A protocol for decentralized video conferencing with WebRTC

Solving the scalability problems of conferencing services for the web

ANDREAS HALLBERG
Abstract

Video conferencing has been a part of many communication platforms over the years. Over the last decades users have moved from dedicated telephony networks to the Internet, and recently to the Web. With the introduction of Web Real-Time Communication (WebRTC) it is now possible to make voice- and video calls simply by visiting a web page, without having to install any additional software. Services that enable multi-user conferences are quite common. However existing solutions such as the Multipoint Control Unit (MCU) inherently do not scale and can be a single point of failure, due to its centralized architecture. This can lead to high maintenance costs and poor service availability.

To solve the scalability- and availability problems of video-conferencing services, a decentralized alternative to the MCU is proposed. A decentralized conferencing system uses the distributed resources of its users instead of relying on a central server. This means that the system can handle an increasing number of users without having to upgrade any server infrastructure. Additionally, failures are only partial and can happen regularly without affecting the rest of the system.

This report presents the development of a protocol built on top of WebRTC that enables completely decentralized multi-user conferencing. It includes a distributed algorithm for voice-activated switching to reduce the computation and network resources used. A load-balancing technique based on media stream relays is used to distribute the resource requirements of the conference participants. The protocol is implemented as a Javascript library that can be included in a web application.

A proof-of-concept web application is developed using the library and its performance is evaluated. The performance data is analyzed and the results are used to make incremental improvements to the protocol and implementation.

Although not all features of the protocol are implemented, the tests show promising results. The application allows multiple users to participate in high-definition video conferences, with no server infrastructure aside from a Mini PC that hosts a web server and a WebRTC signaling server.
Abstract

Videokonferenser har varit en del av många olika kommunikationsplattformar genom åren. Tekniken har flyttats från dedikerade telefonnät, till Internet, och på senare tid till webben. I och med introduktionen av WebRTC (Web Real-Time Communication) är det idag möjligt att enkelt deltaga i röst- och videosamtal genom att gå till en websida utan att behöva installera någon programvara annat än en webbläsare.

De flesta existerande konferenstjänster är byggda med en centraliserad arkitektur, vilket kan leda till tekniska problem när antalet användare ökar eller när fel uppstår i systemens centrala servrar. Dessa problem kan leda till driftstopp och skada tjänstens tillgänglighet för användarna.

Den här rapporten täcker utvecklingen av ett protokoll som tillsammans med WebRTC kan användas för att bygga en helt decentraliserad konferenstjänst. Målet är att tjänsten ska vara oberoende av centrala servrar, och på så vis lösa problemen med skalbarhet och tillgänglighet. Protokollet implementeras i en webbapplikation som testas och utvärderas över flera iterationer för att hitta nya förbättringar.

Testerna visar lovande resultat. Slutsatsen dras det är fullt möjligt att bygga en konferenstjänst på detta sätt, och möjligheter för framtida optimeringar och testfall föreslås.
## Contents

1 Introduction  
  1.1 Background .................................................. 1  
  1.1.1 Defining scalability ....................................... 2  
  1.2 Problem ........................................................ 3  
  1.2.1 Problem Statement .......................................... 3  
  1.3 Purpose .......................................................... 3  
  1.4 Benefits, ethics and sustainability ........................... 4  
  1.4.1 Ethics ......................................................... 4  
  1.4.2 Sustainability ............................................... 4  
  1.5 Methodology ..................................................... 4  
  1.6 Delimitations .................................................. 5  
  1.7 Outline .......................................................... 5  

2 Video Conferencing and Related Technologies ................. 7  
  2.1 Video Conference Optimizations ............................... 7  
  2.1.1 Mixing ....................................................... 7  
  2.1.2 Switching .................................................... 8  
  2.2 Web Real-Time Communication ................................. 9  
  2.2.1 Architecture ................................................ 9  
  2.2.2 Interface for web applications ............................ 10  
  2.2.3 Signaling .................................................... 11  
  2.3 Peer-to-peer networks ......................................... 12  

3 Methodology ....................................................... 13  
  3.1 Philosophical Assumptions .................................... 14  
  3.2 Research Methods ............................................... 14  
  3.3 Research Approaches .......................................... 14  
  3.4 Research Strategies ........................................... 15  
  3.5 Data Collection ................................................ 15  
  3.6 Data Analysis .................................................. 15  

4 Modeling language and notation ................................. 16  
  4.1 Distributed Systems Abstractions ............................ 16  
  4.2 Javascript and JSON ........................................... 17  
  4.3 Graphs .......................................................... 17  

5 Requirements ...................................................... 19  
  5.1 Non-functional requirements .................................. 20  

6 System Architecture .............................................. 22  
  6.1 Interfaces ...................................................... 22  
  6.2 Hardware ....................................................... 24
7 Evaluation Environment

7.1 Test scenario .................................. 26
7.2 Hardware ....................................... 26
7.3 Data collection .................................. 27
7.3.1 Running the test ............................... 27
7.3.2 Resource usage monitoring ................. 29
7.3.3 Conference Quality ........................... 29
7.4 Data analysis ................................... 30

8 Full-mesh ........................................ 31
8.1 Application ..................................... 31
8.2 Model ........................................... 31
8.3 Measurement results ............................. 34
8.4 Problems ....................................... 37
8.4.1 Excessive use of TURN ...................... 38
8.4.2 Signaling race condition ..................... 38

9 Active Speaker switching .......................... 39
9.1 Application ..................................... 39
9.2 Model ........................................... 40
9.3 Results ......................................... 42
9.3.1 Resource Usage ............................... 42
9.3.2 Conference Quality ......................... 45
9.3.3 Recovering from failure ..................... 47
9.4 Problems ....................................... 48
9.4.1 Load imbalance ............................... 48

10 Load balancing with relayed streams .......... 50
10.1 Model .......................................... 50
10.1.1 Agreement .................................. 52
10.1.2 Ordering .................................... 52
10.1.3 Tree structure ............................... 52
10.2 Problems and future work ..................... 52
10.2.1 Recovering from failure .................... 53
10.2.2 Dynamic node utility ....................... 53

11 Conclusion .................................... 54

A Distributed Systems Abstractions ............... 56

B Measurements from full-mesh conference with two participants 58

C Voice-activated switching protocol .............. 60

D Tree relay protocol ................................ 63
1 Introduction

Video conferencing has been a part of many communication platforms over the last 30 years. It is used in homes as well as corporate environments. The technologies used to enable video conferencing are constantly evolving as the requirements on quality, latency, security, price and scale increases. The clients and protocols used are also changing along with the rest of the technological landscape. Users today may have multiple devices, with different software, hardware and network constraints, all of which are expected to interoperate in a seamless way. This is especially important for communication services.

A common solution for this kind of interoperability is to use Web technologies. Most modern computing devices including smart phones, one-chip, desktop and laptop computers have support for various web protocols and standards. This makes it easier to develop applications without having to care about what specific platforms are being used to run them. One of the newest additions to the web technology stack is Web Real-Time Communication (WebRTC)[1]. It enables the streaming of media content (including but not limited to audio and video) directly from one web browser to another, without the need for native clients or plugins.

Although it has not been completely implemented or standardized yet, WebRTC is being used by millions of users through services like Google Hangouts[2], Facebook Messenger[3] and WhatsApp[4]. Adding basic conference capabilities such as one-to-one audio and video communication to a web application today is a relatively simple task, thanks to the standardization of the WebRTC Application Programming Interface (API)[1]. Although setting up point-to-point conferences is trivial, doing the same for multipoint conferences is not.

1.1 Background

A multipoint (MP) conference is a conference with three or more endpoints, or clients. There are a number of technologies and architectures that can be used to enable MP conferencing. One common solution is to use a logically and physically centralized Multipoint Control Unit (MCU). Clients of a conferencing service that want to participate in a MP conference call the MCU, which in turn is responsible for coordinating the call and to generate output streams for each client. The streams, i.e the audio and video sent by each client, can be manipulated and optimized for each receiving endpoint.

One major problem with building a conferencing service using an MCU is scaling. The MCU has to receive, process and send at least one stream for every client, which requires more computing and networking resources as the number of clients grow. Figure 1 (left) depicts a simplified MP conference with three participants and an MCU. Each individual client sends one stream and receives another, while the MCU has to manage $N$ streams for a conference with $N$ participants.
While the capacity of an MCU can be increased by building it with sophisticated hardware and placing it in a high-capacity network such as with an Internet Service Provider (ISP) or a corporate network, it fundamentally does not scale, due to its centralized nature.

The right part of the figure shows a decentralized conference without an MCU. In this scenario, each participant communicates directly with every other participant \((N - 1)\) for a conference with \(N\) participants). The resource requirements for a conference has been distributed among the participants. This particular configuration is known as a **full-mesh** conference[5].

Full-mesh is one of the easiest and perhaps the most intuitive solution to decentralized multi-user conferencing. The video and audio from each participant must somehow reach every other participant, so it makes sense to simply send the media directly. It is also resilient to node- or network failures, since any node can disconnect at any point in time without affecting the service for others. However, this solution suffers from similar scalability problems as the centralized alternative: The number of streams handled by each individual client is almost identical to that of the MCU, while they are unlikely to have the same hardware and networking resources.

There have been successful attempts to handle MP conferencing without a central point of control, and without the scaling issues of full-mesh. One notable example is Skype, which is an application that until 2014 used the distributed resources of its users to enable text, audio and video communication[6]. The Skype protocol allowed clients to connect to each other and relay traffic through so called *supernodes*. Any client with enough resources and an IP-address that was directly reachable on the Internet could become a supernode, which meant that a relatively small part of the service had to be controlled by a central server[7].

### 1.1.1 Defining scalability

The problem of **scaling** or **scalability** is discussed throughout this report. Scalability can mean very different things depending on the context, so it is important to set some definitions. In the case of distributed systems in general, scalability describes how the performance of a system is affected as the number of nodes (processes, clients, servers) increases. In the context of MP conferencing, scalability can either refer to how the **service is affected as the total number of users grow**, or how a **conference is affected as the number of participants grow**. These will be referred to as Service Scalability and Conference Scalability respectively.

Consider the full-mesh conferencing solution presented earlier: In this scenario the participants provide all the resources needed to hold a conference. This means that very little resources...
are required by the service provider, even for a very large number of conferences. This solution is optimized for Service Scalability. But you can also see that the resources needed by the participants of a conference increases exponentially as the number of participants grow, in other words this solution has poor Conference Scalability.

1.2 Problem

Multipoint Control Units require sophisticated hardware and high-bandwidth networks to operate with a large number of users. This makes the development, operation and maintenance very expensive for the operator, and by extension the customers. They suffer from the same problems as all centralized systems; they inherently do not scale, and can be a single point of failure. Because an MCU may need to process the media that is sent between participants, the contents of a conversation cannot be encrypted end to end, which can pose a threat to security and privacy. Finally, an MCU located far away from the users can lead to inefficient use of networking resources. Packets may have to be sent back and forth to an MCU across the globe, even though the participants are on the same local network. This leads to increased latency, decreased media quality, and a poor user experience.

Older decentralized systems such as Skype have been proven to scale well, but they require software to be downloaded, installed and updated by users in order to function. They depend on native clients for all combinations of hardware and operating systems, which can lead to high maintenance costs as the underlying systems change.

It is trivial to build a completely decentralized multipoint conferencing system using WebRTC and a full-mesh overlay network. The full-mesh overlay solves the problem of Service Scalability, but its Conference Scalability is problematic for real-world applications. Since the point-to-point connection and transfer of media is already done by WebRTC, the area that needs further investigation is the overlay network that nodes use to communicate that media. It has to be more sophisticated than full-mesh in order to scale. A protocol must be designed to build that overlay network. Furthermore, the protocol may have to contain other optimization techniques specific to WebRTC video conferencing.

