HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

Distributed Systems
& Ledgers Lab

Distributed WebRTC Signaling

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2018

Authors: Demian Thoma, Lukas Röllin
Advisor: Prof. Dr. Thomas Bocek

Abstract

Many of the services available on the internet are centralized. To improve scalability and availability, complex distributed architectures have to be designed and implemented.

Starting right away with a decentralized design on the other hand can scale better and increase availability with a growing network.

WebRTC uses a peer-to-peer connection between browsers. The developer has to provide a communication channel for signaling between the two browsers, before the WebRTC connection can be established. In most cases this is achieved using a centralized server.

DWRTC extends WebRTC with a decentralized connection setup. Users connect to different nodes on the Internet. These nodes are connected via a P2P network which stores the routing information. The connection setup messages are routed through this network. The WebRTC connection can then be used to send data, audio and video directly from web browser to web browser.

This term project implements this idea. As a proof of concept, it also includes a video call application using DWRTC to establish calls to a partner.

Management Summary

Motivation

In today's Internet, many of the available services are centralized. Different solutions are introduced to allow those services to scale as they do not have this ability built-in. However, decentralized applications allow for easier scalability.

WebRTC uses peer to peer connections out of the box. This requires a connection setup which is predominantly centralized. A fully decentralized WebRTC, on the other hand, enables anyone with a current web browser to use and benefit from its architecture.

Idea

The main idea is to decentralize WebRTC by using a distributed hash table (DHT) for storing routing data to exchange connection information.

As a proof of concept, an implementation of a video call application using DWRTC was developed, allowing users to establish video calls.

Result

The architecture consists of two layers supporting an independent usage of the underlying layers:

1. Decentralized backend layer for WebRTC connection setup (Kotlin).
2. Frontend layer for WebRTC (JavaScript).

In addition a video call application has been built. By using the frontend layer it allows users to establish a peer-to-peer video call via the backend layer to a partner.

Outlook

There are multiple possible improvements that can be made to the application. The most important improvements would be providing client side encryption or signing of messages routed through the backend layer.

Acknowledgements

We would like to thank the following people for their help with this term project:

Prof. Dr. Thomas Bocek for his helpful inputs on P2P problems and the complete project.

AnneMarie O'Neill for her thorough reading, correction of this document and useful feedback.

Angelo Gründler, Robin Bühler and Denis Dion for their timely feedback on technical aspects of the project.

Contents

Abstract i

Management Summary ii

Acknowledgements iv

1 Introduction 3
- 1.1 Current State . . . 3
- 1.2 Goal . . . 3
- 1.3 Motivation . . . 4
- 1.4 Overview . . . 4

2 Design 5
- 2.1 Technical Description . . . 7
  - 2.1.1 Complete Sequence Diagram of a DWRTC session . . . 9
- 2.2 Architecture . . . 10
  - 2.2.1 Layers . . . 10
  - 2.2.2 Libraries . . . 11
- 2.3 Design Decisions . . . 12
  - 2.3.1 Signaling Layer . . . 12
  - 2.3.2 WebSocket Layer . . . 14
- 2.4 Deployment . . . 16
  - 2.4.1 Docker . . . 16
  - 2.4.2 Internet . . . 16
- 2.5 Further evaluation: NAT workaround techniques . . . 18
  - 2.5.1 ICE . . . 18
  - 2.5.2 Decentralized TURN . . . 19
  - 2.5.3 Distributing TURN Servers . . . 20
  - 2.5.4 Improving Latency to TURN Servers . . . 20

3 Conclusion 22

3.1 Outlook 22

4 Appendix **25**
4.1 Organization 26
4.1.1 People 26
4.1.2 Meetings 26
4.1.3 Code Repository 26
4.1.4 Code Documentation 26
4.1.5 Task Handling 26
4.1.6 Time Tracking 27
4.1.7 Methodology 28
4.1.8 Quality measures 28
4.1.9 Tools 29
4.2 Requirements 30
4.2.1 Use Cases 30
4.2.2 Non-Functional Requirements 31
4.2.3 Result 32
4.3 Design Diagrams 33
4.3.1 Package Diagram 33
4.3.2 Class Diagrams 34
4.4 Project Plan 35
4.5 Risks 36

References **38**

List of Figures **40**

List of Tables **41**

Glossary **43**

Chapter 1

Introduction

1.1 Current State

In today's Internet, most of the available services are centralized. Different solutions are introduced to allow those services to scale as they do not have this ability built-in. Decentralized applications allow for easier scalability.

WebRTC uses peer to peer connections out of the box. This requires a connection setup (also known as signaling or bootstrapping) which is predominantly centralized. We propose fully decentralizing WebRTC by distributing the bootstrapping process.

Boldt et. al propose using DNS and Master Peers for this idea,[1] while Knoll et al. suggest Internet Relay Chat (IRC) among other concepts. [2]

1.2 Goal

This term projects implements WebRTC signaling using a P2P network employing a distributed hash table (DHT) which stores routing information. User agents connect to one of the servers in this network. These servers relay the bootstrapping information between their clients.

To proof this concept, a video calling app is also implemented.

1.3 Motivation

Peer to peer applications are explicitly harder to manage than centralized applications as they follow the principle of eventual consistency instead of the ACID principle [3].

1.4 Overview

Chapter 2 includes the design of the implementation of DWRTC. Chapter 3 adds information about what was achieved and how it can be extended. Chapter 4 concludes with detailing the organization and requirements of this term project.

Chapter 2

Design

WebRTC enables AV and chat communication without using additional plugins [4]. All major browsers support the basic functionality of WebRTC. [5]

WebRTC allows web developers to integrate AV and data sending capabilities into their own websites. The developer only needs to implement the exchange of signaling messages [6]. Once all information has been relayed, the browsers will connect either via a P2P connection, or various levels of relays.[7] WebRTC reduces the minimal requirements to use the application. Only a web browser supporting WebRTC is required.

We use TomP2P to implement the signaling channel which enables WebRTC to be completely distributed.

Architectural Idea Both users use a supported web browser to connect to a website which will communicate with the server and later initiate the WebRTC connection. In this example, User A connects to Server X and User B to Server Y.

User A starts a new session. An ID to identify the session is displayed. User A passes the ID to User B via an external channel (e.g. an Instant Messenger).

User B sends signaling information to User A. To achieve this, Server Y will lookup the responsible node (Server Y) and initiate the connection.

It is irrelevant which server User B contacts, provided that the server is connected to the distributed network.

