NTNU

Norwegian University of
Science and Technology

Dashboard for Quality of Experience Studies of WebRTC Based Video Communication

Marianne Rie Melhoos

Master of Science in Communication Technology
Submission date: June 2016
Supervisor: Poul Einar Heegaard, ITEM
Co-supervisor: Doreid Ammar, ITEM
Katrien De Moor, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title:	Dashboard for Quality of Experience Studies of WebRTC Based Video Communication
Student:	Marianne Rie Melhoos

Problem description:

Internet video applications and services are taking up an ever increasing share of the Consumer Internet traffic. Their success and use are strongly influenced by the quality they provide and the experiences they make possible for users. At the same time, however, the delivered quality and the experience for the user of video applications and services may be very negatively influenced by technical constraints (e.g., limited bandwidth) and parameters (e.g., packet loss, delay, etc.).

In this project, the focus is on users' experiences with Web Real-Time Communication (WebRTC)-based video communication services, such as appear.in or Google hangouts. The main objective is to develop a web interface tool that illustrates, in a graphical and interactive way, the most relevant factors that impact the quality of WebRTC-based video communication, with a primary focus on the performance of the network(s) over which the conversation is transmitted.

The main tasks include:

- Briefly overview the most relevant Quality of Service (QoS) and Quality of Experience (QoE) factors in the context of WebRTC-based real-time video communication.
- Development of a web interface allowing to illustrate the most relevant factors that impact the performance of WebRTC-based real-time video communication.

Responsible professor:	Poul Heegaard
Supervisor:	Doreid Ammar, Katrien De Moor

Abstract

WebRTC-based applications and services have become more and more popular over the last years. These types of applications support Real-Time Communication (RTC) with audio, video and sometimes also data sharing. WebRTC-based applications are trouble-free, require no installation, and are in-browser applications. A participant can connect and access these types of applications through a wide range of devices and can communicate with others in real-time, and exchange information instantly or with negligible delay.

Even though the performance of WebRTC-based applications and services are continuously improving, these applications and services face some challenges. For example, in a WebRTC-based conversation, there are a number of technical constraints (e.g. limited bandwidth) and parameters (e.g. packet loss, delay, etc.) that may cause an end-user to experience various negative quality deteriorations (e.g. video freezes, bad or no audio, etc.). As WebRTC-based applications and services are up and coming, it is important to address such issues.

To provide the best possible Quality of Experience (QoE), this requires lab studies and also large-scale ongoing lab studies in order to obtain a deep understanding of the various technical and non-technical factors that may have an influence on the QoE [44]. Due to the high number of session-related parameters, these studies will provide a significant amount of data, all of which will need to be analyzed. Therefore in order to proceed further analysis, an analyzing tool is crucially needed.

To meet this need to provide an analyzing tool which is able to analyze large amounts data, this master thesis will present the implementation of a web interface, the WebRTC-dashboard. The WebRTC-dashboard utilizes session-related data from analyzing platforms in order to analyze n-party WebRTC-based video conversations. The WebRTC-dashboard supports to combine network statistics, subjective user feedback from different analytic platforms, and video recordings with the ability to replay them as they were in real-time. This WebRTC-dashboard also allows end-users to interact and customize an analysis for his/hers purpose, which opens the possibility to identify new correlations between various impacting factors.

This master thesis highlights the great potential the WebRTC-dashboard has and what it can accomplish. In the context of future development, this thesis will also discuss what challenges that have occurred during the development process, and how these challenges can be handled.

Sammendrag

I løpet av de siste årene har applikasjoner og tjenester som baserer seg på sanntidskommunikasjons over web, også kjent som WebRTC, blitt meget populære. Disse applikasjonene tilbyr kommunikasjon i sanntid ved bruk av både lyd og bilde, og i enkelte tilfeller også fildeling. Applikasjoner som er basert på WebRTC er tilgjengelig direkte i nettleseren, og kjent for å være enkle og installasjonsfrie. Disse applikasjonene kan enkelt benyttes ved hjelp av nettleseren i en rekke forskjellige enheter, og informasjon kan deles nærmest umiddelbart (eventuelt med en ubetydelig forsinkelse) med andre enheter.

Selv om WebRTC-teknologien er i kontinuerlig utvikling, står den fremdeles foran en rekke utfordringer. Ta for eksempel en ordinær samtale som benytter WebRTC, hvor kvaliteten på samtalen fra sluttbrukernes perspektiv kan bli redusert av pakketap, forsinkelser og begrenset båndbredde. Siden WebRTC-teknologien virkelig er i vinden for tiden og flere tjenester blomstrer opp, er det viktig at disse problemene blir adressert for å heve brukeropplevelsen for sluttbrukerne.

For å kunne levere en WebRTC-tjeneste av høyest mulig kvalitet, er det viktig å besitte god innsikt i hvordan de tekniske og ikke-tekniske faktorene påvirker brukeropplevelsen. Disse faktorene må, for øvrig, observeres og analyseres gjennom forskning. I en vanlig WebRTC-samtale generes store mengder rådata. For at denne dataen skal kunne analyseres og brukes i videre forskning, er det avgjørende å ha et analyseverktøy som effektivt kan behandle, analysere og presentere rådataen på en informerende måte.

Denne masteroppgaven presenterer et web-basert grensesnitt, også kalt et WebRTC-dashboard, som benytter rådata fra WebRTC-samtaler mellom n deltakere til å analysere tjenestens brukeropplevelse. Dashbordet som blir presentert gjennom denne avhandlingen muliggjør det å spille av tidligere samtaler som om de skulle være avspilt i sanntid. Dette blir gjort ved å kombinerer nettverksstatistikk, tilbakemeldinger fra sluttbrukere som blir hentet via to forskjellige analytiske plattformer og videoopptak av samtaler. I tillegg kan sluttbrukeren av tjenesten tilpasse analysen til eget formål gjennom en rekke innstillinger. Dette kan identifisere tidligere ukjente korrelasjoner mellom tekniske og ikke-tekniske faktorer som påvirker brukeropplevelsen.

Videre vil denne avhandlingen kaste lys over potensialet ved et slikt WebRTC-dashbord, og hva et slikt verktøy kan utrette for leverandører av WebRTC-tjenester. Masteroppgaven diskuterer også de ulike utfordringene som har oppstått under utviklingen av verktøyet, hvordan disse har blitt håndtert og hva slags begrensninger verktøyet har.

Preface

This master thesis is an original and independent work by Marianne Rie Melhoos. The thesis is the final contribution to the Master's degree in Communication Technology at the Norwegian University of Science and Technology (NTNU).

The primary goal of this master thesis is to develop a web interface, which illustrates the most relevant factors that impact the performance of WebRTC-based video communication. In addition, an objective of this master thesis is to provide a brief overview of the most relevant Quality of Service (QoS) and QoE factors in the context of WebRTC-based video communication.

I want to thank my supervising professor at Department of Telematics (ITEM), Poul Einar Heegaard, for providing guidance during the preparation of this master thesis, and to Doreid Ammar for providing with useful guidance, information regarding WebRTC and appear.in, as well as for showing interest in this master thesis. I would also like to express my gratitude to Katrien De Moor for giving me feedback in respect to the WebRTC-dashboard.

Also, I would like to extend my gratitude to Arthur Melhoos, my father, for both giving feedback and reading through this master thesis.

Lastly, to the wonderful people I have been sharing an office with during the last year, thank you for listening.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface.

Bash Bourne-Again Shell.

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart.

CPU Central Processing Unit.

CSS Cascading Style Sheets.

D3 Data-Driven Documents.

GNU GNU's not Unix.

GUI Graphical User Interface.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

ID Identification.

IEEE Institute of Electrical and Electronics Engineers.

IETF Internet Engineering Task Force.

ITEM Department of Telematics.

ITU International Telecommunication Union.

JS JavaScript.

JSON JavaScript Object Notation.

KBAC Knowledge-Based Admission Control.

mp4 mpeg 4.

MVC Model View Controller.

NPM Node Package Manager.

NTNU Norwegian University of Science and Technology.

OS Operating System.

P2P Peer-to-Peer.