1.2.1 Problem Statement

How can a protocol be designed such that it can be used to build a decentralized conferencing service optimized for both Conference Scalability and Service Scalability, given that WebRTC is used to transfer media from one peer to another?

1.3 Purpose

The purpose of this report is to present the development and evaluation of a distributed protocol for WebRTC video conferencing. The engineering problems and solutions will be documented, and the necessary theoretical background of WebRTC and video conferencing in general will be provided.

The purpose of the protocol itself is to solve the scalability and availability problems of existing conferencing solutions, in a way that is cheap and easy to maintain for a conference service provider.

The project aims to show that it is possible to provide video conferencing services without expensive hardware or a central point of control. The evaluation will determine whether the protocol proposed in this report is more efficient and scalable than the previously described full-mesh and MCU solutions.
1.4 Benefits, ethics and sustainability

This project was done with the supervision and help of a team within Ericsson Research. The result is meant to be used in a web-based communication platform to provide an inexpensive and easily-maintained video conferencing service.

The protocol is mainly developed for the Ericsson Research project, but it could be beneficial to anyone working with the development of video conferencing services. The project should give some insight into how a conferencing service can utilize the distributed resources of its users, instead of relying on centralized server infrastructure. A service deployed this way would decrease the cost for maintenance and deployment, which is beneficial to the service provider and by extension its customers.

1.4.1 Ethics

Decentralized software systems are by definition hard to control and monitor from any single point. This can be problematic in the context of communication services, where service providers can be required by law to intercept the traffic of its users and provide it to law enforcement officials[8]. Furthermore, all communication with WebRTC is cryptographically protected using Secure Real-time Transport Protocol (SRTP) and Datagram Transport Layer Security (DTLS)[9]. The ethical issues of encryption, personal integrity and law enforcement is not discussed in detail in this report. It is however important to note that the software developed in this project makes no attempt to provide tools for legal interception.

1.4.2 Sustainability

Having access to a reliable video conferencing service or remote collaboration tool can be an alternative to traveling, particularly in corporate environments. It has been shown that such services can be used by a company to reduce their energy demand and greenhouse gas emissions, provided that the required equipment is used efficiently[10]. In the case of decentralized conferencing, the users provide most of the resources needed to hold a conference, i.e the networking equipment and client devices. In other words, the resources are added to the system and consumed only as they are needed. There is no need for potentially wasted server infrastructure. This has the added benefit that the service can easily be deployed to people in low-population areas and developing countries.

1.5 Methodology

The project uses a combination of qualitative and quantitative research methods, as defined by Håkansson [11]. The development and implementation of the protocol is done in an iterative process, where it is evaluated using both abstract and quantifiable data in order to identify problems and make improvements.

An abductive research approach is taken, as the protocol is evaluated based on its model and performance measurements gathered in an experimental environment. In other words, the protocol’s utility and properties in a real-world application is entirely theoretical.

The methods used in the project can be categorized as Applied research. It seeks to solve a practical problem with a practical solution, based on research in the area of distributed software design and web communication.
1.6 Delimitations

WebRTC is not completely standardized yet, and not all browser implementations nor documentation are complete. Although browser-specific differences will be discussed, the evaluation and design will be based on the current development version of Google Chrome (version 47).

WebRTC contains a large collection of standards for media encoding, networking, browser implementations and programming interfaces. Explaining it in detail is outside the scope of this report, as the specification alone would be too extensive to be included in a thesis project. Instead, the protocol developed in the project is limited to the WebRTC Browser API implemented in Google Chrome. This has the benefit that the protocol will run on any machine or software stack that can run Chrome, but it also means that the possibilities for optimization is constrained by the browser APIs.

The performance tests will not include every single relevant metric, they are limited to a few such as call setup time, CPU usage, and bandwidth usage. The tests will be done on a cluster of equally capable devices.

The actual costs of deploying and maintaining real conferencing services are rarely available to the public, and varies heavily from one provider to the other. Any assumptions regarding such costs will be rough estimates.

1.7 Outline

Chapter 2 contains the theoretical background used in the project, with focus on three areas in technology: Video Conferencing Optimization, WebRTC and Peer-to-Peer. It introduces two techniques for optimization in video conferencing, namely Media Switching and Mixing. Section 2.2 describes the architecture, protocols and interfaces of WebRTC. Section 2.3 presents the concept of peer-to-peer systems and protocols, and a few examples of peer-to-peer overlay networks that have been used in other applications.

Chapter 3 introduces the research methodologies and methods used during execution of the project. It gives a summary of the paper “Portal of Research Methods and Methodologies for Research Projects and Degree Projects” by Anne Håkansson[11] and describes how the methods are applied.

Chapter 4 presents the tools used to model software in the project. In addition to a short description of Javascript and graphs, it introduces the modeling language used in the book “Introduction to Reliable and Secure Distributed Programming”[12]. The language and notation is useful for describing and analyzing protocols for distributed applications.

The requirement elicitation is done in Chapter 5. While most of the requirements were loosely defined at the start of the project, some of them are refined to work as a benchmark for the software developed in the later chapters.

Based on the requirements, a system architecture is developed in Chapter 6. The components and interfaces of a conferencing system are defined and mapped to the necessary hardware. In summary, the conferencing system consist of four components: A web application, a signaling channel, WebRTC, and the conferencing protocol implemented in a javascript library.

Chapter 7 describes the environment used for the evaluation of the software. The hardware setup, the tools used for running tests, and the collection and analysis of data is described.

Chapter 8 presents the development process and evaluation of the first version of the protocol. This is based on the full-mesh model introduced earlier, and works as a point of reference for the improvements made later on. The measurement results confirm that the full-mesh model has a scalability problem, as they show that the nodes are incapable of streaming high-quality video to each other in a conference with five nodes.
A second iteration is presented in Chapter 9. This solution is still based on the full-mesh model, but it is optimized with Voice Activated Switching which was introduced in Section 2.1.2. The results show significant improvements, as nodes use less computing- and network resources when participating in a conference. It is shown that the switching algorithm and the conferencing protocol works even in the event of node failures. However, there is still a scalability problem related to load-imbalance that causes node resources to be exhausted even in conferences with a small number of participants.

To address this problem, the final version of the protocol is presented in Chapter 10. A mechanism for load-balancing using media relays is added to distribute the resource requirements among the conference participants, and increase the Conference Scalability of the protocol.
2 Video Conferencing and Related Technologies

This chapter introduces the relevant concepts and technologies used in video conferencing, as well as some of the ongoing research in the area. Section 2.1 gives a short summary of two possible optimizations that can be used to reduce the resources needed by a video conferencing service. Section 2.2 is a summary of the protocols and interfaces that make up WebRTC. Because WebRTC is still in the process of being standardized, most of the references and written material on the subject are in the form of Request for Comments (RFC) documents from the Internet Engineering Task Force (IETF) and standardization documents from the World Wide Web Consortium (W3C). While some details in these documents can be changed in the future, they are considered stable enough to provide reliable information about the technology. Finally, Section 2.3 introduces the concept of peer-to-peer systems. It gives some examples of protocols and overlay networks that have been used to build peer-to-peer applications in the past.

The technologies presented in this chapter serves as a theoretical base for the protocol developed in the project.

2.1 Video Conference Optimizations

There are many parts of video conferencing that can be optimized, and many variables for which to optimize. The bandwidth needed for video transfer is reduced through various video codecs, call set-up time can be reduced through better signaling methods, and using conferencing servers with specialized hardware can significantly decrease the resources needed to host a conference. The scope of this report is limited to the higher abstraction layers of video conferencing, so most of these optimizations will be largely ignored. The two most relevant optimizations for multi-point conferencing in particular are mixing and switching.

2.1.1 Mixing

There are a number of techniques for media mixing. For simplicity, mixing will from now on refer to the process of combining multiple video streams into a single output stream (also known as Continuous Presence) as shown in Figure 2:

![Figure 2: Four video frames are inserted into a mixer, and compressed into a single frame.](image)

Mixing video streams in a conference is a way to decrease the bandwidth and transcoding requirements for participants. Figure 3 depicts a two-party conference with a mixer. Alice and Bob send their individual video streams to the mixer (labeled A and B respectively). The mixer
figures out what resolution, bitrate, codec and other properties the participants wants to receive, and sends back a merged version of the streams (labeled $A \land B$). This lets each participant send video data in any supported format, without having to consider what the other participants can receive.

![Diagram](image)

Figure 3: Alice and Bob is in a conference with a mixer.

### 2.1.2 Switching

Switching is a simpler method for reducing bandwidth. With this method all participants send their streams to some central point, which from now on will be referred to as a switcher. The switcher chooses which stream to relay to the other participants. The method for choosing which stream to relay can differ, one example is Voice Activated Switching (VAS) where only the person who is currently talking will be seen by all other participants. Figure 4 shows Alice and Bob in a conference with switching. Both send their streams to the switcher, but only Bob’s stream is being sent back (if using VAS, Bob would be the one talking).

![Diagram](image)

Figure 4: Alice and Bob is in a conference with a switcher. Only Bob’s stream is being relayed to Alice.

A consequence of both switching and mixing is that every participant only needs to send one stream and receive another, regardless of how many others are in a conference.
2.2 Web Real-Time Communication

Web Real-Time Communication (WebRTC) is a framework that allows peer-to-peer communication between web browsers. The technologies in the WebRTC stack and its APIs are currently being standardized by the World Wide Web Consortium (W3C)[1] and the Internet Engineering Task Force (IETF)[13], and implemented by browser vendors such as Google[14], Ericsson[15] and Mozilla.

WebRTC allows browsers to stream audio, video and arbitrary data directly to one another without the need for a central server. This makes it possible to write and run real-time applications such as games and communication services directly in the browser; there is no need for plugins or platform-specific applications.

2.2.1 Architecture

![Simplified WebRTC Architecture, image inspired by [16].](image)

The inner workings and implementation details of WebRTC are outside the scope of this report. However it is important to understand what WebRTC does and how to interact with it. Figure 5 depicts a high-level view of the architecture. WebRTC by itself contains a Voice Engine, Video Engine, and tools for Transport and communication. In short, this means that anything related to media encoding (converting audio and video from one format to another) and compression, as well as low-level networking is handled by the framework. Web browsers and other native applications can access the framework through its C++ API. Web applications cannot access this low-level API for security- and interoperability reasons, so web browsers need to provide
another way for developers to use it. The standard way of doing this is through a Javascript API[1].

Web applications can use the standardized Javascript API to access the functionality of WebRTC. Implementation details such as codecs, transport protocols and interoperability between web browsers are handled by the browser developers and WebRTC implementation. The architecture depicted here is provided by webrtc.org[16], an open-source project maintained by Google, Mozilla, Opera and others. There are other open-source implementations of WebRTC, such as Ericsson’s OpenWebRTC[15].

2.2.2 Interface for web applications

The main components of the API specified by W3C [1] are:

- **RTCPeerConnection** - Used to represent a one-to-one communication channel between two peers.

- **RTCSessionDescription** - Connections and streams are set up using a sequence of offer/answer messages and Session Description Protocol (SDP). This contains metadata about the session, such as IP addresses and ports, audio/video codecs and more. A connection between two peers are initiated by one of them sending an offer, and the other responding with an answer. When this is done, both peers are ready to stream media according to the SDP.