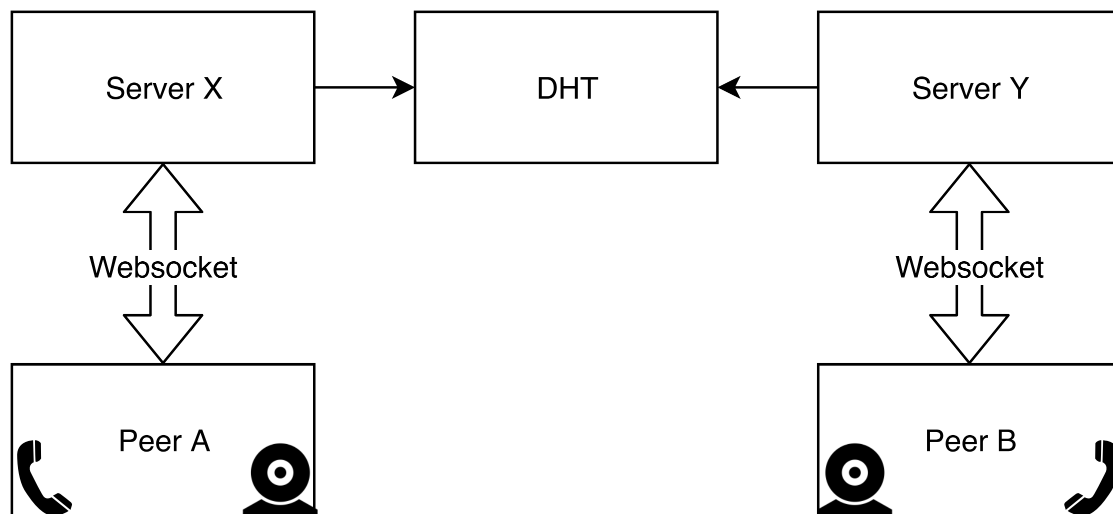

Figure 2.1: Architecture (Simplified)

2.1 Technical Description

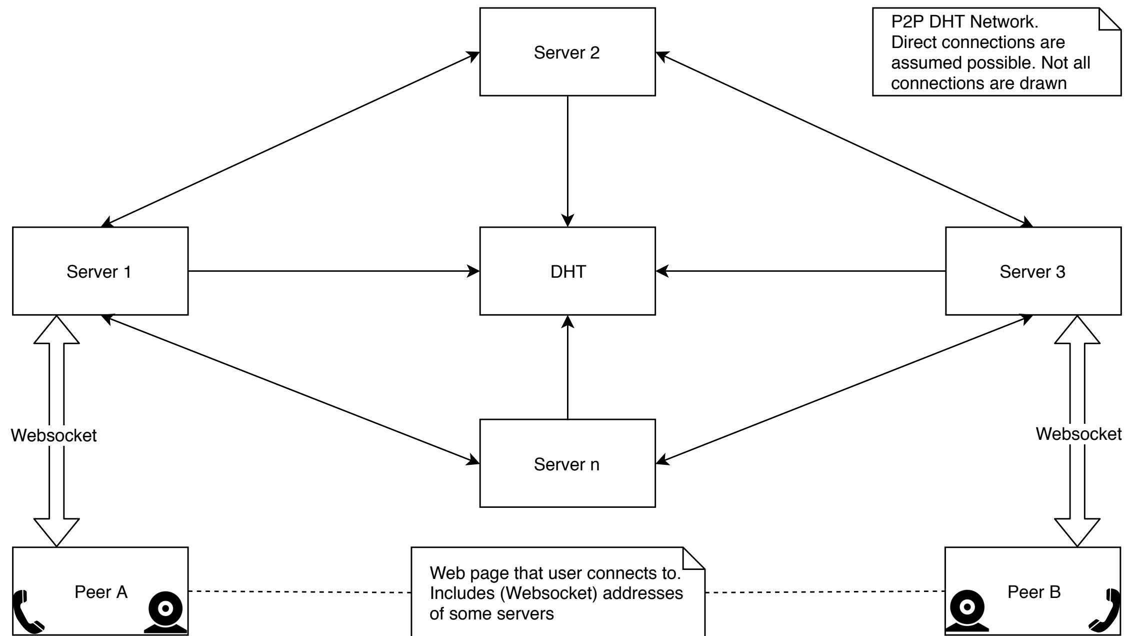

Figure 2.2: Architecture (Big Picture)

Web Application The application's architecture allows developers to build their own app using DWRTC. As a part of this project, we built a video calling demo app.

WebSocket Connection When a user starts a new session, the web page will open a WebSocket connection to one of the servers. This WebSocket connection is used to exchange signaling information between the web page and the server.

Session ID The session ID is a unique, generated ID (by the web framework), that is used for this specific session. The user passes the ID via an external channel to their partner.

The partner enters the ID to initiate the connection.

Server Application The server application has two interfaces:

- a WebSocket interface enabling communication with the browser
- an interface to the P2P layer

The server application is running permanently. If another TomP2P peer is known on startup, it will bootstrap to this peer's DHT.

DHT The DHT is established permanently. DWRTC uses the DHT for mapping Session IDs to their responsible server.

Each server registers itself in the DHT under all its users' keys.

Connecting and Sending a Message to Another Server Alice (Peer A) and Bob (Peer B) are trying to communicate with each other using DWRTC. The connection setup is best explained by the text below and Figure 2.3 Sequence Diagram of a DWRTC session.

Alice's server receives a signaling message for Bob. To establish the connection Alice's server will obtain the address of Bob's server from the DHT. Alice's server opens a direct connection to Bob's server and sends the message.

Relaying Messages Received from Another Server Bob's server receives a message addressed to Bob. Bob's server relays the message to Bob's browser via the WebSocket connection. Bob's web browser processes the message accordingly.

Message Types The following message types exist:

- Client messages (e.g. signaling).
- ID messages: Used to announce the session ID to the client (e.g. video call application).
- Error: Used for errors by the backend.