PC Personal Computer.

PHP PHP: Hypertext Preprocessor.

PLI Picture Loss Indication.

QoE Quality of Experience.

QoS Quality of Service.

RTC Real-Time Communication.

Sass Syntactically Awesome Stylesheets.

SMS Short Message Service.

SQL Structured Query Language.

SRS Software Requirements Specification.

TFS Team Foundation Server.

txt text.

UI User Interface.

URL Uniform Resource Locator.

VoIP Voice over IP.

W3C World Wide Web Consortium.

WebRTC Web Real-Time Communication.

WWW World Wide Web.

Chapter 1
Introduction

1.1 Motivation

Applications and services enabling Real-Time Communication (RTC) have gained wide popularity over the last years. Some of the known synchronous, video- and audio-mediated services today are Skype [31], Facebook Messenger's video calling feature [22], and FaceTime [30]. They support RTC with audio and video sharing, and Skype and Facebook Messenger's video calling feature also support data sharing. One person can view and communicate with another person in real time, and exchange information instantly or with negligible delay. Some applications also support group conversations, where up to ten persons can communicate at the same time [31]. On the other hand, these services and applications require plug-ins, log in, or a particular device (for example FaceTime requires a device running OS X) to be able to access these applications and services.

Compared to Skype, FaceTime, and other similar services, the Web Real-Time Communication (WebRTC)-based applications are trouble-free, require no installation, and are in-browser applications. Google Hangouts [15] and appear.in [5] are examples of WebRTC-based applications. The WebRTC provides the possibility for the WebRTC-based applications to utilize its service without requesting their users to create user accounts, login or remember passwords. For example, appear.in does not require any user login. However, Google Hangouts does. One can simply connect and access the application through a wide range of devices, such as smartphones, tablets, and laptops. These devices must use a web browser that supports WebRTC (e.g. Google Chrome, Mozilla Firefox, and Opera) or a dedicated application. Also, WebRTC communication includes an open sourced Application Programming Interface (API) that enables developers to implement their RTC-based application in their web browser [36].

Even though the performance of WebRTC-based applications and services are continuously improving, these applications and services face some challenges. Both appear.in and Google Hangouts allow users to use various types of devices and be in different environments when participating in multi-party video conversations. Consequently, a participant may perceive the conversation differently than other participants [45]. For example, two out of three individuals may have ideal conditions for a positive and pleasurable QoE, yet still, their actual overall QoE might not be as good because one of the participants is suffering from a weak mobile network. Additionally, during a real-time video- and audio-mediated conversation there are several technical constraints (e.g. limited bandwidth) and parameters (e.g. packet loss, delay, etc.) that may cause the user to experience various negative quality deteriorations (e.g. video freezes, bad or no audio, etc.).

As WebRTC is evolving as a new supporting technology, it is important to address these challenges. From a developer's point of view, it is important to address these facts in order to prevent users from getting displeased and stop using their services and switching to another competitor. To prevent users from getting frustrated and to provide the best possible QoE requires a deep understanding of the various technical and non-technical factors that may have an influence on the QoE [44].

To face these challenges, lab studies, and large-scale ongoing lab studies are required. With numerous technical and non-technical parameters to choose from, these studies will provide a significant amount of data, all of which needs to be analyzed. Currently, there exist analytic platforms that are able to retrieve data (QoS, QoE, and non-technical parameters) from WebRTC-based applications, measuring them, and organizing them. However, since the data collected from analytic platforms usually are in JavaScript Object Notation (JSON) format, it is difficult to regain any information from the data in plain text. Therefore to enable further analysis and to identify the factors impacting the QoE, an analyzing tool is crucially needed.

In order to meet the need of creating an analyzing tool, this master thesis will present the implementation of a web interface. The web interface illustrates, in a graphical and interactive way, the most relevant factors that impact the performance of WebRTC-based communication. This web interface, also referred to as the WebRTC-dashboard, helps to give a deeper insight into finding a correlation between the technical and non-technical factors that influence the QoE. Also, this WebRTC-dashboard includes video and audio recordings to highlight video freezes, audio loss, etc., as well as subjective user feedback. Finally, this web interface supports statistics for not only two-party conversations but for n-party conversations. Accordingly, this allows for a better understanding of how one individual can affect the overall QoE in a multi-party conversation, which has not been studied to a great extent to date.

1.2 Objectives

The project description has two primary objectives for this master thesis:

- Briefly, give an overview of the most relevant QoS and QoE factors in the context of WebRTC-based real-time video communication.
- Development of a web interface that illustrates the most relevant factors that impact the performance of WebRTC-based real-time video communication.

1.3 Scope

Since there exist multiple WebRTC-based applications, this master thesis narrows its scope down to one WebRTC-based application. This master thesis focuses on implementing a WebRTC-dashboard which focuses on analyzing session-related data generated from the WebRTC-based application appear.in. Even though, this scope is limited, for future implementation, the web interface should be able to support to analyze session-related data from other WebRTC-based applications.

1.4 Methodology

The methodology used to meet this master thesis' objectives is divided into two processes: development and literature study. These processes are described in the following two sections.

1.4.1 Development

The development process covers implementation of the web interface and comprises the main part of this master thesis, and is divided into four parts.

- **System definitions:** First, the system definitions were defined at the beginning of the development process. The system definitions cover the division of the system architecture, technologies to be used, and the Software Requirements Specification (SRS).
- **Implementation:** Second, the implementation process covers the actual development of the web interface. The implementation is the main part of the development process, and also the most time consuming of all parts.
- **Testing:** Third, simultaneously along with the implementation, testing was conducted. Testing was used to ensure that the web interface satisfies both functional and non-functional requirements.
- **Feedback:** Fourth, likewise the testing part, along the implementation process the feedback was received from both the responsible professor and supervisors.

The feedback had a significant influence on the Graphical User Interface (GUI), functionalities, and how to improve the web interface.

1.4.2 Literature Study

A literature study has been conducted to research the topics in the project description and to decide the appropriate technologies for the WebRTC-dashboard. The literature study includes studying WebRTC, the most relevant QoS and QoE factors in the context of the WebRTC-based video communication, and the technologies that suit the purpose of the web interface.

1.5 Outline

This master thesis is divided into six chapters, the topics of each chapter are as follows:

- **Chapter 1, Introduction:** The introduction chapter introduces the motivation and objectives for this master thesis as well as the scope and methodology.
- **Chapter 2, Theoretical Background:** The theoretical background chapter gives a brief introduction to WebRTC, the definition and influencing factors affecting the QoE and QoS, the challenges with WebRTC applications, and the related work.
- **Chapter 3, Overview of the WebRTC-dashboard:** The overview of WebRTC-dashboard chapter presents a brief introduction to the WebRTC-dashboard, followed by the WebRTC-dashboard's Software Requirements Specification (SRS), and the analytic platforms the WebRTC-dashboard uses to retrieve WebRTC-based session-related data.
- **Chapter 4, Development of the WebRTC-dashboard:** The development of the WebRTC-dashboard chapter includes how the WebRTC-dashboard is implemented. It contains system architecture, testing, technologies, and the challenges that occurred during the implementation process, and how they were handled.
- **Chapter 5, Description of the WebRTC-dashboard:** The description of the WebRTC-dashboard chapter gives a description of the functionalities and limitations of the WebRTC-dashboard.
- **Chapter 6, Conclusion and Future work:** Conclusion and future work chapter include the conclusion and future work in respect to the WebRTC-dashboard. Future work covers suggestions on how to remedy some of the WebRTC-dashboard's limitations and some nice-to-have features.

Chapter 2
Theoretical Background

This chapter presents the relevant theory evaluated in the literature study, and gives the background information needed for this master thesis. This chapter includes a brief introduction of the WebRTC, current challenges with WebRTC applications, definitions of QoS and QoE, factors which influence QoE, and related work.