- **RTCIceCandidate** - When setting up a connection, peers need to know what address/port to use for communication. These address/port pairs are collected using the Interactive Connectivity Establishment (ICE)[17], Session Traversal Utilities for NAT (STUN)[18] and Traversal Using Relays around NAT (TURN)[19] protocols. To give a crude summary of the protocols: A pair of clients can send requests to a STUN server to get a list of address/port pairs, or candidates, that they can possibly use to contact each other. STUN includes mechanisms to bypass some firewall rules and Network Address Translation (NAT). If the server has TURN capabilities then some of those candidates point to the server itself, which means that clients can use it as a middle-man for relaying data. ICE is responsible for collecting this list of candidates, exchanging them between clients, and selecting one that should be used for peer-to-peer communication. If everything else fails, a TURN candidate is selected, and all subsequent data between clients will be exchanged through the TURN server. The choice of what candidates to use depends entirely on how the network between the peers is configured.

- **MediaStream** - A MediaStream abstracts the low-level details of an audio/video stream. A media stream is usually created with the getUserMedia API. When this is called, the web browser connects to a feed from an external hardware device such as a web camera or microphone. The feed can then be added to an RTCPeerConnection and be streamed to the remote peer, or be displayed in the browser.

- **RTCDataChannel** - The setup for a DataChannel is similar to a MediaStream, but instead of using a hardware device as input it can be used to send arbitrary data such as text messages. Data is sent using the Stream Control Transmission Protocol (SCTP)[20].
The low-level procedures and protocols of WebRTC are quite complex, but the high-level API simplifies the process of setting up a call, as seen in Figure 6. Here, Alice initiates the call by setting up an RTCPeerConnection with a MediaStream and sending an offer SDP (RTCSessionDescription). Bob receives the offer, prepares his own stream, adds it to the connection and responds with an answer SDP. At this point both Alice and Bob have most of the information needed to stream media to each other. Some information is exchanged later on, e.g. trickle ICE candidates [21].

2.2.3 Signaling

Although media and data can be sent directly between browsers once the RTCPeerConnections have been established, they still need a communication channel, or signaling channel, for the initial offer/answer protocol, i.e. the “offer SDP”, “answer SDP” and “ICE candidates” previously depicted in Figure 6.

The process of setting up a signaling channel is not specified by the WebRTC standard. In theory, any medium can be used for signaling, as long as information (SDP and ICE candidates) can be transferred from one browser to the other. In practice however, the choice of signaling channel is important as it determines the level of security, performance and availability of the application. For instance, writing down the SDP messages on post-it notes and passing them around among peers may be possible in theory, but it would take quite a while to set up a connection that way. More suitable methods for signaling would be the Session Initiation Protocol (SIP)[22], Extensible Messaging and Presence Protocol (XMPP)[23], WebSockets[24], or any other messaging or transport protocol.
2.3 Peer-to-peer networks

Peer-to-peer, as defined by [25] is:

“a class of applications that takes advantage of resources — storage, cycles, content, human presence — available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS and have significant or total autonomy from central servers.”

Though there are many ways to build and use peer-to-peer networks, one thing they have in common is to use some form of overlay network[26] for addressing, discovery, resource sharing etc. In short, an overlay network can be described as a self-organizing application-specific network that lets nodes communicate and route information. The structure and properties of an overlay network is a consequence of the protocol used to build it. In other words, overlay networks are just abstractions used to build and analyze application-level protocols. Some types of overlay networks are depicted in Figure 7:

*Centralized* — Although resource-sharing is done directly between peers, they depend on some central point for discovery (i.e to find each other). An example of a centralized overlay network was Napster[25].

*Unstructured* — Peers are not organized in any pre-defined way, and each peer only have a “local” view of other peers. Communication is done through routing (messages/search queries are relayed to the target node) or flooding (a node who receives a message relays it to all neighboring nodes). Examples of applications/protocols are Gnutella and Cyclon[27].

*Super-peer* — This is a hybrid between a centralized and unstructured network. Some nodes are chosen to act as centralized relays (supernodes), and route messages from multiple neighbors. This was for instance used by the Skype protocol[7].

*Structured* — Nodes are structured in a globally agreed-upon way, usually through ordering of some unique identifier. This is commonly used when implementing Distributed Hash Tables (DHT) such as Chord[28].

1The words “node” and “peer” are used interchangeably here.
3 Methodology

Research methodologies, as described by Håkansson [11] can be divided into multiple categories (shown in Figure 8), the most abstract ones being **Quantitative** and **Qualitative** research. In short, quantitative research methods are used to collect and analyze data that is based on measurements. The data is well-defined, described in known units, and can be used to numerically compare it with other equally defined data.

Qualitative research on the other hand involves more abstract data. If the work involves the collection and processing of information that cannot be mapped to measurable units (and thus, cannot be used for numerical comparison), then a qualitative approach is needed.

Since the intended result of the project is a protocol, and a prototype implemented in software, the methods used must be compatible with those of software engineering. Such methods include an iterative process of planning, development, and evaluation based on abstract requirements, but it also involves collection and analysis of quantifiable performance data. Thus, methods from both categories must be used, although in general they tend to be more of the qualitative kind.

What follows is a summary of the research methods depicted in “The Portal of Research Methods and Methodologies”, Figure 8. Each section is concluded with a description of how the methods have been chosen and applied to this project.

![Figure 8: “The portal of research methods and methodologies” Håkansson [11].](image)
3.1 Philosophical Assumptions

The philosophical assumptions represent different ways of looking at research in general, and decides what kind of conclusions can be expected to be made from the result. For instance, **positivism** assumes that the research (and by extension, the result) is objective and independent from the observer and methods used. Results can be extrapolated and knowledge of a phenomenon can be inferred from one environment to another. This is similar to **post-positivism**, the difference being that the post-positivist recognizes that the experience and opinions of the researcher can affect the result. **Realism** on the other hand does not make the same inferences. Results and conclusions are assumed to be valid for the specific environment in which they were made, and cannot automatically be applied to other environments.

The expected results of this project is a protocol that depend on relatively new and untested technology, namely WebRTC. The environment in which it will be used i.e the Internet and the Web is unpredictable and too complex to simulate. In addition, the tools and environment used to evaluate the protocol are specifically developed for this project. With this environment in mind, a **realist** approach is taken when interpreting the results.

3.2 Research Methods

The Research Methods provide a framework for how the research tasks are to be initiated, carried out and completed. **Experimental research** methods are used by defining and finding variables, and investigating relationships and causalties between them. The researcher can change certain variables and observe how the rest of the system responds. **Non-experimental research** methods involve a similar kind of variable analysis, but instead of changing them to find causalities, the researcher can only study correlations between them in order to make predictions about system behavior.

**Fundamental research** methods are used to observe and understand naturally occurring phenomena. Existing theories are combined with new observations in order to either develop new theories, or refute old ones. **Applied research** methods involve applying the results from fundamental or basic research to solve practical problems.

This project seeks to solve a practical problem, namely the scalability of decentralized conferencing, by developing new software. The techniques used to solve the problem are derived from research in distributed systems and software engineering. In other words, the methods used can be categorized as **applied research**. Since **scalability** is a measure of performance, the software has to be tested and evaluated using **experimental research** methods.

3.3 Research Approaches

The research approaches are used for drawing conclusions and establishing what is true or false about the research result. The three most common approaches are inductive, deductive and abductive research. The approach is chosen based on the pre- and postconditions of the study. If there is enough knowledge about a problem or area to form a well-defined hypothesis that can be confirmed or disproved by collecting and analyzing data, a **deductive** approach can be taken. If that is not the case then an **inductive** approach is used. The result of an inductive study is a new theory, while the result of a deductive study is facts about a pre-existing theory. A combination of the two is called an **abductive** approach. The result of an abductive study is, given a set of possible hypotheses, the one that best explain the collected data or observed phenomenon.

The problem of the project can be crudely summarized as “decentralized conferences do not scale”. But even when constraining the problem to the Conference scalability introduced in
Section 1.1.1, it is still hard to form a well-defined hypothesis on why it does not scale. There are a number of possible explanations such as bandwidth limitations, encoding capabilities, or WebRTC itself. To find the possible causes, and to ultimately solve the problem an abductive approach is needed.

3.4 Research Strategies

Research strategies and designs provide guidelines for how the research is to be organized and conducted i.e the practical application of the research methods described in Section 3.2. The Experimental research strategy involves conducting experiments and collecting large sets of quantitative data in an environment where all the variables that can affect the result are known and controlled. Statistical analysis of the data is used to confirm or disprove an hypothesis. Action research is a systematic, iterative process of planning, taking action, observing and evaluating the solution to a practical problem. Unlike the experimental strategy, action research can be performed when working with limited data sets and qualitative research methods.

As stated in the previous sections, the problem that this project attempts to solve is a software problem that requires a software solution. Therefore, the chosen research strategy must work well together with the processes of software engineering. Action research is a good fit, as it contains the process of identifying the problems, developing a solution, evaluating it and finding new ways to improve it.

3.5 Data Collection

The most relevant data collection method for this study is Experiments. Experiments are described as the collection of large data sets that can be mapped to variables (as described in Section 3.2).

The software developed in this project is evaluated by performing experiments. The experimental environment is described in Section 7. Key performance metrics such as CPU, memory and network usage are measured for each iteration of the software in order to improve it. Some aspects of the software, such as the correctness of an algorithm, can be hard or impossible to identify by collecting data during runtime. Such data is instead collected by observing the model and general behavior of the software.

3.6 Data Analysis

Data analysis methods describe how the data from Section 3.5 are analyzed and interpreted. Statistical methods are used to analyze collected and possibly aggregated data either by drawing inferences from it (inferential statistics), or to describe features of it (descriptive statistics). Computational mathematics uses modeling and simulation for analysis. Analytic induction is an iterative process that alternate between data collection and analysis. Iteration continues until a theory is validated.

This project uses a combination of the three analysis methods. Correctness and other properties of the protocol is analyzed solely by its model, i.e through computational mathematics. The performance of the protocol implementation is analyzed with statistics. The development is done in an iterative process similar to analytic induction, where the iteration stops when the requirements are fulfilled.
4 Modeling language and notation

In addition to standardized UML\cite{29}, software in this report is modeled in a few different ways. Since the protocol and software developed in this project is inherently distributed, a modeling language and notation must be used to properly reflect that. Section 4.1 introduces a language specified in the book “Introduction to Reliable and Secure Distributed Programming”\cite{12} that is specifically designed to represent distributed applications. Sections 4.2 and 4.3 gives a quick introduction Javascript and Graphs, both of which are useful when describing peer-to-peer systems in relation to WebRTC.

4.1 Distributed Systems Abstractions

![Figure 9: Composition model, image taken from \cite{12, p. 38}](image)

To analyze and prove the properties of distributed applications and protocols in an unambiguous way, the modeling language and notation described in \cite{12} will be used. According to the composition model depicted in Figure 9, software is described as a set of components that communicate through asynchronous events. Each component provides some properties to the system that other components can use. An event for component co with type EventType and attributes a1 and b1 is denoted by \(\langle \text{co}, \text{EventType} | a1, b1 \rangle\). The interface of a component, i.e incoming and outgoing events, as well as its properties are defined as a Module. An example module is defined in Module 1, and its implementation is in Algorithm 2. The relevant modules described in \cite{12} are included in Appendix A.

