houden. Het is prima als je daar af en toe overheen gaat, maar nu gebeurd het echt te vaak.

Over de duplicatie zeggen jullie in het readme-bestand het volgende:
"We hebben de duplicerende code niet in een common/ map gestopt omdat de library en game uiteindelijk in twee verschillende repositories gepubliceerd zullen worden en het spel alleen de publiceke interface van de library mag gebruiken en de library niet van het bestaan van het spel af mag weten."

Ik begrijp jullie redenering, maar dit is niet de beste manier om met libraries om te gaan. De methodes van de library naar de directory van het project copy/pasten is een vrij primitieve vorm van dependency management. Er zitten ook risico's voor de onderhoudbaarheid aan: je weet niet meer zeker welke versie van de library je gebruikt, en er is een kans dat iemand per ongeluk de code van de library aanpast. Er bestaan voor de meeste technologieën tools om met zo'n situatie om te gaan, zoals Maven voor Java en Gems voor Ruby. Ik weet eerlijk gezegd niet of er iets soortgelijks voor CoffeeScript bestaat, maar je zou eens kunnen kijken.

Tot slot is het goed om te zien dat er naast productiecode ook veel testcode is toegevoegd.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.