2.1 WebRTC

WebRTC is a free, open source framework defined by the World Wide Web Consortium (W3C)[1] that enables direct browser-to-browser (Peer-to-Peer (P2P)) RTC [36]. WebRTC-based applications and services do not require any additional software installations (such as plug-ins) and are in-browser applications. Examples of WebRTC-based applications are appear.in (will be presented in Section 2.1.2), Google Hangouts (will be presented in Section 2.1.3), and Telefónica's Hello (will be presented in Section 2.1.4). Since the WebRTC-based applications and services use the browser as its platform, these services are easily accessed.

Besides WebRTC, there is no other free, high quality, complete solution that can enable RTC [36]. Until the launch of WebRTC, RTC technology had only been available to large established companies which could afford the expensive licensing fees [40]. On the account of that WebRTC is not a finished standard as yet, only Google Chrome, Mozilla Firefox and most recently Opera are the only web browsers that have fully enabled the WebRTC standard. Other Operating System (OS)-provided browsers (Internet Explorer and Safari) have noticeably not implemented WebRTC yet. Considering this fact, the Internet Engineering Task Force (IETF)[2] and the WebRTC Working Group[3] are currently working on a WebRTC standard which is expected to become widespread as soon as their work is completed [38].

[1] The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web (WWW).

[2] Internet Engineering Task Force (IETF) develops and promotes voluntary Internet standards.

[3] The WebRTC Working Group defines client-side APIs to enable RTC in web browsers.

2.1.1 WebRTC API

WebRTC's framework allows developers to implement their WebRTC-based applications through simple APIs. These APIs include the fundamental components needed to build a high-quality RTC-based web application. These fundamental components provide audio, video, and non-media data packets performance statistics that are transmitted over peer-connections in WebRTC services [46]. A WebRTC framework contains three APIs which are needed to fully establish a RTC connection [37]. Each of these APIs is responsible for creating, continuing and closing the RTC process, and are summarized as follows:

- **GetUserData API:** The GetUserData API is responsible for accessing the user's microphones and video cameras, capturing audio and video data and transferring data through the peer-connections [37].
- **PeerConnections API:** The PeerConnections API is responsible for creating P2P connections to transmit and receive audio, video, and non-media data from one browser to another [37]. These PeerConnections hold *MediaStreams*, which are responsible for sending media (audio, video, and screen sharing) content. Each *MediaStream* contains two *tracks*, one for sending and one for receiving [46].
- **DataChannel API:** The DataChannel API is in charge of sending non-media data (such as text chat, file transfer, etc.) through the peer-connections [37].

2.1.2 appear.in

appear.in is a WebRTC-based free browser-to-browser service. It does not require any user registrations or software installation (such as add-ons to your web browser). Since WebRTC is not standardized as yet, appear.in is only accessible on certain web browser versions that support WebRTC (e.g. Google Chrome, Mozilla Firefox, and Opera).

appear.in supports service to a multi-party conversation up to eight participants, through a Uniform Resource Locator (URL), which appear.in utilizes to create a virtual room. Additional functionalities supported by appear.in are text messaging, mute, camera off and on, adjustment of the size of the video screen, and lock and leave the virtual room.

Even though there are several other WebRTC-based applications and services, such as Google Hangouts, Hello, webEx [35], and AnyMeeting [4], this master thesis focuses on analyzing session-related data generated from appear.in conversations. The reason for this is that appear.in provides an additional custom statistic interface called getstats.io, which is presented later in Section 3.3.2.

2.1.3 Google Hangouts

Google Hangouts is a WebRTC-based application, which is a RTC platform developed by Google. Google Hangouts supports a multi-party conversation up to ten participants and includes instant messaging, video, Short Message Service (SMS), and Voice over IP (VoIP) features. Even though it is free, it requires a Google account to access all the features (such as text messaging) [15].

2.1.4 Firefox Hello

Hello is Mozilla Firefox's WebRTC solution, and this service is developed in collaboration with Telefónica[4]. Similar to, appear.in, *Hello* does not require user registrations or additional software downloads. It provides substantially the same features as appear.in, however, *Hello* is only accessible through the Mozilla Firefox web browser.

2.2 Challenges with WebRTC Applications

appear.in, Google Hangouts and *Hello* are a few of many WebRTC-based applications. These and other WebRTC-based applications offer a broad range of video chat and conferencing services. Applications and services enabling audio- and video-mediated RTC have become more favored over the past years [45]. Even though the performance of these services and applications are continuously improving, it remains challenging to satisfy the end-user's QoE demands, at all times, and in all circumstances. This section discusses some of the challenges WebRTC-based applications and services are currently facing.

First of all, multi-party WebRTC conversations usually take place with a certain amount of technical asymmetry. For each of the participants, the technical conditions may vary (e.g. different device, network, connection, etc.). These conditions can have a great impact of the overall QoE, which means that if at least one party has a weak network link, it will affect the overall QoE of all the others included in the same conversation [45].

Second, there is always some inherent and unavoidable delay in WebRTC-based applications and services. When audio- and video-mediated data is sent from a sender to a receiver, it first must be recorded, prepared, encoded for transmission, transmitted over a network, decoded by the receiving device, and finally presented to the receiver [60]. During each of these steps, additional delay and technical artifacts can appear. These artifacts may impact the final QoE, such as audio and video quality, and synchronization between the parties [45].

[4]Telefónica is a Spanish broadband and telecommunications provider [32].

Third, WebRTC allows developers to create their WebRTC-based service or application via WebRTC API described in Section 2.1.1. If one desires to build a social-awareness system, which satisfies the end-users, then one of the challenges is to obtain a thorough understanding of which factors that impact the QoE and how to measure them [54].

Last and finally, in order to evaluate the QoE in WebRTC-based applications and services one must identify the factors that impact the QoE. There are several factors known to affect the QoE, such as QoS parameters, the user, the system, and the context of the conversation. All of these factors will be discussed in Section 2.4. Since there exists numerous QoS parameters (packet loss, delay, etc.), subjective factors (self-reported feedback from the end-user), and objective factors (based on externally observable feedback), it is a challenge to identify the factors that impact the QoE in WebRTC-based applications and services. Also, not all of the factors are easily measured [61].

To address the issues presented in the previous sections, this master thesis reviews the implementation of a web interface that helps to identify a correlation between QoE and other factors (QoS parameters, user, and context). Additionally, this web interface will present these factors in a graphical and interactive way to give the user a thorough understanding of the different factors. However, it is important to highlight that this web interface does not solve the problem of improving the QoE in WebRTC-based applications and services, but on the other hand contributes to obtaining a deeper insight in finding the correlation between technical and non-technical factors influencing QoE.

2.3 Definition of QoS and QoE

There are several different perspectives of how to measure and define QoS and QoE. The following section will present various perspectives for evaluating QoS and QoE related to WebRTC.

2.3.1 Quality of Service

Typically, QoS is measured by evaluating the performance of the service itself. To measure the performance of a service is done by looking at the delivery network capacity and resource availability [57]. The performance of a service is how good the quality of the service has when the service at its best and how often the service fails. There are several definitions of QoS, but the definition as stated by International Telecommunication Union (ITU)[5] is as follows:

[5]International Telecommunication Union (ITU) is a specialized agency of the United Nations that is responsible for issues that concern information and communication technologies.

Quality of Service (QoS): "Totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service" [55].

The definition stated by ITU relates to *satisfying the needs of the user of the service*, which only refers to the users using the system, and not the system itself. However, another definition stated by [52] refers to *satisfying the different entities within the system*, in addition to the users. In the definition stated by [52], it is important to note that the *service* is not necessarily a physical interface, and the *user* is not necessarily an end-user of the service. For example, a transport protocol is the user of the service provided by a network protocol [52].

Quality of Service (QoS): "Degree of compliance of a service to the agreement that exists between the user and the provider of this service" [52].

In a computer network perspective, QoS relies on multiple network statistics, such as bandwidth, jitter, latency (required in real-time services), and loss characteristics [29]. Most services that are network connected use several sets of mechanisms to ensure QoS, such as: controlling the network resources to ensure that the services focus on the most significant traffic for the service's purpose [29].

While QoS weighs the actual service delivered, QoE measures the user-perspective experience of the service. Instead of measuring network statistics, QoE measures the users' preferences of the service. This kind of measurement tells how to create a service that the users' appreciate and as such increase the chances that the user will continue to use the service.