<table>
<thead>
<tr>
<th>Module 1: Interface and properties of a module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Module:</strong></td>
</tr>
<tr>
<td><strong>Name:</strong> ExampleModule, instance em</td>
</tr>
<tr>
<td><strong>Events:</strong></td>
</tr>
<tr>
<td><strong>Request:</strong> (\langle \text{em}, \text{SomeRequest}</td>
</tr>
<tr>
<td><strong>Indication:</strong> (\langle \text{em}, \text{SomeResponse}</td>
</tr>
<tr>
<td><strong>Properties:</strong></td>
</tr>
<tr>
<td><strong>EM1:</strong> Synchronosity: Every request has a response.</td>
</tr>
</tbody>
</table>
Algorithm 2: Implementation of the module ExampleModule

**Implements:**
- ExampleModule, instance \( em \)

**upon event** \( (em, SomeRequest \mid id) \) do
  process \( (id) \);
  **trigger** \( (em, SomeResponse \mid id) \);

4.2 Javascript and JSON

```javascript
// A Javascript object
var constraints = {
  audio: true,
  video: true,
}

// A callback function
function onLocalStream (s) {
  // s is a media stream from webcam/microphone.
}

function onError (e) {
  // e is a getUserMedia error
}

// Call WebRTC GetUserMedia
getUserMedia (constraints, onLocalStream, onError)
```

Figure 10: A javascript program using the GetUserMedia API.

Most of the software developed in this project is used in a web application, and many of the problems are solved using the WebRTC API. To avoid unnecessary translation errors and ambiguity, some solutions are best represented as pure Javascript, or at least pseudo code that can be easily translated to Javascript. Simple key-value objects and messages are sometimes best described using the Javascript Object Notation (JSON). An example program is provided in Figure 10. This program uses the WebRTC getUserMedia API to capture the stream from a web camera and microphone.

4.3 Graphs

Graphs can be especially useful when modeling and analyzing peer-to-peer systems. A graph is a set of *nodes* which are connected by *edges*. The meaning of the nodes and edges depends entirely on the context in which they are used. In general, a node represent some entity, process or computer. An edge represents some communication channel between two nodes. Nodes and
edges can be named and numbered, a number on an edge could for instance represent the *cost of transferring information* over it.

![Figure 11: A weighted directed graph.](image)

Figure 11 shows a graph with nodes A, B, C and D. A can send information to B with a cost of 1, and to D with a cost of 5. There is no way for A to communicate with C, unless information can be *routed* or *relayed* through B.

Some graphs have already been used earlier in this report, namely in Section 2.3, where they were used to display the properties of different peer-to-peer overlay networks. This is an effective way to visualize how a system behaves without knowing any details about how it is implemented.
5 Requirements

So far the only stated requirements for the conferencing protocol has been along the lines of “it must be more scalable than full-mesh, and the centralized alternatives”. These are very ambiguous requirements that need to be refined. In order to evaluate the protocol and its implementation, some formal requirements needs to be set. The focus of this report is not on how to develop a conferencing application, it is rather how to make conferencing applications more scalable. That being said, a top-down approach starting with the possible use-cases of a conferencing system before moving on to the more relevant non-functional requirements can be of some help when designing the protocol.

![Use case diagram for the conferencing system.](image)

The use-case diagram in Figure 12 shows the basic usage of the MP conferencing system. Most of them are self-explanatory and will not be elaborated further. In fact, they can all be derived from the use case “Participate in conference”, Table 1, and can be summarized as the following: The users who want to participate in a conference enter a web page in their browser and clicks a button with the text “join”. This will let them stream the audio from their microphone and the video from their web camera to all the other participants. The media from the other participants will in turn be streamed back to the user. The user exits the conference by clicking a button with the text “leave”. After that point, no media will be exchanged between the user and the other participants unless it joins the conference again.
<table>
<thead>
<tr>
<th>Use case name</th>
<th>Participate in conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actors</td>
<td>Initiated by participant</td>
</tr>
</tbody>
</table>
| Flow of events | 1. Participant clicks a button with text “join”.  
2. Participant can see and hear all other participants.  
3. Participant clicks a button with text “leave”. |
| Entry conditions | Participant has entered a web page.  
Participant wants to join a conference. |
| Exit conditions | Participant has been in a conference. |
| Quality requirements | 1. The time it takes between event 1 and event 2 is at most one second.  
2. All participants are visible at all times during the conference.  
3. All participants are audible at all times during the conference.  
4. The maximum number of participants in a conference is at least 10. |

Table 1: Use case: Participate in conference.

5.1 Non-functional requirements

The non-functional requirements of the system are important. They describe how a system should perform, be packaged and deployed, rather than how the system should be used, which would be described as functional requirements. The non-functional requirements presented in Table 2 were for the most part set at the start of the project, and some adjustments were made to support the use-cases.
### Usability
- **User expertise**: The user needs to know how to enter a web page in a browser.
- **Interface Standards**: The application needs to be able to run in any Chrome browser with at least version 47. The user needs a computer with a microphone, a camera, and a web browser.

### Reliability
- **Service Robustness**: The service cannot have any single point of failure.
- **Conference Robustness**: A conference needs to be available even if some users have severe or fatal connectivity problems.
- **Security**: The audio and video contents of a conference should only be accessible to the participants.

### Performance
- **Scalability (conference)**: Users should be able to participate in conferences with at least 10 concurrent users.
- **Scalability (service)**: The service should be accessible even with a very large number of concurrent conferences.
- **Latency**: There should be no noticeable delay between conference participants.
- **Latency (call set-up)**: The time to set up, i.e join or start a conference should be at most one second.

### Packaging
- **Installation**: The user should not have to install any software, aside from a WebRTC-compliant web browser.
- **Re-usability**: It should be possible to re-use parts of the system by embedding it in a web application as a packaged library.

---

**Table 2: Non-functional requirements**

It is not yet clear whether all of these requirements can be met in a real-world deployment. Regardless of how well the system is optimized, there will always be external factors that cannot be taken into consideration. One example is the requirement on **Latency (call set-up)**. There is no way to predict how the network between users will look like or what hardware is used. This type of requirement will however be considered in the evaluation environment. Other requirements are only vaguely defined for the same reason, such as **Scalability (service)**. These requirements serves more as guidelines for the evaluation, rather than properties of a real-world deployment of the system.
6 System Architecture

The installation and interface standards requirements in Section 5.1 implies that the conferencing system has to be usable in a web application. The re-usability requirement explicitly states that parts of the system has to be re-usable as a packaged library. Based on these requirements, the high-level architecture seen in Figure 13 is chosen. Everything related to the protocol is contained within the module conference.js. The module is, as the name implies, a javascript library. The reason for containing it in a separate module is to be able to re-use it in multiple applications. A consequence of this design choice is that no application-level logic or assumptions can be made inside the module. This has to be handled by the application itself, while interacting with the conference module through its Application Programming Interface (API) and configuration.

![High-level architecture](image)

Figure 13: High-level architecture

Another thing to note is the signaling.js module. This contains the signaling mechanism that one browser can use to pass messages to another. In practice, this signaling module is only used to bootstrap the internal signaling in conference.js (more details later). The reason for this separation of conferencing and signaling is to give the application developer control over how peers find each other. Consider for example a scenario where a company wants to provide a conferencing service to its users, and already have a messaging solution in place such as XMPP, Socket.io or SIP, with authentication, monitoring, infrastructure etc. They would only have to implement this small component to make it work together with the rest of their system.

6.1 Interfaces

For an application to support the use-cases it needs some way to control the conference, with operations such as “Join”, “Leave”, “Share camera”, and indications such as “Participant joined/left/shared camera”. The architecture depicted in Figure 14 is expanded with two interfaces: a Conference API and a Signaling API. WebRTC, which is included in the Browser API, is extracted to emphasize its importance for the conference.js module.
The Conference API is used by the application in order to start and end a conference, and get the information needed to display the video streams. The API is shown in Module 3, with the notation described in Section 4.1. The properties are derived from the non-functional requirements.

**Module 3: Interface and properties of ConferenceJS.**

**Module:**
- **Name:** Conference, **instance** `conf`

**Events:**
- **Request:** `(conf, Join)`: User requests to join a conference.
- **Request:** `(conf, Leave)`: User requests to leave a conference.
- **Indication:** `(conf, state | s)`: The state of the conference was changed to `s`.
- **Indication:** `(conf, stream | p, s)`: A media stream `s` from user `p` is available for display.
- **Indication:** `(conf, streamdown | p, s)`: A media stream `s` from user `p` is no longer available.

**Properties:**
- **C1:** Fast set-up: If user `p` joins a conference with user `q`, then the time it takes until `p`, can see `q` is at most one second.
- **C2:** Robustness: If a correct user `p` is in a conference with a correct user `q`, then `p` should get audio and video from `q` without disruption.
- **C3:** No stream duplication: If a user `p` is in a conference with a user `q`, then `p` should get exactly one media stream from `q`.

The API is kept as simple as possible, providing only the absolute minimum in terms of methods and callbacks that a conferencing application would need to implement. The Signaling
API, depicted in Module 4, is kept minimal as well.

**Module 4: Interface and properties of SignalingJS.**

**Module:**
- **Name:** Signaling, instance \(\text{sig}\)

**Events:**
- **Request:** \(\langle \text{sig}, \text{send} \mid p, m \rangle\): Request to send message \(m\) to process \(p\).
- **Request:** \(\langle \text{sig}, \text{broadcast} \mid m \rangle\): Request to broadcast message \(m\) to all processes.
- **Request:** \(\langle \text{sig}, \text{isAlive} \mid p \rangle\): Request to check if process \(p\) can respond to messages.
- **Indication:** \(\langle \text{sig}, \text{MessageDeliver} \mid p, m \rangle\): Delivers a message \(m\) that was sent from \(p\).
- **Indication:** \(\langle \text{sig}, \text{BroadcastDeliver} \mid p, m \rangle\): Delivers a message \(m\) that was broadcasted from \(p\).

**Properties:**
- **S1:** Broadcast isolation: If a broadcasted message \(m\) is related to a conference with participants \(P\), then only \(P\) can see the contents of \(m\).
- **S2-S4:** Same as PL1-PL3 of perfect point-to-point links, introduced in Module 6.
- **S5-S7:** Same as BEB1-BEB3 of best-effort broadcast, introduced in Module 10.

As long as the signaling.js module implements this interface, it can be used as a communication channel for conferencing. It should be noted that the protocol in conference.js makes some assumptions about the ordering and reliability of this signaling mechanism, which makes it non-trivial to implement.

### 6.2 Hardware

While there are no hardware requirements for the protocol itself, aside from a device with a network connection, some hardware is required for an application to use it. Figure 15 shows one possible hardware architecture that meets the requirements.

![Figure 15: Hardware components of a conferencing system using the protocol.](image)

The user needs a device that is capable of running a web browser (Installation requirement).
That browser needs access to input devices such as a web camera and a microphone, and output devices such as a display and a speaker \((\text{Interface standards requirement})\). The conference.js library and the application needs to be requested by the client web browser, which means they have to be hosted on a web server \((\text{Re-usability requirement})\). The web content can be served from a cloud service or a Content Delivery Network (CDN) in order to ensure Service Scalability \((\text{Scalability (service) requirement})\). The architecture of the signaling mechanism is not specified in the protocol, but it does have to provide a way for information to be passed from one web browser to another within the timing constraints set by the \textit{Latency} requirements. The hardware required is essentially just two or more interconnected machines that are capable of running a web server that can be reached using browser-compliant protocols.
7 Evaluation Environment

To evaluate the performance of the protocol and to find problems in the implementation, an environment has to be set up for running the software and to collect data. The environment described here includes all the hardware and software used during the evaluation. Most of the software used for data collection and analysis consists of various free-to-use and open-source projects, while some of it was developed specifically for this project.