2.1.1 Complete Sequence Diagram of a DWRTC session

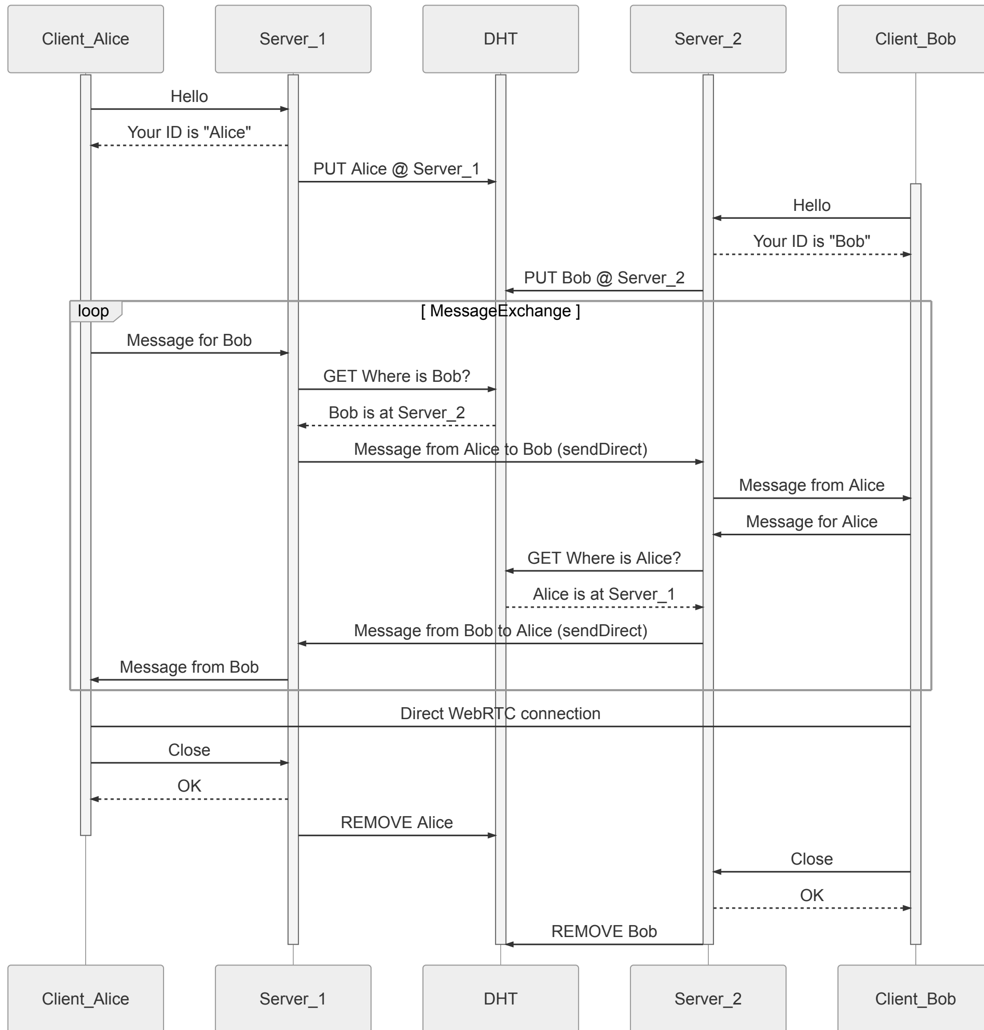

Figure 2.3: Sequence Diagram of a DWRTC session

The DHT is maintained by TomP2P. It is formed when servers bootstrap to each other. TomP2P manages servers joining and leaving the DHT.

2.2 Architecture

2.2.1 Layers

We decided to split DWRTC into two layers.

- `websocket` API layer
- `signaling` P2P layer

They are based upon the third-party `tomp2p` DHT layer. The layers can be compared to the OSI model's TCP/IP model, since lower ones are not concerned with the details of upper ones.

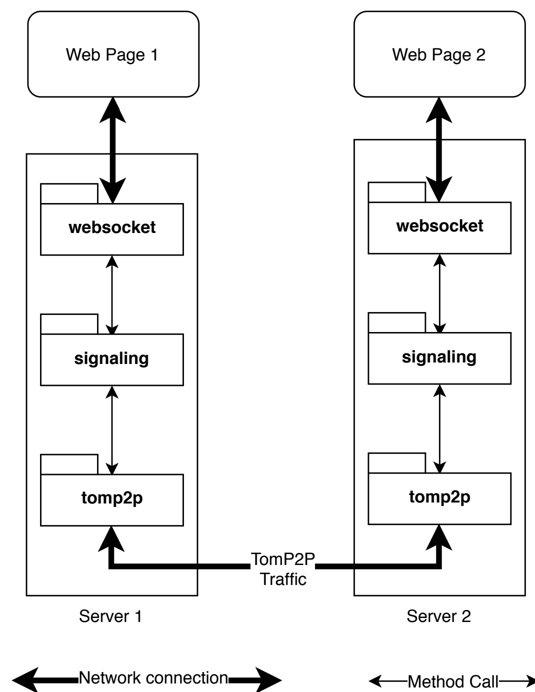

Figure 2.4: Flow through the layers

The layers are placed on top of TomP2P.

A request goes from the web browser to the high-level `websocket` layer through the low-level `signaling` layer.

TomP2P's routing abilities are used to route the messages to the other peer.

2.2.2 Libraries

- The base layer is TomP2P (`https://tomp2p.net/`).
- The `websocket` layer uses the Javalin web framework (`https://javalin.io`). Jackson is used for JSON serialization (`https://github.com/FasterXML/jackson`).
- Configuration is read via Konfig (`https://github.com/npryce/konfig`).
- Log messages use kotlin-logging (`https://github.com/MicroUtils/kotlin-logging`).
- Tests are run via kotlintest (`https://github.com/kotlintest/kotlintest`).
- http4k's WebSocket module is used for E2E tests (`https://www.http4k.org/`) since Javalin offers no WebSocket client.

2.3 Design Decisions

2.3.1 Signaling Layer

sendDirect instead of DHT

DWRTC uses sendDirect messages instead of using the DHT. This adds complexity, since messages need to be sent to the correct address (see below)

Using the DHT would therefore be easier. However, it is not possible for new DHT data to cause a trigger to run. Therefore, polling would need to be used and performance would decrease.

Finding addresses

The DHT is a map of Session ID to server network address. Especially in complex setups, there are many network addresses.

When a session is started, the server discovers its external addresses by querying well-known peers with which address they can see this peer (the bootstrapping peers are reused as well-known peers). The first of these addresses is added into the DHT and is used for sending messages later.

When a message needs to be routed to this server, only this first discovered address is used. Initially, it was planned to add all discovered addresses into the DHT. When sending, all addresses would be tried until one would signal a successful response. However, this proved difficult due to deadlock issues on the network in Jetty, that we were not able to pinpoint exactly.

Therefore, the approach above was chosen. With this method, there is at least one confirmed address in the DHT.

User Input

DWRTC is not focused on security (see section 4.2.2.1 Security).

The `ClientMessage.senderSessionId` ID is overwritten before a message is sent to the TomP2P layer. This disallows a user from deceiving their sender ID.

`ClientService.removeClient` only accepts an `InternalClient` (that was created using `ClientService.addClient`), so one cannot disconnect another user via the Kotlin API.

Message Format

The `Message` class uses a `type` as its discriminator. The availability of other fields depends on the `type`. This allows for a very flexible format that also ensures type-safe casting.

Developers using DWRTC can define their own message types with the payload residing in the message body (see API documentation at `https://docs.dwrtc.net`).

Bootstrapping

Bootstrapping is the act of joining an established P2P network.

The `ClientService` class supports two bootstrapping mechanisms:

1. Bootstrapping with a given TomP2P `PeerAddress`.
   The `PeerAddress` bootstrap mechanism is meant for tests, where the peer's address is already available in the correct, technical format.
2. Bootstrapping using a normal IP/port pair (using `PeerConnectionDetails`).
   The IP/port pair bootstrap mechanism is meant for user input.

ClientService

The `ClientService` class is the one-stop starting point for all P2P/DHT operations. It is a service object that creates and bootstraps the TomP2P peer. All objects are created through methods of this object and the TomP2P peer is shared with these objects.

Futures

A `Future` is a proxy object allowing for asynchronous completion of an operation. Operations that run on completion (callback functions) can be added.

The `signaling` layer contains its own `Future`s. These are meant to abstract the TomP2P `Future`s.

Extension classes for TomP2P are available in the `util` layer. They are mostly used by these `Future`s to allow the usage of Kotlin specific lambda expressions or anonymous functions [8].

InternalClient/ExternalClient

An `InternalClient` is created when a new WebSocket session is started. It is possible to send and receive messages.

An `ExternalClient` is created when an `InternalClient` wants to send messages to it. An `ExternalClient` can only receive messages. On the other peer, the messages are then routed to a corresponding `InternalClient`.

Note: an `ExternalClient` *can* be on the same server.

Message Routing

When an `InternalClient` is created, the `ClientService` registers its session ID in a message dispatcher table. The dispatcher then sends all the received messages to the correct `InternalClient`

2.3.2 WebSocket Layer

Message Format

The `Message` format is reused for WebSocket specific messages (`WebSocketErrorMessage`, `WebSocketIdMessage`) and application-specific messages (`ClientMessage`)

WebSocket Handler

The `WebSocketHandler` consists of four main components:

- `WebSocketHandler.clients` is a map of session ID to `InternalClients`.
- `WebSocketHandler.sessions` is a map of session ID to WebSocket sessions.
- `WebSocketHandler.onReceiveMessageFromWebSocket` uses the session ID to get the `InternalClient`. This is then used to send a message through the P2P layer.
- `WebSocketHandler.onReceiveMessageFromSignaling` uses the session ID to get the WebSocket session. This is then used to send the message to the specific WebSocket.