When evaluating QoE, one notices that QoS is closely related. QoE heavily depends on the technical aspect of service performance, and if QoS requirements are not fulfilled (e.g., minimum packet loss), then the probability of satisfying the QoE is small [57]. It is, therefore, relevant to take QoS into account when evaluating QoE of a system.

2.3.2 Quality of Experience

Compared to QoS, Quality of Experience (QoE) takes other parameters into account when measuring the performance of the system. QoE incorporates the conceivable influence of non-technical attributes such as user characteristics and context of the user [49]. Moreover, QoE is subjective and individual, which means that even though the QoS is the same, a user may not have the same QoE as other users. Currently, the most well-known definition of QoE as defined by the ITU, and is as follows:

Quality of Experience (QoE): "The overall acceptability of an application or service, as perceived subjectively by the end-user" [55].

The definition stated by ITU has, however, put QoS in a subjective perspective. According to this, the concept of the 'overall acceptability' for measuring QoE may be unclear [58]. Consequently, a new and more accurate definition was proposed by Qualinet[6] [28], which takes human-related factors into account:

"Quality of Experience (QoE) is the degree of delight or annoyance of the user of an application or service. It results from the fulfillment of his or her expectations on the utility and/or enjoyment of the application or service in the light of the user's personality and current state." [28].

In addition to system specific and context-related factors, Qualinet suggests a definition that takes the emotional state of the end-user into account when evaluating the QoE. It entails that the goal of QoE is no longer only about satisfying the expectations of the end-user related to the utility of an application or service [51], but also includes how the end-user feels, and how his/hers experiences with the service involve and stimulate people emotionally [59]. Compared with ITU's definition, the last definition highlights possible influencing human-related factors.

QoE is strictly subjective when referring it to the end user's feelings, expectations, personal relations, and motivations, etc. These are complicated and challenging to measure. Recent literature has, therefore, focused on what parameters influence the quality of the users' QoE when using WebRTC services [45, 54, 56, 61]. In the following section, these parameters will be presented and discussed.

2.4 Factors Influencing QoE

As mentioned in the previous paragraph, it is complex and challenging to measure QoE. There is no distinct, defined method to measure QoE especially when using the new definition from Qualinet [28]. Qualinet has, however, acknowledged that "creativity (Content), technology (Deliver and Interaction), market/finance (Business models), and users (Usages)" [28] factors influence QoE. Qualined defines an influencing factor as follows:

Influencing factor: "Any characteristic of a user, system, service, application, or context whose actual state or setting may have influence on the Quality of Experience for the user." [28].

[6]Qualinet is a European Network of Excellence of Quality of Experience (QoE) in Multimedia Systems and Services.

It is important to highlight, that there are not only QoS parameters that influence the QoE in respect to WebRTC. QoE is also influenced by several other parameters, such as user, and context-related factors. These influencing factors of QoE are presented below.

2.4.1 QoS Parameters

QoS parameters are identified as one of the factors influencing QoE. In the context of WebRTC, recent literature has focused on three factors within QoS that impact QoE, namely network conditions, application-level aspects, and type of hardware [45, 54].

Network conditions cover the quality of the network. For example, if the device used for participating a WebRTC-based service is connected to a wired or wireless network. Usually, a device connected to a wireless network has a higher probability of being disconnected from a conversation than a device connected to a wired network [61].

Application-level aspects include network parameters such as packetsloss, delay, jitter, bandwidth, etc. These parameters can have a great impact on the QoE. According to [47], synchronization between the audio and video is highly prioritized. It is better to delay the audio to ensure the synchronization, but only in the event, the end-to-end delay is below 600ms. If the end-to-end delay is greater than 800ms, it is considered unacceptable [54].

The type of hardware can have a great impact of if ever the user can participate in the WebRTC-based application or not. WebRTC-based services and applications require a significant amount of processing power [61], and if the connected device does not satisfy the processing requirements, then the end-user will not be able to connect to the service.

2.4.2 User

It is important to distinguish between a person and the role this person takes in a video conversation. On the personal level, it is important to consider the person's prior experience with the video conversation system, and what the person is expecting. While, on the role level, one must also take into account what kind of role the person has when participating in a video conversation [54]. For example, given some negative QoS parameter influencing factors (e.g. delay), a person with the role as a moderator of a conversation might not have the same experience as the other not-moderator participants during the same conversation.

Also, the purpose of the video conversation can cause the participants to experience a different QoE. A video conversation employed in a business context has higher

end-user expectations, then a conversation in a private context. For the most video conversation in a business context, the participants have a goal for the conversations, while for private conversations the primary objective is to experience a sense of presence or social connection [61].

2.4.3 Context

The context is established by the interplay between the user and the current situation [54], and can have an influence on the communication. It is important to address factors such as activities (lecture, discussion, free-play, etc.), the settings (home, school, etc.), and environment (light, noise, the number of people, etc.) of the conversation when evaluating the context of a WebRTC conversation. In the context of WebRTC, it is understandable that parameters such as background noise, and bad lighting can influence the QoE negatively.

2.5 Related Work

The WebRTC-dashboard focuses on finding correlations between the technical and non-technical factors that influence the end-users' QoE. Other similar solutions are Chrome's WebRTC internal interface, getstats.io and Knowledge-Based Admission Control (KBAC) [43]. Both Chrome's WebRTC internal interface and getstats.io supports to fully visualize the session-related statistics in real-time and downloads them to perform post-processing analysis, while KBAC is a dashboard, which focuses on admission control [43]. These solutions, however, do not support the same flexibility as the WebRTC-dashboard.

Research regarding the QoE in a multi-party WebRTC-based conversation has gained popularity the in recent years. Due to the multitude of potential influencing factors (QoS parameters, user, and context), the evaluation of quality and users' QoE in the context of the multi-party WebRTC-based conversation is challenging [44, 45].

Several studies have already been conducted to identify the influencing factors of the QoE in the context of a WebRTC-based conversation. One of them is [45], and this study investigates session-related performance statistics from a WebRTC-based application called appar.in and explores how they may relate to users' QoE. It also highlights the limitations of Google Chrome's WebRTC internal tool and focuses on identifying the factors that cause QoE killers (for example video freezes).

Moreover, [44] investigates performance statistics in order to detect potential QoE issues. It utilizes a multi-method approach to gain more insight in the relevant influencing factors and investigates the relationship between performance-related parameters and users' QoE. This is achieved by identifying the most relevant influencing

factors (both technical and non-technical), how the influencing factors influence the users' QoE and the corresponding user behavior, and understanding the relationship between them.

Another study, [61] investigated the QoE issues in the context of using mobile devices in multi-party WebRTC-based conversations. It considered different series of interactive, three-party WebRTC-based conversation with various smartphones and laptops. This research's results indicated that the end-users that used smartphones have lower expectations than the end-users using laptops. The author also highlighted that the many smartphones may not be able to meet the high Central Processing Unit (CPU) requirements needed to ensure a smooth QoE.

Lastly, a study which is necessary to mention is [46]. This study focuses on highlighting the limitations of the Google Chrome WebRTC internal tool, and how to overcome these issues. Even though Chrome statistics consist of multiple constraints, [46] believes that these statistics can be used QoE studies in respect to WebRTC services.

Chapter 3
Overview of the WebRTC-Dashboard

This chapter presents an overview of the WebRTC-dashboard. The overview of the WebRTC-dashboard chapter will first give a brief introduction to the WebRTC-dashboard, followed by the WebRTC-dashboard's Software Requirements Specification (SRS). After this, the next sections cover the platforms the WebRTC-dashboard uses to retrieve WebRTC-based session-related data.

3.1 WebRTC-Dashboard

WebRTC-dashboard is a web interface, and its purpose is to analyze WebRTC-based real-time conversations. To do so, the WebRTC-dashboard includes network parameters, device information, audio and video recordings, and user feedbacks. The WebRTC-dashboard gathers data from mainly two analytic platforms. These platforms are Google Chrome's WebRTC internal interface (presented further in Section 3.3.1) and getstats.io (presented further in Section 3.3.2), and they gathers session-related data from WebRTC-based conversation through the WebRTC API.