The goal was to have an environment that could simulate multiple users of the conferencing system, preferably enough to properly test the performance requirements set earlier. Data would be collected and analyzed in order to determine whether the software met the requirements. Unfortunately, one limiting factor was access to hardware. In order to truly measure the scalability of the protocol, particularly the Service Scalability, a large number of hardware devices would be needed. The project did not have that kind of access. Fortunately, the Service Scalability is quite predictable and can sufficiently be evaluated using the system model (not described in this chapter), due to its decentralized nature.

The focus of the evaluation environment is thus on Conference Scalability, i.e to see how the system performs for an increasing number of participants in a single conference.

7.1 Test scenario

The testing scenario is inspired by the “Participate in conference” use case presented in Section 5. Each user enters a URL in a web browser, clicks a button with the text “join”, waits a few minutes, and then clicks a button with the text “leave” before exiting the browser. For the evaluation, these steps have to be performed in an automated way. Performance and call-quality data needs to be collected during this time so that it can be analyzed once the conference has ended.

7.2 Hardware

The hardware used in the environment consist of seven machines, depicted in Figure 16. All server software, simulating the service provider, executes on a single Mac Mini machine. This machine will be referred to as the server. The client software i.e the users' web browsers run on five Intel NUC Mini PC:s, which will be referred to as the clients. The machines are connected through a 100 Mbit/s ethernet switch.

![Figure 16: Hardware setup](image)
The specifications[30] for the server machine are:

- **Model:** Mac mini (late 2012)
- **Processor:** 3.1 GHz Intel Core i5 3210M Ivy Bridge (2 cores, 4 threads)
- **Memory:** 16 GB 1600 MHz DDR3
- **Network:** 10/100/1000 Gigabit Ethernet
- **Operating System:** MacOS 10.10.2

There is no particular reason for why this machine was chosen. It was simply available, and since it was also used for development it was easy to use it for service deployment.

The specifications[31] for the client machines:

- **Model:** Intel NUC Kit D54250WYK
- **Processor:** 2.6 GHz Intel Core i5 4250U Haswell (2 cores, 4 threads)
- **Memory:** 16 GB 1600 MHz DDR3
- **Network:** 10/100/1000 Gigabit Ethernet
- **Graphics Processor:** Intel HD Graphics 5000
- **Operating System:** Ubuntu Server 15.10

These machines were again chosen out of convenience. They are relatively small general-purpose machines, with performance comparable to a low-end laptop computer[32].

### 7.3 Data collection

A number of open-source software tools are used to run the test scenario and to collect performance data. Introducing external tools does increase the complexity of the experimental environment, which can make it harder to interpret the results in a correct way. But some correctness is sacrificed for reproducibility and significantly decreased development time.

There are two categories of collected data: Resource Usage and Conference Quality. Resource usage data is collected by monitoring the CPU usage and network traffic of the client machines. Conference Quality includes measurements such as resolution and frame rate of the video streams in a conference. It is collected using the APIs provided by WebRTC.

#### 7.3.1 Running the test

To run the test scenario, the browser-automation tool Selenium WebDriver[33] is used. This tool makes it possible to start and control web browsers on remote machines through scripts. Docker[34] is used to simplify the deployment. Each web browser and in fact almost every application is run as a Docker Container. This makes it easier to configure, monitor and isolate each application, which is particularly useful when running the same set of applications on several machines.

Selenium and Docker are used by starting and running the test on one Chrome web browser on each of the five client machines. This simulates five individual users participating in a conference. The clients are remotely controlled from one central hub located on the server machine with the use of Selenium Grid Hub[35]. It works by executing a script on the central hub, which causes commands to be executed on the client nodes. The script can be seen in Figure 17.
var wd = require ('webdriverio');

var matrix = wd.multiremote (
    browserA: {
        browserName: 'chrome',
        chromeOptions: {
            args: [
                'use-fake-device-for-media-stream',
                'use-fake-ui-for-media-stream',
                'use-file-for-fake-video-capture=/fakemedia/johnny.y4m'
            ]
        },
        browserB: ...
        browserC: ...
        browserD: ...
        browserE: ...
    });

var sleepTime = 270000;

matrix
    .init ()
    .url ('https://SERVER_URL')
    .pause (1000)
    .click ('#joinButton')
    .pause (sleepTime)
    .click ('#leaveButton')
    .pause (1000)
    .end ();

Figure 17: Script used to run the test scenario on five browsers.

Three arguments are used when starting the Chrome browsers, as seen on lines 8...10:

- **use-fake-device-for-media-stream** tells Chrome that it should not use a physical device to capture audio and video with the WebRTC getUserMedia API. The client machines are not equipped with a web camera or microphone. Instead, media streams are generated by the browser for use in tests and simulations. This has the advantage that all clients encode and send almost identical streams, which should make the performance data more predictable and symmetrical, and thus easier to reproduce.

- **use-fake-ui-for-media-stream** tells the browser not to show the permission dialog box when getUserMedia is called. If this argument is not given, the user explicitly has to give Chrome permission to capture media when it is requested.

- **use-file-for-fake-video-capture=/fakemedia/johnny.y4m** replaces the browser’s default generated video stream with a video file. This argument is used because the default stream is not representative of a real web camera, in terms of encoding resources and
bandwidth used when streaming it between browsers[36]. The file “johnny.y4m”[37] is a 10 second video clip with the resolution 1280 × 720 pixels, and the frame rate 60 frames per second (FPS).

7.3.2 Resource usage monitoring

A number of tools are used to monitor the computing and networking resources of the client machines. Prometheus[38] collects hardware performance data and provides mechanics for storing, formatting, and querying it for analysis. The data is transported from the clients to the server machine, and visualized in real-time by Grafana[39].

Performance data from the Selenium containers (Selenium Chrome Node) is collected using Google Cadvisor[40], a container monitoring tool. This data is transported, stored and processed in the same way as the hardware performance data. All monitoring software is deployed as Docker containers on both the server and the client machines. The setup is depicted in Figure 18.

![Figure 18: Docker containers](image)

7.3.3 Conference Quality

The quality of a conference call is measured by the resolution and frame rate of the media sent between participants. These metrics are collected because of the way WebRTC adapts them according to the available CPU and networking resources[41]. Degraded resolution or frame rate indicates that a machine is unable to process or transfer high-quality video, which is why it is important to correlate the conference quality with resource usage. Measuring the quality of video calls is a whole research area on its own, which could be done by accounting for a wide range of variables. Measuring only the frame rate and resolution is certainly a simplified approach, but it should be enough to give some indication of how the quality is perceived by the user.

The conference quality metrics are collected using the WebRTC GetStats API[42]. The API is called periodically by the application in each client browser. Each call gathers information about every RTCPeerConnection i.e the connection between two participants. It includes information about media tracks such as encoded frames, delay, jitter, resolution, bytes sent and received.
etc. The information is sent to the server machine and stored for analysis. The total number of measurements gathered from one test scenario is approximately 3000, or 600 per participant.

7.4 Data analysis

The data collected during the tests are analyzed in order to get a better understanding of how the protocol performs and how it can be improved. Key performance metrics such as network and CPU usage, and quality metrics such as video frame rate and resolution are extracted from the data. Because every data point includes a timestamp, correlations can be found through time-series analysis.

The evaluation environment and the protocol can be re-configured between tests, for instance by changing the number of conference participants. This is done to further understand the data and to find causations between the Quality and Resource Usage measurements.

After each iteration, the implementation is compared against the requirements in order to find problems and to determine how it can be improved. Once the collected data has been analyzed and problems have been found, the model used to implement the protocol is further examined to find possible causes. The problems are resolved during the next iteration by changing the model or the way it is implemented, before performing the tests again.
8 Full-mesh

To get a better understanding of how WebRTC performs and how it is best used in a video conferencing scenario, a first step is taken to implement a protocol for the full-mesh model. The model and the implementation is evaluated in the environment presented in the previous chapter, and the results work as a point of reference for the more sophisticated solution presented later on. To understand how the conferencing protocol is used, a good place to start is with the application on top of it.

8.1 Application

The use-cases presented earlier describe a trivial application where users enter a web page and click a button to join a conference. To achieve this, the user interface has 4 components, depicted in Figure 19. The buttons used to join and leave in the top-left of the figure, the local stream i.e the user’s web camera just below the buttons, and the streams from all the other participants in the bottom. This figure was produced by joining a conference in 4 browser tabs on the same machine, which means that all participants use the same web camera as its source.

Figure 19: The conferencing application user interface.

8.2 Model

The goal here is to create a full-mesh, or a fully-connected overlay network, where every node is connected to every other node, as depicted in the graph in Figure 20. A connection includes not just the means to send data, but to also to send and receive media streams. The nodes in the graph are web browsers, and the directed edges represent WebRTC PeerConnections. Each PeerConnection include the media stream from the participants’ cameras and microphones.
The protocol is described in Algorithm 5. When the user clicks the “join” button, the event \((conf, \text{Join})\) is triggered. This in turn triggers the \((sig, \text{Broadcast} \mid [\text{JOIN}])\) event that will be propagated to all other participants (or none if there are no participants at the moment). A participant who receives the broadcasted message will initiate a WebRTC PeerConnection and send an offer, as specified in the WebRTC offer/answer protocol, Section 2.2.2. Once the offer/answer protocol is completed between all participants in the conference, they should be able to see and hear each other.
Algorithm 5: Full-mesh conference

Implements:
Conference, instance \textit{conf}, Module 3.

Uses:
Signaling, instance \textit{sig}, Module 4.

\begin{verbatim}
upon event \langle \textit{conf}, Init \rangle do
  connections := \emptyset;
  streams := \emptyset;

upon event \langle \textit{conf}, Join \rangle do
  localStream := getUserMedia();
  trigger \langle \textit{conf}, Stream | self, localStream \rangle;
  trigger \langle \textit{sig}, Broadcast | [JOIN] \rangle;

upon event \langle \textit{conf}, Leave \rangle do
  trigger \langle \textit{sig}, Broadcast | [LEAVE] \rangle;
  trigger \langle \textit{conf}, Streamdown | [ALL] \rangle;

upon event \langle \textit{sig}, BroadcastDeliver | p, [JOIN] \rangle do
  connection := createPeerConnection(p);
  connection.setLocalDescription(localStream);
  connections := \{connection\} \cup \text{connections};
  trigger \langle \textit{sig}, Send | p, [OFFER] \rangle;

upon event \langle \textit{sig}, BroadcastDeliver | p, [LEAVE] \rangle do
  connection := \text{connections}[p];
  connections := \text{connections} \setminus \{\text{connection}\};
  trigger \langle \textit{conf}, Streamdown | p, \text{remoteStreams}[p] \rangle;

upon event \langle \textit{sig}, MessageDeliver | p, [OFFER] \rangle do
  connection := createPeerConnection(p);
  connection.setLocalDescription(localStream);
  connection.setRemoteDescription([OFFER]);
  connections := \{\text{connection}\} \cup \text{connections};
  trigger \langle \textit{sig}, Send | p, [ANSWER] \rangle;

upon event \langle \textit{sig}, MessageDeliver | p, [ANSWER] \rangle do
  connection = \text{connections}[p];
  connection.setRemoteDescription([ANSWER]);
\end{verbatim}
8.3 Measurement results

Five machines join the same conference simultaneously, wait for just over 4 minutes (260 seconds) while performance data is being collected, and then leave the conference and exit the application.

Figure 21 shows the CPU usage during this time. One line is displayed for each of the machines, labeled nuc06...nuc10. The CPU usage ranges from 0% to 400%. This is a common way to display performance data for multi-core processors, and these machines in particular have 4 cores each.

![CPU Usage Percentage of chrome-node](image)

Figure 21: CPU Usage for all nodes in a full-mesh conference.

It is hard to say anything about the performance by just looking at the CPU usage, but the fact that every machine spends almost all of its computing resources on a conference with only five participants does say something about the protocol’s ability to scale for larger conferences.