IDs

The WebSocket session IDs are reused for the user's session ID in the DHT. The ID is assumed to be unique.

2.4 Deployment

2.4.1 Docker

For simple deployments, DWRTC production builds are created as Docker images. The built image is available on Docker Hub (`https://hub.docker.com/r/dwrtc/dwrtc/`) and the Dockerfile is included in the source code repository.

A developer building an application using DWRTC can test it using Docker Compose. The provided Docker Compose file deploys two containers that are bootstrapped to each other. The two containers both expose their web servers, consisting of the WebSocket and the demo app. They also expose the TomP2P port, allowing a developer to bootstrap their extensions to DWRTC to a working network.

2.4.2 Internet

DWRTC is also publicly available on the Internet. This is intended for users to try out the demo app and for developers to experiment with the API.

The Internet deployment consists of multiple nodes. All nodes run DWRTC and a coturn instance (TURN server). The web server is accessed through traefik (`https://traefik.io/`) as a reverse proxy, which also provides automatic TLS certificate deployment using Let's Encrypt (`https://letsencrypt.org/`)

The installation is automated using Terraform and Ansible with the following advantages:

- easy deployment (two commands)
- scaling: more nodes require only a few changes
- consistency: all nodes are set up exactly the same way
- recoverability: the setup can be recreated should it fail

Terraform Terraform (`www.terraform.io`) enables infrastructure to be expressed as code.[9] For DWRTC, this consists of two servers on an IaaS service with the author's SSH keys deployed.

Ansible Ansible (`www.ansible.com`) is a software that automates software provisioning, configuration management, and application deployment. [10] It is used to install and start the necessary software and to maintain the base system.

2.5 Further evaluation: NAT workaround techniques

To achieve fully decentralized WebRTC the following parts have to be implemented:

- Signaling
- TURN

The next section describes ICE (STUN/TURN) in the context of WebRTC.

2.5.1 ICE

The Interactive Connectivity Establishment (ICE) framework allows a web browser to connect to other web browsers. If possible it uses a direct P2P connection. It employs STUN to determine what types of NAT are employed. If direct connections are impossible, the connection is relayed over TURN servers. [7]

ICE candidates are sent alongside the SDP Offers/Answers. The candidates are established by the ICE protocol. One of the peers acts as a *controlling agent* and is responsible for choosing the preferred ICE candidate pair. [11]

STUN Session Traversal Utilities for NAT (STUN) determine what connection restrictions are present between two peers. STUN sends a request to a known STUN server residing on the Internet. The STUN server responds with the client's public IP address. The client announces this address to the partner's peer. [7]

The client may request further information from the STUN server to determine the type of NAT. [7]

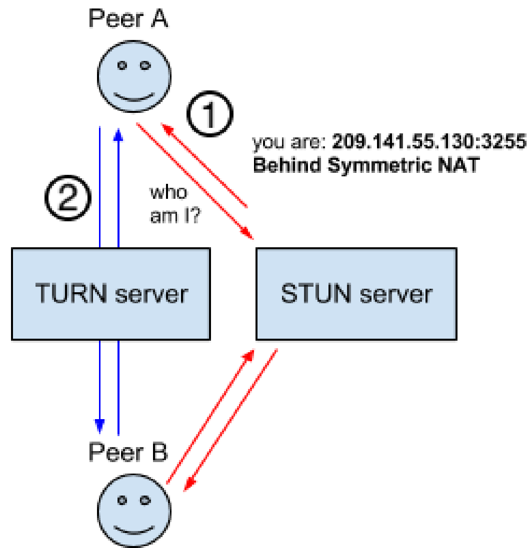

Figure 2.5: STUN and TURN servers. [12]

TURN The TURN protocol is used to exchange messages behind a *Symmetric NAT*. [11]

1. If the STUN protocol discovers that the client is behind a *Symmetric NAT*, the Traversal using Relays around NAT (TURN) protocol is initiated. The TURN protocol uses its own TURN server.[7]

2. To a firewall, a connection to a TURN server looks like a normal client/server connection. The TURN server acts as a relay. It receives a message from Alice, determines that it is for Bob and sends it to Bob (and vice-versa).[7]

2.5.2 Decentralized TURN

The idea is to distribute TURN servers on the Internet. Alice wants to connect to Bob. Alice connects to TURN server 1, Bob to TURN server 2. TURN server 1 determines via the DHT that Bob is connected to TURN server 2. TURN server 1 sends all streaming messages to TURN server 2.

Performance This approach leads to a decline in performance. Stream messages require an unnecessary round trip between TURN servers before they are sent to the user.

In the case of DWRTC, signaling messages are delayed too, but delays in the signaling process are not obvious to the user since signaling occurs before a call has started. The exchanged data consists of a few small text messages, whereas relaying streaming information entails continuously sending large pieces of data.

Delays in streaming messages are more pronounced. If the latency is higher than 250ms RTT, users would start to notice[13].

Decentralized TURN is therefore not feasible, the main argument being the added latency of two hops.

2.5.3 Distributing TURN Servers

It was decided to distribute an existing TURN server implementation alongside the deployed DWRTC nodes on the Internet (see section 2.4 Deployment). The TURN servers are not part of a normal DWRTC installation but specific to our deployment.

The decision had been made to use coturn as a STUN/TURN server (`https://github.com/coturn/coturn`). coturn evolved from the original RFC 5766 project.[14] Judging from the README, it implements the most RFCs and features.

The video call app is configured to use these TURN servers. A developer may use these TURN servers or can configure their own.

The TURN servers are secured by static (so-called *long-term*) credentials. This is because browsers do not accept unauthenticated TURN.[15]. It also protects against DDoS attacks where attackers find these servers by scanning the Internet, since they would not know the credentials. However, if an attacker finds the DWRTC credentials through examining the source code, they would be able to attack these servers.

This ensures that users of the video call app can communicate even if they reside behind symmetric NATs.

2.5.4 Improving Latency to TURN Servers

Our supervisor, T. Bocek asked, if it is possible to favor overall latency when a browser chooses a TURN server. Overall latency is defined as the total latency of all peers to a TURN server.

If multiple TURN servers are specified, the browser must choose a priority for each server. It is not specified how this priority is chosen, but it is *recommended* to use factors such as latency or packet loss. [16] However, there are no APIs to influence the browser's priority decision. A circumvention of this limitation is to offer only one server to the ICE layer.

Choosing Lowest Overall Latency We propose an algorithm that allows choosing one server with the lowest overall latency.

1. The *controlling agent* sends a message to all peers containing the TURN servers that need a latency evaluation.
2. All peers measure the latency to each TURN server.
3. All peers send their latency measures to the controlling agent.
4. The controlling agent totals the received latencies for one TURN server.
5. The server with the overall lowest latency is chosen.
6. This server is submitted to the ICE layer.
7. The ICE layer includes this as its only TURN server in its candidate list.

The latency measures may use trickling, i.e. results are sent back to the controlling agent as they become available [17]. The controlling agent may enforce timeouts.