This aside, the WebRTC-dashboard's goal is to provide the end-user with a deeper understanding of what kind of technical- and non-technical factors can influence the QoE in a graphical and interactive way. The WebRTC-dashboard consists of six parts which are briefly presented below.

- **Conversation Handler Panel:** The Conversation Handler Panel is responsible for handling WebRTC-based conversations. It allows end-users to choose and manage the settings of a conversation, and customize how the conversation's data will be presented in the QoS Panel.
- **Quality of Experience (QoE) Panel:** The QoE Panel is responsible for presenting subjective user feedbacks, and device information. The device information contains the information about the device the participants used during the WebRTC-based conversation for a selected WebRTC-based conversation.

- **Media Player Panel:** The Media Player Panel, is responsible for play, pause, stop and drag (slider) actions. These actions are used to manage the generated charts and attached videos for a selected WebRTC-based conversation.
- **Video Panel:** The Video Panel is responsible for showing the attached videos, and editing settings in respect to these videos. These settings are mute, adjust the size of the videos, and hide.
- **Quality of Service (QoS) Panel:** The QoS Panel is responsible for the actions of adding, removing, and plotting charts. These actions allow end-users to generate charts, remove charts, and plot charts using statistics by choice.
- **Navigation bar:** The Navigation bar includes two modals. First, the Help modal is a guide of how to use the WebRTC-dashboard. Second, the Admin modal supports and allows an end-user to add and remove WebRTC-based conversation to/from the WebRTC-dashboard.

3.2 Software Requirements Specification

Software Requirements Specification (SRS) is a complete description of what the software system is expected to perform, and as well as what it is not expected to do [48]. This specification follows the Institute of Electrical and Electronics Engineers (IEEE)'s SRS standard [50].

3.2.1 Functional Requirements

Functional requirements describe the functionalities that the software system is expected to perform [48]. One of the main functional requirements of the WebRTC-dashboard is that the system (the WebRTC-dashboard) shall support and analyze an n-party WebRTC-based conversations (requirement Identification (ID) *1.1* from Appendix A.1).

As mentioned previously in Section 3.1, the WebRTC-dashboard consists of six parts. Each part is responsible for different functionalities of the WebRTC-dashboard. The functional requirements for each part are summarized in Appendix A.1. An example for a functional requirement is, the end-user shall be able to select a WebRTC-based conversation he/she wishes to analyze on the Conversation Handler Panel (requirement ID *2.1* from Appendix A.1.2).

As the WebRTC-dashboard was highly dependent on the feedback from the supervisor and responsible professor during the preparation of this master thesis, the functionalities of the WebRTC-dashboard were added and implemented in the development process. For that reason, not all the functional requirements were defined from the beginning but added to the implementation process. However, all

the functional requirements are listed in Appendix A.1 and are fully implemented in the WebRTC-dashboard.

3.2.2 Non-Functional Requirements

Non-functional requirements describe all the remaining requirements which are not included in the functional requirements [48] i.e. they include everything else than what the system is expected to perform. Two non-functional requirements are for example: the system (the WebRTC-dashboard) shall be supported by Google Chrome web browser (requirement ID *8.1* from Appendix A.2) and the text in the system should be written in English (requirement ID *8.3* from Appendix A.2).

3.2.3 External Interfaces

The WebRTC-dashboard is a web interface developed mainly in JavaScript (JS) and is accessible through the web browser. Since not all web browsers support the fifth version of HyperText Markup Language (HTML) video tag or the video format mpeg 4 (mp4), the WebRTC-dashboard is only tested on the desktop web browser Google Chrome. Also, the design of the WebRTC-dashboard is reserved for Personal Computers (PCs) and Macintoshes, consequently the WebRTC-dashboard is not tested on tablets and smartphones.

3.2.4 Performance

Almost every functionality that the WebRTC-dashboard supports is handled instantly. However, there are two exceptions, and both of them are located in the Admin modal. First, the function to add a new WebRTC-based conversation to the WebRTC-dashboard. This process requires the WebRTC-dashboard to load files from client to server (the loading time depends on the size of the files), then to run the script to retrieve the content of the files, and finally, to build and reload the WebRTC-dashboard. Second, the function to remove WebRTC-based conversations. This operation is a less time-consuming process than to add conversations. When the WebRTC-dashboard removes a conversation, it must first remove the files, build it, and finally reload the WebRTC-dashboard.

3.2.5 Attributes

One of the significant issues when considering maintainability is that getstats.io does not have a standardized JSON format as yet. Therefore, if getstats.io changes the JSON format, the WebRTC-dashboard needs to be updated in order to support the statistics gathered from getstats.io.

Since the WebRTC-dashboard does not require a user login, the security of the WebRTC-dashboard has not been prioritized during the implementation process. Even if security has not been prioritized, the task runner *Gulp* (presented in Section 4.1.3) is used to minify the JS files located on the server, so that they are not longer readable for the human eye.

3.2.6 Design

The WebRTC-dashboard is designed using HTML5 (presented in paragraph *HTML5* in Section 4.1.3) and Cascading Style Sheets (CSS) (presented in paragraph *CSS* in Section 4.1.3). The design of WebRTC-dashboard was defined in co-operation with the supervisor.

3.3 Data Retrieval

As previously mentioned in Section 3.1 the purpose of the WebRTC-dashboard is to give the end-user a deeper understanding of what kind of technical- and non-technical factors that can influence the QoE. In order to do so, the WebRTC-dashboard uses network parameters, device information, audio and vide recordings, and user feedbacks from three different platforms. These platforms are presented in the following sections.

3.3.1 Google Chrome's WebRTC Internal Interface

Initially, Google Chrome's WebRTC internal interface[1] was developed for WebRTC application developers to debug, and to understand the features and functions of their WebRTC service. It supports the function to fully visualize the session-related Chrome statistics (illustrated in Figure 3.1) and to download them to perform post-processing analysis. Recently this interface has been used to obtain a deeper insight into the QoE aspects of WebRTC services [45], and is therefore included in the WebRTC-dashboard.

To use the Google Chrome's WebRTC internal interface properly, each participant must open the interface before starting the conversation, and not close the browser window before the session has ended. Also, it is important that the WebRTC-based conversation takes place using the same web browser (Google Chrome) as the interface. Each participant can locally visualize the Chrome statistics in real-time and retrieve them from the interface whenever and how many times he/she wishes to do during the conversation. However, due to analytic reasons, it is recommended that all the participants in the conversation retrieve the Chrome statistics synchronously at the end of the conversation.