Figure 22 shows how much of the available bandwidth is used to send media data to other participants. The bandwidth, or upload rate, peaks at almost 11 Mbit/s before stabilizing at just over 8 Mbit/s. The initial peak seems to correlate with the peak in CPU usage.
Since all data is sent through a switch with a maximum throughput of 100 Mbit/s, it is unlikely that network congestion or packet loss is limiting the data flow in any way.

The rate of received data, Figure 23, is almost identical to the rate of sent data for every node. This is expected since the nodes should only communicate with each other, the exception being the signaling server and the data collection server. Some performance data is sent from the nodes to the data collection server (not shown here) during the test. This would explain the slight difference in data sent and data received.
To better understand the performance data, in particular the initial peak in CPU and network usage, the quality of the video streams are examined. The internal audio- and video engine of WebRTC can adjust the frame rate and resolution of the transmitted media according to the nodes’ ability to encode, decode and transfer it. This means that any degradation in media quality is an indication of resource exhaustion.

To produce figures 24 and 25, one pair of nodes is selected at random, and the data gathered from the WebRTC “getStats” API is analyzed. Since the performance data of all five nodes are almost identical, the choice of which pair to study is arbitrary. Although the results are verified by comparing it with other pairs.

Figure 24 shows the frame rate (frames per second, FPS) of the video stream sent and received between two nodes.
The initial frame rate of the video streams is approximately 20 fps. After a minute it has increased to almost 60 fps, which is the frame rate of the streaming source.

Figure 25 shows how the resolution of the video is changed as the conference progresses. Although only the video width is shown in the graph, the height is always set to retain the aspect ratio 16 : 9. The resolutions for a video of widths 1280, 960 and 640 are 1280 \times 720, 960 \times 540 and 640 \times 360 pixels respectively.

When the conference is first started, the nodes attempt to send their video streams with a 1280 \times 720 resolution. This is the same resolution as the video source. The resolution is downgraded to 960 \times 540 and after a minute it remains stable at 640 \times 360. The initially high resolution correlates with the peak in CPU and network usage shown earlier, and the decrease in resolution correlates with the increase in frame rate.

8.4 Problems

By looking at the resolution and frame rate of the video streams, it is apparent that the nodes are not capable of streaming high-resolution video at a high frame rate to every other node, even
in a conference with only five participants. It has already been established that the network is not fully utilized, which means that limitations of the CPU must be the cause of this downgraded video quality. Furthermore, by comparing these results with the two-party full-mesh conference in Appendix B it becomes clear that this is a scalability problem. When there are only two participants in a conference, the video frame rate and resolution remains close to the original source (1280×720 at 60 fps), while transmitting 3 MBit/s (upload rate and download rate) over the network.

There are other problems with the full-mesh protocol that are not immediately apparent by looking at the collected data. They can however be found by observing the model.

8.4.1 Excessive use of TURN

In some network configurations where nodes cannot communicate directly with each other, for instance between certain enterprise networks, they must resort to using a TURN server. While a TURN server does not have to process the media streams, it does have to relay them between nodes. Consider the configuration in Figure 26, a conference with four participants who are all using the same TURN server. If every node sends a 3 Mbit/s stream to every other node, the TURN server would be required to send and receive $3 \times 10 = 30$ Mbit/s. The required bandwidth increases exponentially with the number of participants.

![Figure 26: A TURN server relays traffic for nodes that are behind protected networks.](image)

While this is not uniquely a problem for the full-mesh conference, it still has to be taken into consideration since a conferencing service’ ability to scale depends on how much it relies on TURN servers.

8.4.2 Signaling race condition

WebRTC requires that SDP messages are handled in a specific order[1, section 4.4.1], but the current protocol does not rely on an ordered broadcast abstraction for signaling. This means that the initial offer/answer protocol (started by ⟨conf, Join⟩) can fail if two participants try to join at the same time. At the moment no attempt is made to recover from such a failure.
9 Active Speaker switching

To address the performance problems of the full-mesh conference, the protocol needs to include some way to limit the resource requirements of a conference. Section 2.1.2 introduced the process of voice-activated switching (VAS) to reduce the bandwidth- and computing resources used. Switching usually involves a central server that decides which media stream should be sent to the participants, based on who is currently speaking. This is not good enough, since the requirements clearly state that A. The service cannot have any single point of failure and B. The audio and video contents of a conference should only be accessible to the participants. Relying on a central server that has to analyze the media streams would not comply with these requirements.

Additionally, the quality requirements imply that all participants have to be visible throughout the duration of a conference. This means that a switching mechanism cannot just select one stream that should be sent to the participants.

To solve this, a distributed switching mechanism is added to the protocol that controls the quality of the individual video streams. When combined with an audio analysis module, the protocol can select one participant to be the active speaker. The video stream coming from the active speaker is a high-resolution, high-framerate stream, while the stream of every other participant is of lower quality. The changes and relevant modules are in Appendix C.

9.1 Application

Adding a switching mechanism to the protocol fundamentally changes how it is meant to be used in an application. The remote video streams are no longer considered equally important. Instead there is only one that should be the main component of the user interface, and the other low-quality videos can be displayed as smaller thumbnails, only meant to show the presence of other participants. Figure 27 depicts what such an application could look like.

![Figure 27: Application UI using voice-activated switching. The active speaker is selected by the conferencing protocol.](image)

For the purposes of this project however no major changes are made to the application. The fact that one participant is selected as active speaker is only indicated by the quality of its video stream.
9.2 Model

To implement voice-activated switching in a distributed way, the protocol is going to include some of the abstractions introduced in [12]. Their properties and interfaces are described in Appendix A.

To ensure that the switching is done in a reliable way, all the nodes need to agree about which node is currently the active speaker. In other words they need to reach a consensus before any node attempts to perform the switch. There are a number of different consensus algorithms with different levels of complexity. One of the simpler methods is to elect some node to make decisions on behalf of the others, instead of having all nodes participate in a discussion about every decision. This is known as a leader election algorithm or protocol\(^2\).

There are a number of different abstractions for leader election. The abstraction used at this stage of the project is the Eventual Leader Detector (ELD), Module 9. While this is certainly not the strongest algorithm for leader detection (in terms of its agreement and accuracy properties), it should be sufficient for this type of application. The ELD used here is implemented as the Monarchical Eventual Leader Detector[12, p. 100], and it depends on an Eventually Perfect Failure Detector (EPFD), Module 8, which in turn uses the Perfect Point To Point Link (PP2P) abstraction, Module 6. In short, a node will trust another node to become leader if the following is true:

- It is not suspected to have failed, i.e it is trusted by the failure detector.
- It has an ID that is ranked higher than every other unsuspected node. The ID:s are ranked in a globally agreed-upon way.

Introducing new abstractions to the model requires an increasing number of messages to be sent between nodes. Previously all non-media communication was done through the Signaling module, which uses a centralized server to relay messages. To prevent the signaling server (and by extension the Service Scalability) from being affected by this, a LowLatencySignalingChannel is introduced. This module uses WebRTC DataChannels to send most messages directly between nodes. The specification for the DataChannel lists similar properties as PP2P, which means it can be used to implement the abstraction. The implementation of the LowLatencySignalingChannel can be seen in Algorithm 12.

Once a node has been elected as leader, it is responsible for choosing an Active Speaker. It choses the active speaker by periodically analyzing the audio output from every participant using the PeerConnection getStats API. If one participant is louder than the others, then it is chosen as active speaker. The procedure for selecting the active speaker is shown in Algorithm 13.

The final composition or architecture of the conference.js module with VAS is depicted in Figure 28.

\(^2\)The words algorithm and protocol are used interchangeably here. This is because a distributed algorithm by definition includes steps for communication between physically separated components, i.e a protocol.
To address the SDP race condition mentioned in the previous chapter, the protocol is modified so that the offer messages sent between a given pair of nodes are always sent from the same node. If the other node wants to renegotiate the connection, it does so by sending an offerme request. It is arranged so that the node with the highest ID always sends the offer. Additionally, the offer/offerme messages are expanded so that the requesting node can state what video quality it wants to receive. An example is shown in Figure 29, where Node1 (left) requests a high-quality stream from Node2 (right).
9.3 Results

The data collection process is exactly the same as for the full-mesh conference. The conference is started and joined by five participants or nodes, performance and quality data is collected for just over four minutes, and then they exit the application. To make the results easier to interpret, the switching mechanism is configured to sample the audio output levels of each node every 30 seconds.

9.3.1 Resource Usage

Figure 30 shows the CPU usage for each individual node. The first active speaker is chosen after 30 seconds. This means that for the first 30 seconds of the test, every node is sending a low-quality stream to every other node. This can be clearly seen in the figure as the CPU usage of every node is below 100%, which is equivalent to one CPU core being fully utilized.
After 30 seconds, the CPU usage of **nuc06** rises to about 370%. This indicates that it has been chosen as active speaker, and is now sending a high-quality stream to the other participants. At this point the other participants have to receive and decode that high-quality stream, which causes a slight increase in CPU usage.

A new node is elected as active speaker every 30 seconds. After five switches nuc06 is elected again, and remains as active for the remaining duration of the test. There is a drop in the CPU usage of nuc06 at around four minutes. To understand why this is, the other performance and quality metrics have to be examined.

Figure 31 shows the rate at which every node is sending data over the network. There is a clear correlation between the CPU usage and data sent from each node. The data sent from the active speaker should be similar to the data sent during the full-mesh conference, as it is sending high-quality streams to every other node.
The data rate of the active speakers are, as predicted, similar to the full-mesh participants. The data-rate of the non-active participants is significantly lower. For example, the legend shows that the rates for nuc08 and nuc10 whom were never elected as active speaker never exceeds 1.16 Mbit/s. This means that throughout the duration of the test, those two nodes are only sending low-quality streams to every other participant, as they should.

There is a notable decrease in the data rate of nuc06 during the last minute of the test. This seems to correlate with the decrease in CPU usage around the same time. In the full-mesh conference, this pattern was indicating that the node was unable to encode and send high-quality video, which caused WebRTC to lower the quality of the stream.
Although the rate of received data shown in Figure 32 does not provide any information that could not be derived from the sent data, it is interesting to visualize the improvements over the full-mesh model. The active speakers only have to receive at most 1 Mbit/s, while the others receive just under 4 Mbit/s. The decrease in transferred data from the active speaker during the last minute is confirmed.

9.3.2 Conference Quality

To better understand the performance data, the conference quality for a pair of nodes is examined. Figure 33 and 34 depicts the resolution and frame rate measured on streams sent over the connection between nuc09 and nuc06, where the measurements are performed by nuc09. For clarification: The green peaks appear when nuc09 is sending a high-quality stream to nuc06, and the red peaks appear when nuc06 is sending a high-quality stream to nuc09.

The nuc09 node is elected active speaker two times. During its time as active speaker, the resolution of the sent stream is $1280 \times 720$, but the frame rate sent never exceeds 30 fps. The same is true for the first two times nuc06 is active speaker.
When nuc06 is selected as active speaker for the third time after about 180 seconds (three minutes) it remains as such for the remaining duration of the test. During this time there is a pattern that is similar to what was seen in the results of the full-mesh test. The frame rate and resolution fluctuates, where an increase in one correlates with a decrease of the other. At the end of the test the video quality is set to $960 \times 540$ pixels at 40 fps. Unfortunately, the test ends before a convergence can be observed.