This improves the user experience in certain scenarios. For example, Alice (controlling agent) resides in the USA and Bob is located in the UK. By measuring only her latency, Alice may choose a TURN server in Canada, even though a TURN server in Iceland is available. The Icelandic TURN server would yield better overall latency to Alice and Bob than the Canadian TURN server, as it resides on the geographical route between the two peers.

Implementation To measure latency to a server in JavaScript, a recommended approach [18] is to measure the load time of an image element. This method requires a valid picture residing on the server (e.g. 1x1 pixel) and can therefore not be used on arbitrary servers. To ensure the skipping of any caches, a query string with a random value is added.

Chapter 3

Conclusion

A working example of a distributed WebRTC signaling channel was implemented. A video call client written in JavaScript, using the Kotlin backend was built.

Kotlin was an excellent choice to write clear and expressive code despite the fact that the authors had not worked with it a lot before starting the project. Implementing our solution was not an easy task, especially due to the amount of asynchronicity, difficulty in debugging and our lack of experience with TomP2P.

It was not possible to implement all of the ideas. Nevertheless, the implemented result is a working video call application on a basis which could be easily used for other purposes such as P2P file sharing over WebRTC.

3.1 Outlook

Message Authenticity

Messages on the P2P or the WebSocket layer are not checked for authenticity. The messages could be signed on the TomP2P layer using public key cryptography [19] [20] or on the WebSocket layer using the Web Crypto API [21]. Using public key cryptography, the messages could be signed. This would require an initial untampered connection to exchange trusted public keys. The authenticity of a message could then be proven.

The supposed attack is as follows: Alice and Bob are communicating with each other. They are joined by Mallory, who wants to send malicious messages to Alice, posing as Bob. Currently, Mallory could join the P2P network, find Alice's server

and send signaling messages as Bob. She could also connect to the WebSocket of any participating node and send a message posing as Bob.

This was not implemented since security is not a requirement for DWRTC (see section 4.2.2.1 Security)

Cache in `ClientService.findClient`

In the current state, each new message triggers a new `findClient` call. As there are several messages in close succession, a time-based cache could be added. Currently, the DHT is queried each time.

This could not be implemented because `findClient` needs to return a Future. The current implementation of Future does not allow for returning immediate values.

Dashboard for Server Operator

This dashboard would be accessible over the webserver. It would present statistics about the current node, e.g. connected user sessions or bootstrapped peers, along with system statistics such as CPU or RAM usage.

The dashboard idea was disregarded since it would not add much value.

Refactoring Web Layer

Javalin is the currently used web framework. Since http4k's WebSocket module is already used in tests, switching the `websocket` layer to http4k would remove the dependency on Javalin.

Changing the web framework would not add value and it would require time to become familiar with the new product. Therefore we decided to not replace the framework.

Refactorings of Video Call Application

The demo app currently has no tests. This idea would entail adding unit tests and E2E tests.

Testing of the WebSocket layer and downwards is already done in Kotlin. Adding tests for the JS part would not add a much of a benefit compared to the required time to implement it.

Basic Messaging JavaScript Client

DWRTC could be used as a distributed messaging app.

This would also help with debugging, since one is not dependent on a correct WebRTC implementation.

This idea was omitted since debugging was possible using the browser's log console and it would not add much value.

P2P File Sharing

WebRTC allows sending arbitrary data via the data channel. [22] This would allow for a P2P File Sharing application.

A P2P File Sharing application was not developed since it can be done by any other developer using DWRTC. It would not add much value to the core project, since video calling is available.

Testing

The base layer of DWRTC is TomP2P. Therefore, it is tricky to create unit tests without mocking TomP2P, which in itself is a tough task.

Most of the tests can be considered integration tests or even E2E tests. It is difficult to set these up properly, and it has proven to be challenging to debug test failures.

Therefore, DWRTC is not as properly tested as it could be and future work could add additional and more thorough tests.

Chapter 4

Appendix

4.1 Organization

4.1.1 People

- Prof. Dr. Thomas Bocek, supervisor
- Demian Thoma, student
- Lukas Röllin, student

4.1.2 Meetings

Meetings are held in T. Bocek's office every Wednesday at 13:00h. The main goal of these meetings is to address open question, discuss what has been done within the last week and to define goals for the coming weeks. These discussions include the state and direction of the project.

4.1.3 Code Repository

The code is tracked using Git. DWRTC is open source and the code is accessible on Github (`https://github.com/dwrtc/dwrtc`).

4.1.4 Code Documentation

The code documentation is available on `https://docs.dwrtc.net/dwrtc/`.

4.1.5 Task Handling

GitHub is used for issue/task tracking. The issues are available at

- `https://github.com/dwrtc/sa/issues` for high-level issues
- `https://github.com/dwrtc/dwrtc/issues` for code-level issues

All issues can be viewed in the project dashboard: `https://github.com/orgs/dwrtc/projects/1`

This led to 56 high-level issues and 56 code-level issues.

4.1.6 Time Tracking

Each student is expected to spend a maximum of 240 hours on this project. This results in a weekly workload of up to 17 hours.

The students are required to track their working hours. We decided to use Clockify (`clockify.me`).

The work is categorized into the following sections:

- Implementation
- Meetings
- Project Management
- Research
- Writing

This resulted in a total work time of 422 hours, which is 88% of the total maximum of 480h.

The time was spent as follows (numbers are rounded)

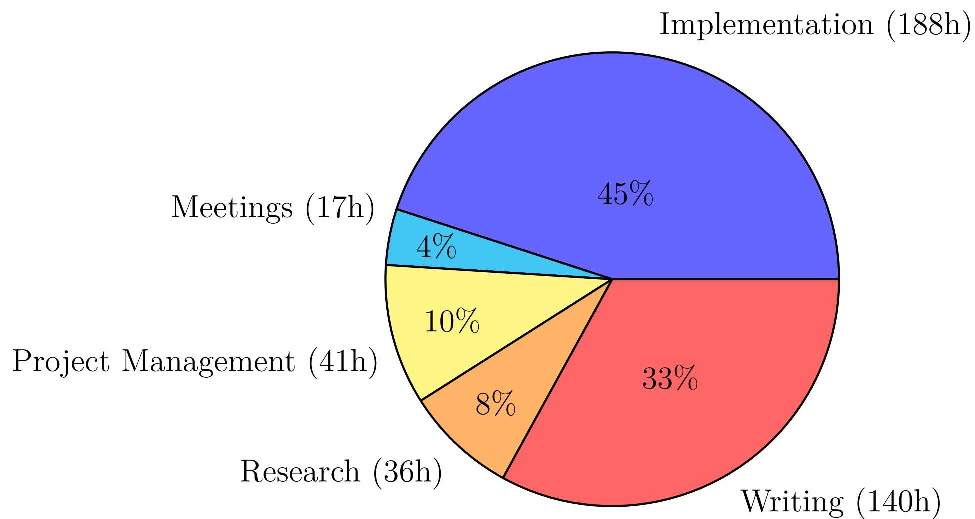

Figure 4.1: Time spent by category (total of 422)

4.1.7 Methodology

For this project, an agile approach without sprints is practiced.

Each week is assigned to one or more project phases. Tasks are assigned to these phases.

The `master` branch represents the current working state. A CI service ensures a clean, running state.