[1] Address: chrome://webrtc-interals

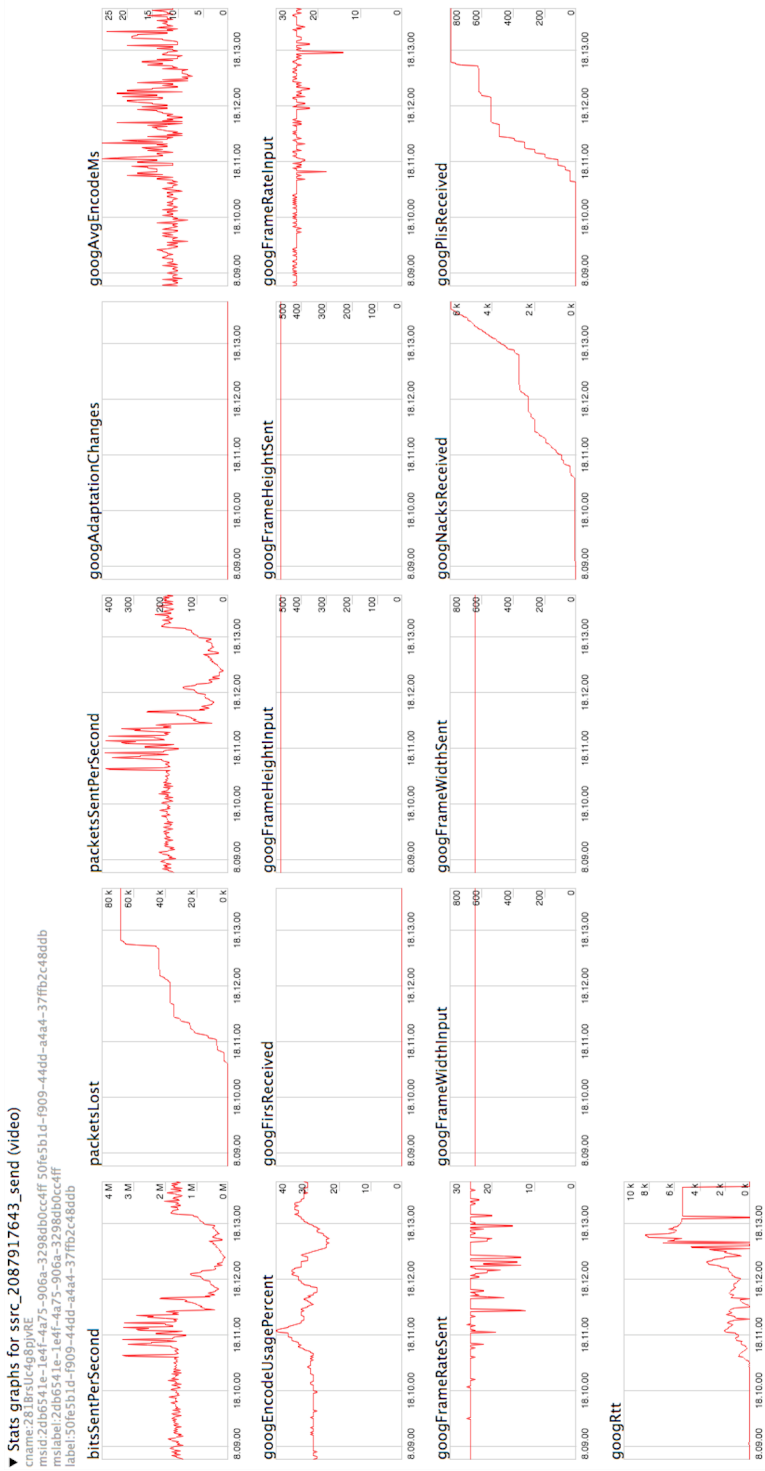


Figure 3.1: Screenshot of the Google Chrome's WebRTC internal interface. (chrome://webrtc-internal)

Each collected Chrome statistic contains all the *PeerConnection* objects defined in W3C API, plus some additional Google-specific statistics, which are organized in JSON format. In a two-party conversation, each participant's Chrome statistics file contains one *Data channel*, which contains two *MediaStreams*, one for video and one for audio. Each *MediaStream* contains two *tracks*, one for sending and one for receiving, and each of them is identified by a unique SSRC ID. The SSRC ID links the two parties in a *PeerConnection* [46]. See Figure 3.2 for an illustration.

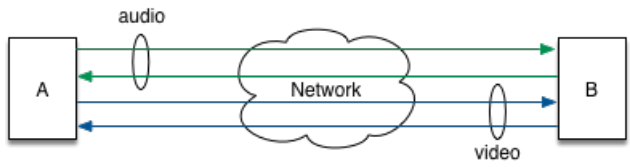


Figure 3.2: Illustration of a two-party video conversation with four tracks.

The Chrome statistics contain a significant amount of session-related statistics (an overview of all the statistics are listed in Appendix B.1). However, not all of them are applicable for the purpose of the WebRTC-dashboard and are therefore not included. The most relevant session-related statistics that are included in the WebRTC-dashboard and are found in Table 3.1.

Google Chrome's WebRTC Internal Interface's Limitations

Unfortunately, the Google Chrome's WebRTC internal interface suffers from some limitations [46]. These limitations presented in the following and cover both the design limitations of the Google Chrome's WebRTC internal interface as well as Chrome statistics limitations.

- **Manual download of statistics:** Google Chrome's WebRTC internal interface supports the function to fully visualize the Chrome statistics in real-time and to download the statistics to perform post-processing analysis [46]. However, Google Chrome's WebRTC internal interface requires the end-user to download the Chrome statistics file immediately after a session (before closing the browser window or depart from the WebRTC-based application), or the statistics will be lost.

- **Remember to download statistics:** At the end of the session, each participant in a conversation must remember to download the Chrome statistics, which is not always easy to remember. Currently, Google Chrome's WebRTC internal interface does not support to download these statistics automatically

Table 3.1: Google Chrome's WebRTC internal interface statistics included in the WebRTC-dashboard.

Media	**Source**	**Covered parameters**
Audio	send	audioInputLevel, bitsSentPerSecond, bytesSent, googEchoCancellationEchoDelayMedian, googEchoCancellationEchoDelayStdDev, googEchoCancellationQualityMin, googEchoCancellationReturnLoss, googEchoCancellationReturnLossEnhancement, googJitterReceived, googRtt, packetsLost, packetsSent, and packetsSentPerSecond.
Audio	receive	audioOutputLevel, bitsReceivedPerSecond, bytesReceived, googAccelerateRate, googCaptureStartNtpTimeMs, googCurrentDelayMs, googDecodingCNG, googDecodingCTSG, googDecodingNormal, googDecodingPLC, googDecodingPLCCNG, googExpandRate, googJitterBufferMs, googJitterReceived, googPreemptiveExpandRate, googPreferredJitterBufferMs, googSecondaryDecodedRate, googSpeechExpandRate, packetsLost, packetsReceived, and packetsReceivedPerSecond.
Video	send	bitsSentPerSecond, bytesSent, googAdaptationChanges, googAvgEncodeMs, googEncodeUsagePercent, googFirsReceived, googFrameHeightInput, googFrameHeightSent, googFrameRateInput, googFrameRateSent, googFrameWidthInput, googFrameWidthSent, googNacksReceived, googPlisReceived, googRtt, packetsLost, packetsSent, and packetsSentPerSecond,
Video	receive	bitsReceivedPerSecond, bytesReceived, googDecodeMs, googFirsSent, googFrameHeightReceived, googFrameRateDecoded, googFrameRateOutput, googFrameRateReceived, googFrameWidthReceived, googMaxDecodeMs, googMinPlayoutDelayMs, googNacksSent, googPlisSent, googRenderDelayMs, googTargetDelayMs, packetsLost, packetsReceived, and packetsReceivedPerSecond.
Bandwidth	both	googAvailableSendBandwidth, googAvailableReceiveBandwidth, googActualEncBitrate, googTargetEncBitrate, googTargetEncBitrateCorrected, googTransmitBitrate, googRetransmitBitrate, googBucketDelay, packetsSent, packetsSentPerSecond, packetsDiscardedOnSend, googRtt, bytesSent, bytesReceived, bitsSentPerSecond, and bitsReceivedPerSecond.

after a session has ended. Moreover, since, the Chrome statistics will be lost after the conversation, there is no way of reconstructing these statistics.

- **A limited number of sample points:** Google Chrome's WebRTC internal interface collects data samples each second. If the WebRTC-based conversation lasts longer than 1000 seconds, only data samples from the last 1000 seconds are recorded i.e. data samples older than 1000 seconds are lost [46].

- **Undocumented Chrome statistics extensions:** It is a challenge to analyze the downloaded statistics because the WebRTC statistics are undocumented. This causes some uncertainty in respect to their reliability when analyzing the attributes [46] of Chrome statistics.

- **Imprecise sampling time:** The Chrome statistics are collected at each of the participant's web browsers, which means that to be able to analyze the conversation, the statistics from all browsers must be recorded, downloaded and manually combined and synchronized [45]. This implies that the Chrome statistics are recorded at the same time, which means that all of the devices participating in the conversation have synchronized clocks [46].

- **Web browser dependent:** Google Chrome's WebRTC internal interface is only available through Google Chrome web browser. For example, Telefónica's *Hello*, which is only accessible through Mozilla Firefox, cannot access the Google Chrome's WebRTC internal interface.

- **Fixed sampling time:** Chrome's WebRTC internal interface uses fixed sampling time of one second. This means that Chrome's WebRTC internal interface gathers session-related data every second; this setting cannot be modified.