To find out what happens if the active speaker is given more time to adapt the video quality, another test is performed. This time the Voice Analyzer and Switcher is configured so that the active speaker switching occurs very rarely. This is done by adjusting the audio sampling rate, and the threshold for what volume is considered ‘speech’ rather than noise.
The results in Figures 35 and 36 show a familiar pattern. The video resolution and frame rate fluctuates for over a minute after a node has been selected as active speaker. The quality finally stabilizes at 640×360 resolution and 60 fps, which is a minor improvement compared to the full-mesh conference.

9.3.3 Recovering from failure

There was a bug in the testing environment that caused the client nodes to sometimes crash when they were stressed[43]. This was particularly noticeable during one test run, as every node crashed as soon as it was selected as active speaker. The results produced during that run can be seen in Figure 37.
The results in the figure are interesting, even though it was a failed test. In addition to showing how much bandwidth is used by the active speaker in conferences of different sizes, they also show that the conference can continue in the event of node failures. This implies that:

- Nodes are able to quickly select a new active speaker.
- They can properly detect node failures.
- They can elect a new leader if the previous one crashes.

### 9.4 Problems

The new protocol is significantly more efficient than the previous full-mesh, at least in terms of total resources used during a conference. The leader election and active speaker switching algorithms work well in the testing environment. Most of the problems with the full-mesh solution were solved. However, there is one issue that limits the conference’s ability to scale.

#### 9.4.1 Load imbalance

The biggest problem with the current solution is that the Conference Scalability is limited by the active speaker node. The active speaker still has to encode and send high quality media to every other participant (Figure 38), which requires a lot of computation and bandwidth. The machines used in the test environment are not capable of streaming video at 1280×720 resolution.
and 60 fps to four peers simultaneously, which causes WebRTC to downgrade the quality of the stream.

Meanwhile the inactive participants only have to decode the high quality stream while encoding and sending low quality, which requires significantly less resources, as seen in the test results for resource usage. This leads to a scenario where one node requires more resources than it has available, while other nodes have resources that are not used.

Figure 38: Load imbalance in switched conference.
10 Load balancing with relayed streams

In order to improve the Conference Scalability, the problem with load imbalance needs to be solved. The core of the problem is that streaming high-quality media to multiple endpoints requires:

- Encoding the media once for every endpoint and codec. The tests have so far shown that this is a very CPU-intensive task. The nodes in the one-to-one conference in Appendix B used half of their CPU-resources just to encode and decode one stream.

- Sending the media to every endpoint. While this was not a limitation during the tests, they did show that it requires approximately 3 Mbit/s of upload bandwidth for each stream. This may not be a problem if all participants are on the same local network, but it definitely is if data is transferred over the Internet, and in particular over mobile networks[44].

The process of encoding and sending high-quality media from the active speaker must somehow be distributed among the participants in order for the protocol to scale for larger conferences. The additions to the protocol can be found in Appendix D.

10.1 Model

One possible solution is to have each participant acting as a relay for the active speaker stream. Figure 39 shows how the active speaker, labeled \( A \), captures, encodes and sends a high-quality stream \( S_A \) to node \( B \). Node \( B \) views the stream, and then encodes and sends it as \( S_{AR} \) to node \( C \). This process is repeated until every node receives the stream.

![Active speaker stream, relayed in a chain](image)

Figure 39: Active speaker stream, relayed in a chain

With this model, the active speaker is no longer responsible for encoding and sending high quality video streams to all other participants. Instead only one stream is sent from each individual node. When a new node joins the conference, it contributes with its resources, making the conference more scalable. However there are some issues with this model that makes it non-trivial to implement:

- **Latency** - Decoding, encoding and sending a media stream over a network takes time. Information have to be passed down the chain from the active speaker all the way to the last node, and each step in the chain introduces some delay. For instance, the latency \( L_{AE} \) between the active speaker \( A \) and the node \( E \) in the figure is \( L_{AB} + L_{BC} + L_{CD} + L_{DE} \). This includes the time it takes to send packets from one machine to the other, and the time it takes for each node to process the media stream.
• **Quality** - Media streams sent with WebRTC in Chrome are encoded using the VP8 video codec and the Opus audio codec, both of which use lossy compression\[45][46]. This means that some information is lost when a stream is sent from one node to the other. If a stream is repeatedly encoded and decoded in a relay chain, it will likely result in degraded quality.

• **Propagating failures and performance bottlenecks** - Every node in the chain is depending on the node before it. This means that a failure near the start of the chain will affect everyone that comes after, which in the worst case could affect every single participant. The same is true for nodes with performance issues, such as inability to encode high-quality media and send it over the network. Essentially, the performance of the chain is only as good as its worst node.

One way to mitigate these problems is to build a relay tree rather than a chain, i.e every node relays media to two or more nodes, as depicted in Figure 40.

![Figure 40: Active speaker stream, relayed in a tree](image)

This decreases the number of nodes that a stream has to pass through in order to reach everyone (\(\log(N)\) for a conference with \(N\) participants), which could lead to lower latency and higher quality for the nodes at the bottom of the tree. Additionally, node failures near the top only affect at most half of the other participants, while failures can occur at the bottom without affecting anyone.

Whether the nodes form a tree or a chain, one way to further mitigate the above problems is to optimize how they are ordered. As node failures and poor performance have a larger effect the closer it is located to the root of the tree or start of the chain, it is preferable to place such nodes closer to the bottom/end. To optimize the ordering, the protocol needs to include a mechanism for measuring the performance, or utility of the nodes and the links that connect them. Such a mechanism could determine the utility of a node based on the following questions:

• What type of hardware does the node use, i.e how well does it perform when encoding video of a certain codec?

• Is it a laptop or mobile device powered by a battery?

• How often does a node crash or deliberately leave a conference?

• How is the network between nodes configured? Are they on the same LAN, is it wireless, or are they on a cellular network? Is it a metered connection?

The more information that can be gathered, the better it can be optimized. Furthermore, the nodes need to agree on the same order to ensure that everyone receives the stream.
10.1.1 Agreement

The protocol already includes a mechanism for consensus: The leader election algorithm used for selecting the active speaker. This can be used to determine the order, or topology of the nodes, by letting the elected leader decide on an order and broadcast it to the others. This information can be included in the event \(<sc, Broadcast [<ACTIVESPEAKER>]\).

10.1.2 Ordering

The simplest way of ordering nodes would be to use the same mechanism used in the leader detection algorithm, which is based on node identifiers. Nodes are ordered based on their utility, which is determined by the utility function $U_{naive}$:

$$U_{naive}(node) = node.id$$

While this would guarantee that nodes are ordered, it does not take into consideration the performance of the nodes, which as mentioned earlier can cause problems. Another solution is to include information about a node’s performance in any of the events sent during the conference initiation, such as the $JOIN$ event:

$$\langle sig, Broadcast[j[JOIN], [PERFORMANCE]] \rangle$$

Where $[PERFORMANCE]$ includes information such as number of CPU cores, network interfaces, available memory etc. With this information, the utility function can be better used to determine the ordering of nodes:

$$U(node) = (C \times node.cpu) + (N \times node.network) + (M \times node.memory)$$

10.1.3 Tree structure

Once the nodes are ordered according to their utility, they need to know how to structure the tree. Perhaps the most optimal way is to place the active speaker at the root, and then build a binary tree where nodes are inserted from left to right (also known as a heap). For instance, if the nodes $[A, B, C, D, E, F]$ have been sorted according to their utility, where $A$ is the active speaker, then they would form the tree shown earlier in Figure 40.

This ensures that high-utility nodes are placed closer to the top, while low-utility nodes are closer to the bottom. The high-utility nodes are more likely to act as relays, which reduces the risk of the tree having performance bottlenecks and failures.

10.2 Problems and future work

Unfortunately, this was not completely implemented and thus cannot be tested. In theory, the relay tree described above should solve the problem with load balancing and increase the Conference Scalability of the protocol. But without properly testing the performance and quality it is hard to say anything conclusive about how it works in practice.

Aside from the obvious future work of implementing and testing the protocol, there are other problems with the model that require future research.
10.2.1 Recovering from failure

The current protocol does not specify how to recover when node- or link failures occur in the tree. Once the tree structure is broken when a node crashes or leaves the conference, there needs to be a mechanism that quickly restores it without causing too much disruption. One solution could be to have one of the children of the crashed node to take the place of its parent, but that would mean that low-utility nodes are moved up in the tree, which could cause further failures. Another solution is to order the nodes again into a new tree structure. But that would likely cause even more disruption, as connections will have to be re-negotiated. Alternatively, other tree structures that are more resilient to failures could be more optimal than the current heap structure. In either case, the detection and recovery from failures need to be specified and optimized, which require some future work.

10.2.2 Dynamic node utility

Information that is used to calculate a node’s utility is currently exchanged when it joins a conference. This implies that the utility value is static. This is not optimal, since the underlying systems and networks are constantly changing. A node may move from one network to the other, and the underlying hardware can be used by other applications. Furthermore, the network between one pair of nodes can be completely different to another. To deal with this complexity, the protocol should include a mechanism that continually benchmarks the performance of the nodes and the links between them, and use that information to optimize the ordering. Some of that information could be collected using the WebRTC GetStats API that was used for data collection in the tests.
11 Conclusion

The project set out to solve the scalability problems of video conferencing services, by designing a protocol for decentralized conferencing with WebRTC. In addition to improved scalability, the decentralized approach and the use of WebRTC was chosen to decrease the cost of service deployment and maintenance, by not depending on dedicated hardware or complex server infrastructure.

The purpose of the report was to provide insight into the problems of WebRTC video conferencing, and to show how the protocol was developed to solve those problems. The goal was for the protocol to be more optimized for both Conference Scalability and Service Scalability, compared to the centralized alternatives and the decentralized full-mesh model. The problems found were mainly:

- **Media encoding** — Video encoding is done almost entirely in software, which means it has to be performed on the CPU. The machines used in the evaluation did not have any hardware that was optimized for the video codecs used by WebRTC in Google Chrome. They lacked the CPU resources to encode multiple high-quality video streams for multiple receivers, which caused a degradation in video quality.

- **Media transfer** — Sending a high-quality media stream over one connection required approximately 3 Mbit/s bandwidth. This becomes a problem as the number of connections grow, especially if nodes are located in mobile networks. Furthermore, the bandwidth requirements for the TURN servers increases significantly with the number of users.

The protocol uses two techniques to solve these problems:

- **Voice-activated switching** — The protocol selects one node to be the active speaker. This node is responsible for encoding and sending high-quality media to the other participants. Every other node sends a low-quality stream, which requires less computation and bandwidth.

- **Load-balancing using relays** — The nodes form a binary tree, where the active speaker is placed at the root, and the others are placed such that high-utility nodes are closer to the top. The high-quality stream is sent from the active speaker to its children, which in turn relays it to their children until everyone receives the stream. The low-quality streams are sent over a full-mesh overlay network.

The voice-activated switching model was implemented and tested, and the results showed some significant improvements over the full-mesh model. There were however still some problems related to load-imbalance, which were meant to be solved with the tree-structured relays.

Although the relay-tree model was specified and added to the protocol, it was not completely implemented and thus could not be evaluated in the testing environment. In addition to implementing the model, there are some proposals for future work:

- Optimize the utility function used to order nodes in the tree. Research the available web APIs that can be used to gather information about network, hardware and encoding capabilities. Alternatively find a way to continually benchmark all the nodes and the links between them.

- Find the optimal way to recover from node and link failures in the tree. Alternatively find another data structure that is more resilient to failures.
In conclusion: Even though some problems remain to be solved before the resulting protocol can be used in practice, the project was a success. Based on the evaluation and confidence in the model, it is plausible that the protocol, with some modifications, can be used to provide a completely decentralized and scalable conferencing service. It has no single point of failure, which means that errors can occur without any significant impact on service availability. The use of WebRTC ensures that all media is cryptographically protected and transferred directly between peers, which helps protecting the integrity and privacy of the user. It also lets the service can be consumed without installing any software other than a modern web browser.