4.1.8 Quality measures

Testing Unit Testing and Integration Testing is employed. Before a pull request is merged, it is required that all tests succeed.

Code Style The code adheres to the following code styles:

- Kotlin: Google Android Style Guide `https://developer.android.com/kotlin/style-guide`
- JavaScript: Prettier `https://prettier.io/`
- HTML/CSS: Prettier

CI The CI is used to build the application, run tests and run code analysis tools.

Linting SonarCloud's Continuous Inspection product SonarQube is applied on each CI run to ensure an idiomatic code base. SonarQube comments on the applicable code parts and suggests improvements.

Reviews A pull request is required to change the code base. Direct modifications of the `master` branch are forbidden.

A pull request requires a review by another contributor to be merged. This should promote an understandable code base (see section 4.5 R1: the students do not understand parts another student wrote) and clear assumptions (e.g. nullability). This resulted in 73 pull requests.

The text of this term project was treated in the same way, resulting in 38 pull requests.

Open Source Projects The following bugs were found and reported:

- simple-peer: `.on("signal")` fires even though `initiator: false` (version 9.1.0 and higher) `https://github.com/feross/simple-peer/issues/366`
- Javalin: Sending to WebSocket from multiple threads crashes Jetty. Maybe synchronize? `https://github.com/tipsy/javalin/issues/423`

The following improvements to projects were made:

- traefik: Allow usersFile comments `https://github.com/containous/traefik/pull/4159`

4.1.9 Tools

The following development tools are used:

- JetBrains IntelliJ IDEA for Kotlin development (`https://www.jetbrains.com/idea/`).
- Microsoft Visual Studio Code for web development (`https://code.visualstudio.com/`).
- Docker as a runtime environment (`https://www.docker.com/`).
- Gradle as a build tool and for dependency management (`https://gradle.org/`).
- Dokka for the code documentation (`https://github.com/Kotlin/dokka`).
- Prettier for code formatting (web) (`https://prettier.io/`).

4.2 Requirements

4.2.1 Use Cases

DWRTC:

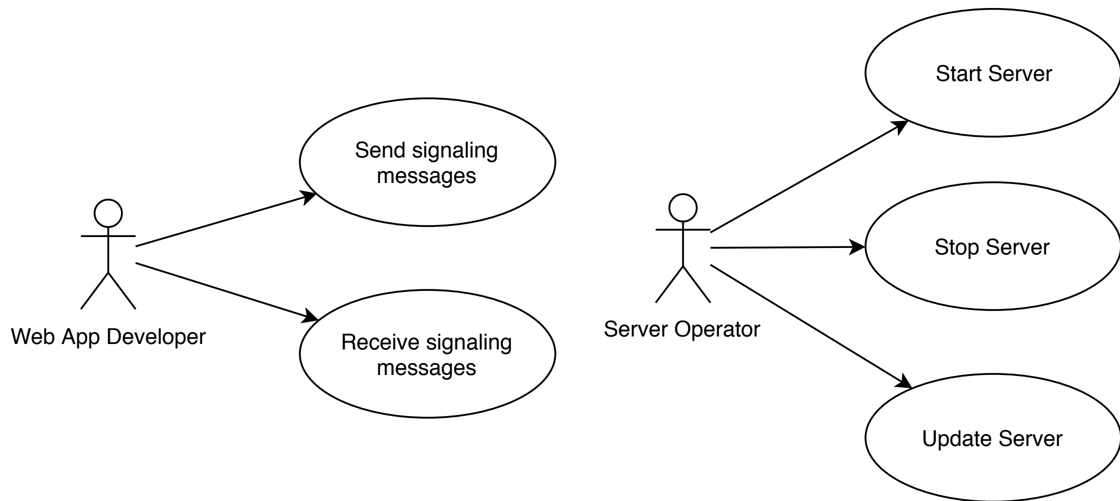

Figure 4.2: Use Case Diagramm of DWRTC

Demo App:

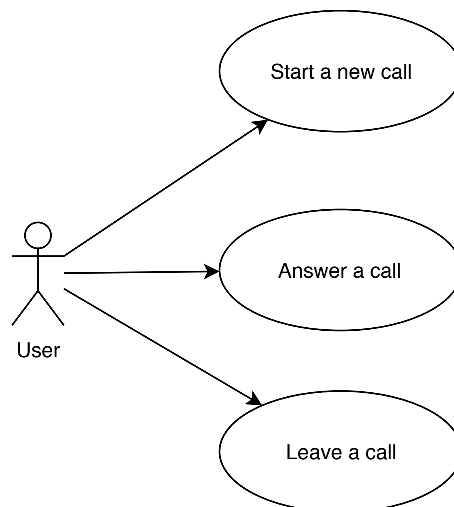

Figure 4.3: Use Case Diagramm of Demo App

4.2.2 Non-Functional Requirements

Based on [23], [24], [25] & [26].

4.2.2.1 Functionality

Accuracy The system does not forward calls to the wrong party.

Interoperabilityy The system works with web browsers that support WebSocket. The P2P layer can only communicate with other peers running DWRTC.

Security This research project is not focused on security.

4.2.2.2 Reliability

Maturity This research project sets no hard requirements on stability. However, measures to maintain the stability are put in place (see subsection 4.1.8 Quality measures).

Fault Tolerance Already established calls are not disconnected by a faulting server.

4.2.2.3 Usability

Understandability The UI is understandable without any expertise. Any user who once made a video call from his computer understands the process.

Learnability A user who has already set up audio/video is able to use the product after 5 minutes without further instructions.

Operability The system logs information about its state.

The user is always informed about what is currently going on and what options are available.

4.2.2.4 Efficiency

Time behavior Once a call is started, it is answerable (WebRTC starts connecting directly) within 30 seconds.

4.2.2.5 Maintainability

Analysability The software keeps a log file on each server on what it is currently doing.

Changeability The software allows changes without unforeseen side effects.

Stability The software employs automated testing. This ensures that unintended changes are discovered in an early stage.

Testability The software architecture allows automated testing.

4.2.2.6 Portability

Adaptability The software runs on any platform supporting the Java Virtual Machine (JVM).

Installability The software is installable in less than 30 minutes.

Replaceability The software does not need to be easily replaceable.

4.2.3 Result

Most requirements were fulfilled. An exception are the usability requirements since no usability tests were executed. Another exception is the testability, the reasons are discussed in section 3.1 Testing