Even though, Google Chrome WebRTC internal interface suffers from multiple limitations, according to [45, 46] the Chrome statistics are still useful, as long as these limitations are known and handled carefully. For instance, Chrome statistics may have valuable information in respect to finding the origins of performance issues in WebRTC-based conversations. Knowing these limitations can also help to get a better understanding of how technical factors and the end-users' QoE may correlate with each other [45].

3.3.2 getstats.io

appear.in recently launched a customized WebRTC analytic interface called getstats.io[2], which provides session-related statistics. Like Google Chrome's WebRTC internal interface, getstats.io supports the function to visualize the session-related

[2] Address: https://getstats.io

Figure 3.3: Screenshot of the getstats.io.
(https://getstats.io)

statistics (illustrated in Figure 3.3) and to download them at any time (also after the conversation has ended). Compared to Google Chrome's WebRTC internal interface, getstats.io supports the function to collect all session-related statistics from all the participants and combine them into one single JSON file. Consequently, this removes the hassle of combining and synchronizes the session-related data files after a conversation.

Compared to Chrome statistics, getstats.io statistics files includes in addition to network related statistics, device-related statistics, such as browser and platform information (see the complete list of statistics supported by getstats.io in Appendix

B.2). By viewing both network- and user statistics the end-user can obtain a better understanding of how these statistics can correlate with each other. For example, the network capacity (throughput) may be different from a participant connected to a wired network, compared to a participant connected to a wireless network.

Table 3.2: getstats.io network statistics included in the WebRTC-dashboard.

Media	**Bound**	**Covered parameters**
Audio	Inbound	BitsReceivedPrSecond, BytesReceived, Jitter, PacketsLost, and PacketsReceived.
Audio	Outbound	BitsSentPrSecond, BytesSent, PacketsSent, RoundTripTime, EncodeCPUUsage, CPULimitedResolution, and BandwidthLimitedResolution.
Video	Inbound	BitsReceivedPrSecond, BytesReceived, Jitter, PacketsLost, and PacketsReceived.
Video	Outbound	BitsSentPrSecond, BytesSent, PacketsSent, and RoundTripTime.

Since both getstats.io and Google Chrome's WebRTC internal interface use the W3C API to retrieve session-related statistics form WebRTC-based conversation, they include many of the same statistics. However, they differ in how often the data samples are collected. getstats.io collects one sample per ten seconds, while Google Chrome WebRTC internal interface retrieves one sample per second. The WebRTC-dashboard includes most of the statistics covered by getstats.io. and are listed in Table 3.2 and 3.3.

Table 3.3: getstats.io user statistics are included in the WebRTC-dashboard.

Parameter
UserID
Browser
BrowserVersion
BrowserEngine
BrowserEngineVersion
OS
Platform
Mobile

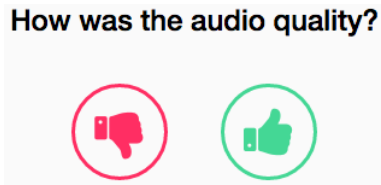


Figure 3.4: Screenshot of feedback window in appear.in.
(https://appear.in)

Furthermore, getstats.io includes subjective user feedback. At the end of an appear.in session, the user is asked to rate the conversation with either thumb up, or thumbs down (see Figure 3.4). This rating is stored in the appropriate JSON file at getstats.io. However, to receive more detailed user feedback, an extended WebRTC study of appear.in has been conducted at NTNU. Instead of a thumbs up and thumbs down option, this research added questionnaires to retrieve more information from the end-users. These questionnaires are only accessible when the NTNU's appear.in test server[3] is used. This research uses the statistics files stored at getstats.io to find the corresponding WebRTC conversation to include the questionnaires. It is important to highlight that these answers are stored at NTNU's servers, and not at Telenor[4].

Even though these questionaries' include a number of questions, only three of them are included in the WebRTC-dashboard; these questions are shown in Figure 3.5. This is because the remaining questions can change and may not always be proper to include.

	5- Excellent	4- Good	3- Fair	2- Poor	1- Bad
How would you rate the overall audiovisual quality of the session (the overall combined audio and video quality)?	○	○	○	○	○
How would you rate the video quality of the session?	○	○	○	○	○
How would you rate the audio quality of the session	○	○	○	○	○

Figure 3.5: Screenshot of user feedback form in appear.in test server.
(https://appear.item.ntnu.no).

[3]Address: https://appear.item.ntnu.no
[4]Telenor is Norwegian multinational telecommunications company, and owns appear.in.

getstats.io's Limitations

Similar to Google Chrome's WebRTC internal interface, getstats.io also suffers from some limitations. These limitations are presented below.

- **Limited number of statistics are visualized:** As illustrated in Figure 3.3, getstats.io supports the function to visualize session-related statistics. However, getstats.io can only visualize a limited number of the statistics, and does not support any end-user customization to show other statistics.
- **Inconsistent naming:** getstats.io uses different naming labels on the visualized statistics than are found in the JSON file. The same statistic can have two different labels. For example, the statistic *RoundTripTime* in the JSON file is referred to as *latency* in getstas.io.
- **Undocumented computations:** Due to the lack of a clear definition of how some of the parameters are computed, it is challenging to verify the statistics visualized in getstats.io. For example, it is a challenge to determine which network parameters (*BitsSentPerSecond, BitsReceivedPerSecond, ByteSent, ByteReceived* and *Round Trip Time*) getstats.io has used to compute *throughput*.
- **Fixed sampling time:** getstats.io uses fixed sampling time of ten seconds, this means that getstats.io collects session-related data every ten seconds and this sampling time cannot be modified.

3.3.3 Audio and Video Recording

Audio and video recording have been included in the WebRTC-dashboard to detect negative video and audio quality deteriorations (such as video freezes, bad or no audio, etc.). These quality deteriorations are easily detected in audio and video recordings. The WebRTC-dashboard supports the function to replay the audio and video together with the session-related statistics gathered from the other analytic platforms. This increases the probability of finding correlations between these quality deteriorations and session-related statistics.

However, to obtain the full potential of video and audio recordings, it is important that the videos are synchronized before uploading them into the WebRTC-dashboard, i.e. the video and audio must start at the beginning of the WebRTC conversation. It is the end-users' responsibility to trim the video such that it starts at the same time as the data samples in the statistics files.

Chapter 4
Development of the WebRTC-Dashboard

This chapter presents the development of the WebRTC-dashboard. First, this chapter will present the implementation of the WebRTC-dashboard, and then the advantages of the WebRTC-dashboard.

4.1 Implementation

This section will present the implementation process of the WebRTC-dashboard. First, this section will cover the system architecture, then testing, followed by the technologies used to implement the WebRTC-dashboard. Furthermore, the challenges that have been experienced during the development process will be covered, and finally code implementation.

4.1.1 System Architecture

The system architecture is one of the first and also one of the most important decisions that is made when developing a software system. In this case, Model View Controller (MVC), was the chosen system architecture, due to the technologies that were chosen for the WebRTC-dashboard. MVC is a software architectural pattern to implement web applications, and gives full control over HTML, CSS, and JavaScript (JS) [23]. Furthermore, MVC consists of three components; a model, a view, and a controller, which are described in the following and illustrated in Figure 4.1.

- **Model:** A model is a place where the application's data objects are stored. The model has no knowledge of the operations of either the controller or the view [23]. When an application's data needs to be changed, the controller sends a notification to the model to update the data. Alternatively, if the model changes, it sends a notification to the controller. For example, when an end-user manually removes a WebRTC-based conversation from the server, the model must inform the controller.

- **View:** A view is what the end-user is presented and how the user interacts with the web application. The view handles user actions and inputs [23]. For example, when the end-user initiates the action of removing a WebRTC-based conversation from the WebRTC-dashboard, then the view notifies the controller, and the controller tells the view what to do.

- **Controller:** A controller connects the model and the view together. The controller contains the logic behind the web application, and it can control both model and the view [23]. For example, if the end-user triggers to remove a WebRTC-based conversation, the view informs the controller of the action, which will result in the controller directing the model to remove the conversation.