It does not require any server infrastructure or hardware, aside from that of any application built with WebRTC; namely servers for signaling and NAT traversal. This simplifies the deployment and maintenance, which should reduce the costs for the service provider, and by extension its customers.
A Distributed Systems Abstractions

Module 6: Interface and properties of perfect point to point links [12, p. 74].

<table>
<thead>
<tr>
<th>Module</th>
<th>Name: PerfectPointToPointLink, instance pl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events:</td>
<td><strong>Request</strong>: (\langle pl, \text{Send} \mid q, m \rangle): Requests to send message (m) to the process (q).</td>
</tr>
<tr>
<td></td>
<td><strong>Indication</strong>: (\langle pl, \text{Deliver} \mid p, m \rangle): Delivers message (m) sent by process (p).</td>
</tr>
<tr>
<td>Properties:</td>
<td><strong>PL1</strong>: Reliable delivery: If a correct process (p) sends a message (m) to a correct process (q), then (q) eventually delivers (m).</td>
</tr>
<tr>
<td></td>
<td><strong>PL2</strong>: No duplication: No message is delivered by a process more than once.</td>
</tr>
<tr>
<td></td>
<td><strong>PL3</strong>: No creation: If the process (q) delivers a message (m) with sender (p), then (m) was previously sent to (q) by process (p).</td>
</tr>
</tbody>
</table>

Module 7: Interface and properties of FIFO-ordered perfect point to point links [12, p. 114].

<table>
<thead>
<tr>
<th>Module</th>
<th>Name: FIFOPerfectPointToPointLink, instance fpl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events:</td>
<td><strong>Request</strong>: (\langle fpl, \text{Send} \mid q, m \rangle): Requests to send message (m) to the process (q).</td>
</tr>
<tr>
<td></td>
<td><strong>Indication</strong>: (\langle fpl, \text{Deliver} \mid p, m \rangle): Delivers message (m) sent by process (p).</td>
</tr>
<tr>
<td>Properties:</td>
<td><strong>FPL1-FPL3</strong>: Same as PL1-PL3 of perfect point-to-point links, Module 6.</td>
</tr>
<tr>
<td></td>
<td><strong>FPL4</strong>: FIFO delivery: If some process sends message (m_1) before it sends message (m_2), then no correct process delivers (m_2) unless it has delivered (m_1).</td>
</tr>
</tbody>
</table>

Module 8: Interface and properties of the eventually perfect failure detector [12, p. 100].

<table>
<thead>
<tr>
<th>Module</th>
<th>Name: EventuallyPerfectFailureDetector, instance ♦P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events:</td>
<td><strong>Indication</strong>: (\langle ♦P, \text{Suspect} \mid p \rangle): Notifies that process (p) is suspected to have crashed.</td>
</tr>
<tr>
<td></td>
<td><strong>Indication</strong>: (\langle ♦P, \text{Restore} \mid p \rangle): Notifies that process (p) is not suspected anymore.</td>
</tr>
<tr>
<td>Properties:</td>
<td><strong>EPFD1</strong>: Strong completeness: Eventually, every process that crashes is permanently suspected by every correct process.</td>
</tr>
<tr>
<td></td>
<td><strong>EPFD2</strong>: Eventual strong accuracy: Eventually, no correct process is suspected by any correct process.</td>
</tr>
</tbody>
</table>

Module 9: Interface and properties of the eventual leader detector [12, p. 100].

<table>
<thead>
<tr>
<th>Module</th>
<th>Name: EventualLeaderDetector, instance Ω</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events:</td>
<td><strong>Indication</strong>: (\langle Ω, \text{Trust} \mid p \rangle): Indicates that process (p) is trusted to be leader.</td>
</tr>
<tr>
<td>Properties:</td>
<td><strong>ELD1</strong>: Eventual accuracy: There is a time after which every correct process trusts some correct process.</td>
</tr>
<tr>
<td></td>
<td><strong>ELD2</strong>: Eventual agreement: There is a time after which no two processes trust different correct processes.</td>
</tr>
</tbody>
</table>
Module 10: Interface and properties of best-effort broadcast [12, p. 124].

Module: BestEffortBroadcast, instance beb
Events:
- **Request**: \( \langle \text{beb, Broadcast } | m \rangle \): Broadcast a message \( m \) to all processes.
- **Indication**: \( \langle \text{beb, Deliver } | p, m \rangle \): Delivers a message \( m \) by process \( p \).
Properties:
- **BEB1**: Validity: If a correct process broadcasts a message \( m \), then every correct process eventually delivers \( m \).
- **BEB2**: No duplication: No message is delivered more than once.
- **BEB3**: No creation: If a process delivers a message \( m \) with a sender \( s \), then \( m \) was previously broadcast by a process \( s \).

Module 11: Interface and properties of reliable broadcast [12, p. 127].

Module: ReliableBroadcast, instance rb
Events:
- **Request**: \( \langle \text{rb, Broadcast } | m \rangle \): Broadcast a message \( m \) to all processes.
- **Indication**: \( \langle \text{rb, Deliver } | p, m \rangle \): Delivers a message \( m \) by process \( p \).
Properties:
- **RB1-RB3**: Same as BEB1-BEB3 of best-effort broadcast, Module 10.
- **RB4**: Agreement: If a message \( m \) is delivered by some correct process, then \( m \) is eventually delivered by every correct process.
B Measurements from full-mesh conference with two participants

Figure 41: CPU usage in a 2-party full-mesh conference.

Figure 42: Network usage (sent, upload) in a 2-party full-mesh conference.

Figure 43: Network usage (received, downloaded) in a 2-party full-mesh conference.
Figure 44: Frame rate of video streams in a 2-party full-mesh conference.

Figure 45: Resolution video streams in a 2-party full-mesh conference.
C Voice-activated switching protocol

Algorithm 12: Low latency signaling channel

Implements:
LowLatencySignalingChannel, instance sc.

Uses:
RTCDATAChannel, instance dc, Module 6.
Signaling, instance sig, Module 4.

upon event ⟨sc, Send | q, m⟩ do
  if channels.has(q) then
    trigger⟨dc, Send | p, m⟩;
  else
    trigger⟨sig, Send | p, m⟩;
  end

upon event ⟨sc, Broadcast | m⟩ do
  trigger⟨sig, Broadcast | m⟩;

upon event ⟨dc, Deliver | q, m⟩ do
  trigger⟨sc, MessageDeliver | p, m⟩;

upon event ⟨sig, MessageDeliver | q, m⟩ do
  trigger⟨sc, MessageDeliver | p, m⟩;

upon event ⟨sig, BroadcastDeliver | q, m⟩ do
  trigger⟨sc, BroadcastDeliver | p, m⟩;
Algorithm 13: Voice Analyser, selecting active speaker.

**Implements:**
VoiceAnalyser, instance `va`.

**Uses:**
EventualLeaderDetector, instance `eld`, Module 9.
EventuallyPerfectFailureDetector, instance `♦P`, Module 8.

```plaintext
upon event ⟨va, Init⟩ do
    suspected := ∅;
    leader := null;

upon event ⟨eld, Trust | p⟩ do
    leader := p;

upon event ⟨♦P, Suspect | p⟩ do
    suspected := {p} ∪ suspected;

upon event ⟨♦P, Restore | p⟩ do
    suspected := suspected \ {p};

upon event ⟨Timeout⟩ do
    if leader = self then
        unsuspected := Π \ suspected;
        audioLevels := getAudioLevelsFromPeerConnections(unsuspected);
        activeSpeaker := maxRank(audioLevels);
        trigger ⟨va, Active | activeSpeaker⟩;
    end
```

61
Algorithm 14: Additions to the Conference protocol, for voice-activated switching.

**Implements:**
Conference, instance `conf`, Module 3.

**Uses:**
LowLatencySignalingChannel, instance `sc`, Module 12.
VoiceAnalyser, instance `va`, Algorithm 13.

```plaintext
upon event ⟨conf, Init⟩ do
  connections := $\emptyset$;
  hqLocalStream := null;
  lqLocalStream := null;
  activeSpeaker := null;

upon event ⟨va, Active | p⟩ do
  trigger ⟨sc, Broadcast | [ACTIVESPEAKER], p⟩;

upon event ⟨sc, BroadcastDeliver | [ACTIVESPEAKER], p⟩ do
  oldActive := activeSpeaker;
  activeSpeaker := p;
  if activeSpeaker $\neq$ self then
    // Request HQ stream from activeSpeaker
    // Request LQ stream from oldActive
  end

upon event ⟨conf, Join⟩ do
  hqLocalStream := getUserMedia([HQ]);
  lqLocalStream := getUserMedia([LQ]);
  trigger ⟨conf, Stream | self, hqLocalStream⟩;
  trigger ⟨sc, Broadcast | [JOIN]⟩;

upon event ⟨sc, BroadcastDeliver | p, [JOIN]⟩ do
  // Same as Full-mesh conference. Except:
  if p $<$ self then
    trigger ⟨sc, Send | p, [OFFER]⟩;
  else
    trigger ⟨sc, Send | p, [OFFERME]⟩;
  end
```

62
Algorithm 15: Additions to the Conference protocol, for tree relay load balancing.

Implements:
Conference, instance `conf`, Module 3.

Uses:
LowLatencySignalingChannel, instance `sc`, Module 12.
VoiceAnalyser, instance `va`, Algorithm 13.

**upon event** `⟨conf, Init⟩` **do**

utility := `U`(`self`);
connections := ∅;
activeSpeaker := `null`;
parent := `null`;
children := ∅;

**upon event** `⟨va, Active | p⟩` **do**

`tree` := `sortAndBuildTree`(`connections`, `activeSpeaker`);
`trigger` `⟨sc, Broadcast | [ACTIVESPEAKER], p, tree⟩`;

**upon event** `⟨sc, BroadcastDeliver | [ACTIVESPEAKER], p, tree⟩` **do**

// ...
parent := `parent`(`tree`, `self`);
children := `children`(`tree`, `self`);
parent.onaddstream(`stream`) = `children.addStream`(`stream`);
// ...

**upon event** `⟨conf, Join⟩` **do**

// ...
`trigger` `⟨sc, Broadcast | [JOIN], utility⟩`;

**upon event** `⟨sc, BroadcastDeliver | p, [JOIN], u⟩` **do**

// ...
connections[p].utility := `u`;
// ...

63
class Tree {
    //Construct a tree, given an array of nodes and a root.
    constructor(nodes, root) {
        this.nodes = nodes.sort(/* Sort based on utility */);
        this.nodes.unshift(root); //Place root at index 0
    }

    //Return the root of the tree.
    root() {
        return this.nodes[0];
    }

    //Return the parent of a node.
    parent(child) {
        let childIndex = this.nodes.indexOf(child);
        let parentIndex = (Math.floor((childIndex - 1) / 2));
        return this.nodes[parentIndex];
    }

    //Return the left and right children of a node.
    children(parent) {
        let parentIndex = this.nodes.indexOf(parent);
        let leftChildIndex = 2 * parentIndex + 1;
        let rightChildIndex = leftChildIndex + 1;

        return {
            left: this.nodes[leftChildIndex],
            right: this.nodes[rightChildIndex]
        };
    }

    //Serialize the tree
    serialize() {
        return this.nodes;
    }
}

Figure 46: Sort and build a tree, from a list of nodes.
References


[34] Docker. URL https://www.docker.com/.


[38] Prometheus. URL http://prometheus.io/.