4.3 Design Diagrams

4.3.1 Package Diagram

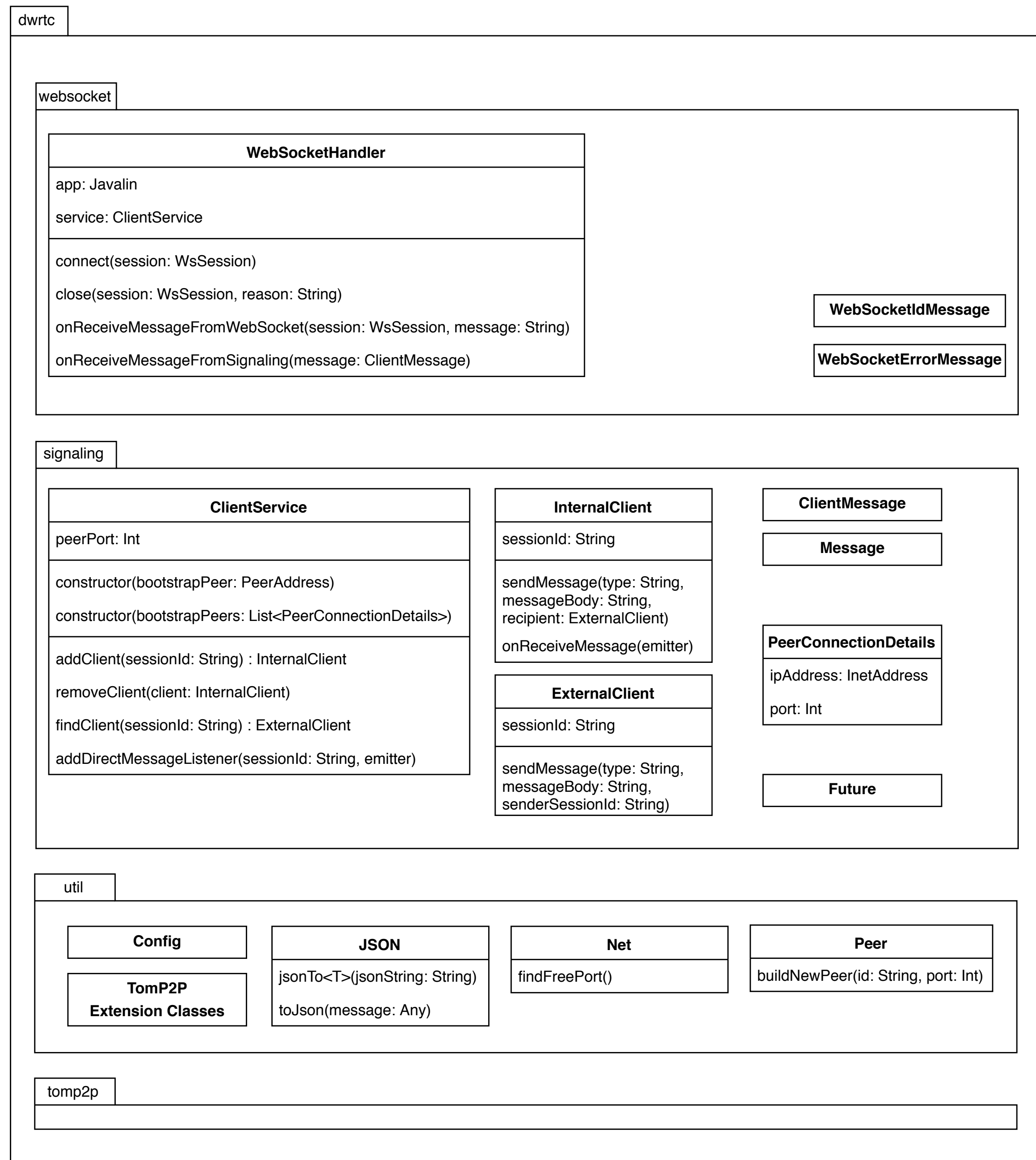

Figure 4.4: Complete Package Diagram

4.3.2 Class Diagrams

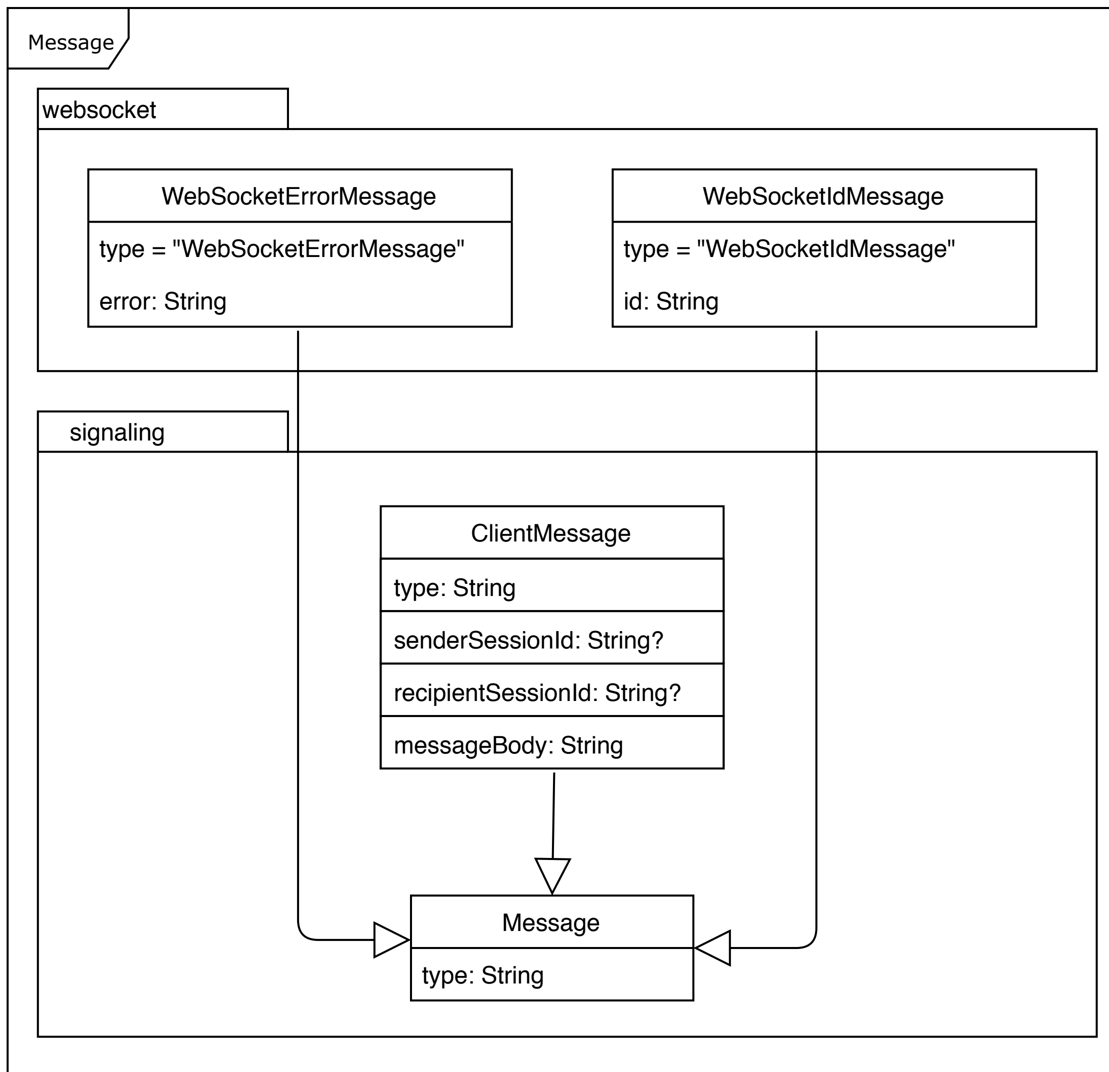

Figure 4.5: Class Diagram for Message class

4.4 Project Plan

Term Week	Start Date	Phase(s)
1	17.09	Kickoff
2	24.09	Elaboration
3	1.10	Elaboration
4	8.10	Elaboration, Implementation
5	15.10	Elaboration, Implementation
6	22.10	Implementation
7	29.10	Implementation
8	5.11	Implementation
9	12.11	Implementation
10	19.11	Implementation
11	26.11	Implementation, QA
12	3.12	Implementation, QA
13	10.12	QA
14	17.12	QA

Table 4.1: Project Plan

The Project Plan was followed very closely and there have been no alterations during the term project.

4.5 Risks

The following section lists the identified risks and their corresponding countermeasures.

R1: the students do not understand parts another student wrote

- code reviews
- explain ideas and tasks
- consider multiple ways of an implementation
- challenge decisions

R2: too much work, effort was underestimated

- review time estimation of tasks with supervisor
- revise uncertain tasks

R3: misunderstanding of requirements

- document requirements
- review by supervisor

R4: problems with frameworks or tools

- familiarize with frameworks and tools
- use maintained frameworks and libraries

R5: wrong prioritization of tasks

- be aware of unknown dependencies
- keep an eye on the project plan

Likelihood \ Consequence	Low	Medium	High
Rare		R1	R2
Possible		R4	R3, R5
Almost certain			

Table 4.2: Risk Table

References

[1] Dennis Boldt and Kaminski Felix and Stefan Fischer. "Decentralized Bootstrapping for WebRTC-Based P2P Networks". In: *The Fifth International Conference on Building and Exploring Web Based Environments (WEB2017)*. WEB2017, The Fifth International Conference on Building and Exploring Web Based Environments. May 21, 2017, pp. 17–23.

[2] M. Knoll et al. "Decentralized Bootstrapping in Pervasive Applications". In: *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07)*. Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07). Mar. 2007, pp. 589–592. DOI: 10.1109/PERCOMW.2007.36.

[3] Thomas Bocek. *TomP2P, a P2P-Based Key-Value Pair Storage Library*. URL: https://tomp2p.net/doc/p2p/ (visited on 12/18/2018).

[4] WebRTC Project Authors. *WebRTC Home | WebRTC*. URL: https://webrtc.org/ (visited on 10/17/2018).

[5] Can I use Contributors. *Can I Use... Support Tables for HTML5, CSS3, Etc.* URL: https://caniuse.com/#feat=rtcpeerconnection (visited on 12/05/2018).

[6] Mozilla Contributors. *Signaling and Video Calling*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling (visited on 10/17/2018).

[7] Mozilla Contributors. *Introduction to WebRTC Protocols*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols (visited on 10/31/2018).

[8] Kotlin Contributors. *Higher-Order Functions and Lambdas - Kotlin Programming Language*. URL: https://kotlinlang.org/docs/reference/lambdas.html (visited on 12/18/2018).

[9] HashiCorp. *Terraform by HashiCorp*. URL: https://www.terraform.io/index.html (visited on 11/25/2018).

[10] Ansible, Red Hat. *How Ansible Works | Ansible.Com*. URL: https://www.ansible.com/overview/how-ansible-works (visited on 11/25/2018).

[11] Mozilla Contributors. *WebRTC Connectivity*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity (visited on 10/31/2018).

[12] Modified, Original by Mozilla Contributors. *TURN Server*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols#TURN (visited on 10/31/2018).

[13] Voip-Info.org Contributors. *Voip QoS*. Sept. 16, 2005. URL: https://www.voip-info.org/qos/ (visited on 10/31/2018).

[14] coturn Contributors. *Coturn TURN Server Project*. coturn, Oct. 31, 2018. URL: https://github.com/coturn/coturn (visited on 10/31/2018).

[15] Gus Hogg-Blake. *Node.Js - Is 'long-Term Credentials' Authentication Mechanism *required* for WebRTC to Work with TURN Servers?* URL: https://stackoverflow.com/questions/26110412/is-long-term-credentials-authentication-mechanism-required-for-webrtc-to-wor (visited on 12/06/2018).

[16] A. Keranen, Ericsson et al. *RFC 8445: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. Section 5.1.2.2. Guidelines for Choosing Type and Local Preferences*. URL: https://tools.ietf.org/html/rfc8445#section-5.1.2.2 (visited on 10/31/2018).

[17] WebRTC Glossary Contributors. *Trickle ICE*. Aug. 28, 2014. URL: https://webrtcglossary.com/trickle-ice/ (visited on 11/15/2018).

[18] StackOverflow Community. Original by Epeli. *Is It Possible to Ping a Server from Javascript?* URL: https://stackoverflow.com/questions/4282151/is-it-possible-to-ping-a-server-from-javascript/11941783 (visited on 11/28/2018).

[19] Thomas Bocek. *PeerCreator*. URL: https://tomp2p-docs.netlify.com/net/tomp2p/connection/peercreator#PeerCreator-net.tomp2p.connection.PeerCreator-net.tomp2p.peers.Number160-java.security.KeyPair- (visited on 11/29/2018).

[20] Manfred Karrer. *Code Snippet for Tomp2p*. URL: https://gist.github.com/ManfredKarrer/987421470568eed2b17b (visited on 11/29/2018).

[21] Mozilla Contributors. *Web Crypto API - Web APIs | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API (visited on 11/29/2018).

[22] Mozilla Contributors. *A Simple RTCDataChannel Sample.* URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Simple_RTCDataChannel_sample (visited on 12/16/2018).

[23] ISO. *ISO/IEC 9126-1:2001 - Software Engineering – Product Quality – Part 1: Quality Model.* URL: https://www.iso.org/standard/22749.html (visited on 10/18/2018).

[24] Mehta Farhad. "Nichtfunktionale Anforderungen". Lecture.

[25] Demian Thoma et al. *Engineering Projekt Teiler: Nichtfunktionale Anforderungen.*

[26] Ian Fleming. *ISO9126 - Software Quality Characteristics.* URL: http://www.sqa.net/iso9126.html (visited on 10/18/2018).

List of Figures

2.1 Architecture (Simplified) 6
2.2 Architecture (Big Picture) 7
2.3 Sequence Diagram of a DWRTC session 9
2.4 Flow through the layers 10
2.5 STUN and TURN servers. [12] 19

4.1 Time spent by category (total of 422) 27
4.2 Use Case Diagramm of DWRTC 30
4.3 Use Case Diagramm of Demo App 30
4.4 Complete Package Diagram 33
4.5 Class Diagram for Message class 34

List of Tables

4.1 Project Plan 35
4.2 Risk Table 37

Glossary

API Application Programming Interface. Surface of an application that enables information exchange..

AV Audio/Video.

CI Continuous Integration. Building and testing each change to a project continuously to ensure an on-going quality and stability..

DDoS Distributed Denial of Service. Attack with the goal of overloading and ultimately shutting down a service..

DHT Distributed Hash Table. Hash table that is distributed over many peers.

DWRTC Distributed WebRTC. This project..

E2E Test End to End test. Test using the complete system..

ICE Interactive Connectivity Establishment. See subsection 2.5.1 ICE.

JSON JavaScript Object Notation. Text format for exchanging data..

NAT Network Address Translation: Mapping layer between two IP networks..

P2P Peer 2 Peer. Connection between two equal clients, contrary to a client/server architecture..

SDP Session Description Protocol. Used in WebRTC to communicate media characteristics..

STUN Session Traversal Utilities for NAT. See section 2.5.1 STUN.

TURN Traversal Relay using NAT. See section 2.5.1 TURN.

UI User interface.

WebRTC Web Real-Time Communication.