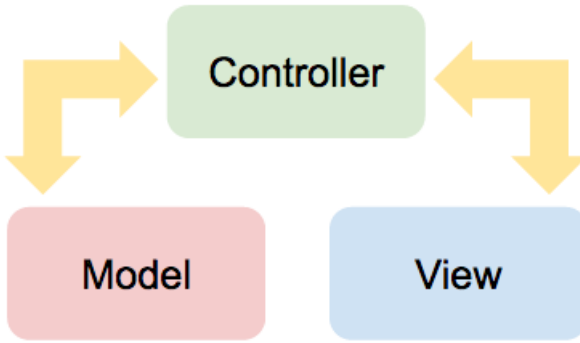


Figure 4.1: Illustration of MVC architecture.

Since the purpose of the WebRTC-dashboard is to perform analysis of statistics that have already been generated, there is no need for the WebRTC-dashboard to modify these statistics after they are uploaded into the WebRTC-dashboard. Therefore, the WebRTC-dashboard does not support a proper database (further explanation is found in Paragraph *No Database* in Section 4.1.3), but uses text (txt), mp4, and JSON files to retrieve data. Figure 4.2 illustrates the system architecture of the WebRTC-dashboard. Even though the WebRTC-dashboard does not support a proper database, it is important to highlight that the WebRTC-dashboard can retrieve data from the model and that the controller will notify the model each time the end-users wish to remove or add a WebRTC-based conversation(s).

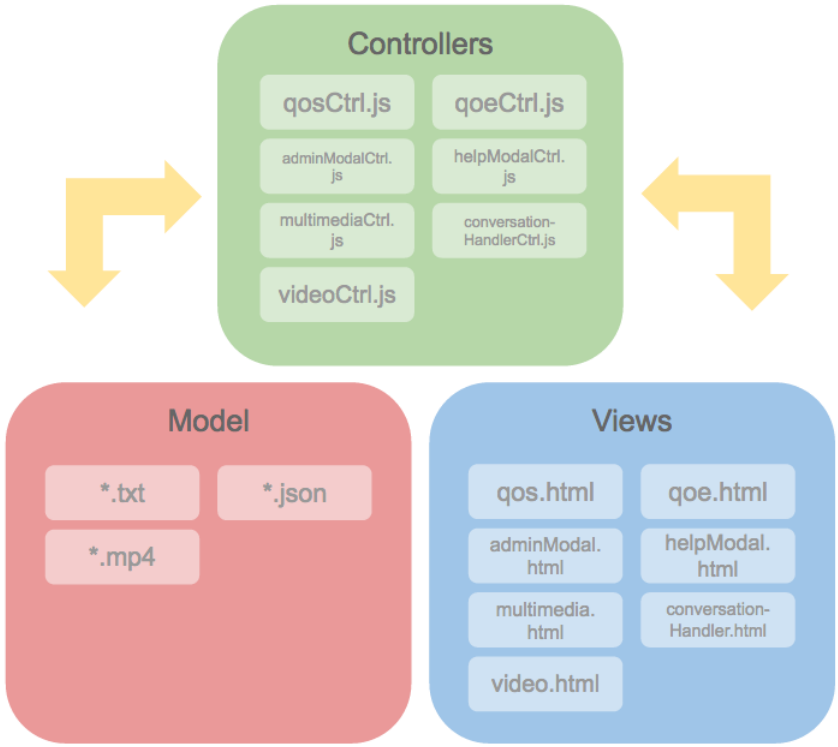


Figure 4.2: Illustration of the system architecture of the WebRTC-dashboard.

4.1.2 Testing

During the implementation process, testing was conducted. In order to test the functional requirements (presented in Section 3.2.1), console logging and simple unit tests were implemented along with the implementation process. Non-functional requirements were also tested, for example, testing the WebRTC-dashboard in the required web browser (Google Chrome). However, due to limited time, only a limited number of the functional and non-functional requirements were tested.

4.1.3 Technologies

One of the many challenges when developing a software project is to identify the technologies that that best suit the purpose of the project. This section will present the technologies used to develop the WebRTC-dashboard and why they were chosen.

JavaScript

JavaScript (JS) is a lightweight, object-oriented language, and is best known as the language used to build web pages. When developing a software project, it is important to consider the response time of the elements included in the project. One of the greatest advantages of using JS is that JS runs client-side. Running client-side means that all the computations and logic are done at the end-users web browser [20]. As a consequence, JS is perfect for e.g. validation forms, showing/hiding elements dynamically, animation etc. However, JS is not recommended for computing security-sensitive data, for example, handling passwords and applying Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) verifications.

PHP: Hypertext Preprocessor (PHP) is another programming language used to create web pages. Compared to JS, PHP is strictly server-sided. Hence, when accessing a .PHP file, a client must send a request to the server, after receiving the request the server must send its content back to the client in order for the end-users to view the content [27]. Consequently, this can cause delays and longer response time.

Since the WebRTC-dashboard's elements (video, charts, etc.) require a fast response time and does not require a login or CAPTCHA verifications, the WebRTC-dashboard use of JS. JS is perfect for the purpose of developing the WebRTC-dashboard. Compared to PHP, JS allows the WebRTC-dashboard to be a highly responsive web interface with dynamic functionalities, without having to wait for the server to react and reply back to the client.

HTML5

JS, CSS, and HyperText Markup Language (HTML) are all cornerstone technologies when creating web pages, mobile- and web applications [53]. HTML is the standard markup language which is used to structure and markup content on the WWW. In 2014, W3C published a fifth generation of HTML, HTML5. HTML5 is an improved markup language with support for the latest multimedia and other optional attributes [34], and is the current standard of HTML. Since the WebRTC-dashboard requires audio and video recordings, HTML5 is utilized.

CSS

Cascading Style Sheets (CSS) is one of the core languages of the WWW, and has been standardized by W3C specification. Compared to JS, CSS is not a programming language, but a style sheet language used to define the presentation of a document written in HTML or another markup languages [11].

There exists several other style sheet languages (extensions of CSS), like *Syntactically Awesome Stylesheets (Sass)* [12] and *less*[1]. *Sass* was considered when deciding the style sheet language to be used for the WebRTC-dashboard. As the name implies, *Sass* is an upgraded version of CSS and allows more advanced features such as variables and nested rules inside the CSS files [12]. After evaluating both *Sass* and CSS, it was found that there was no need for the advanced features *Sass* offered for the WebRTC-dashboard. *Sass* is completely compatible with all versions of CSS [12] and, if necessary, *Sass* can be included in future implementations.

Git

During development, it is important to be able to store and save the code somewhere safe if something unfortunate should happen. There are several places where code can be stored and saved, for example, on an external hard drive or in a drop box[2]. However, when it comes to software projects, Git is the most suggested option. Git is a version control for software development and supports distributed, non-linear workflows [1]. With Git, a user can save (commit), store (push) the code onto the Git server, and later retrieve (pull) the code and continue the development.

Team Foundation Server (TFS) is another version control used for software projects which were also considered. TFS supports many of the same functionalities as Git. However, from experience TFS does not allow to commit (save) code without pushing (storing) it. This may cause Git merge conflicts when several developers are working on the same file. However, since there was only one developer implementing the WebRTC-dashboard, it did not matter which version control was chosen. In this case, personal favoritism did decide, and Git was chosen for the WebRTC-dashboard.

Bash

Bourne-Again Shell (Bash) is the command language interpreter for the GNU's not Unix (GNU) OS and is the default shell on Linux and OS X [7]. Its main purpose is to allow interaction with the computer's OS [42]. Generally, Bash runs commands from a text window (shell), but also supports reading commands from files, which are called scripts. Some of the commands that Bash supports are filename globbing (wildcard matching), piping, variables and iteration through files [7].

Since getstats.io does not have a standardized JSON format yet, the JSON structure can change. Therefore the WebRTC-dashboard uses Bash scripts to extract data from the JSON file, and then store the data into txt files, these files are later used by the WebRTC-dashboard to retrieve data. Consequently, from a maintenance

[1] http://lesscss.org/
[2] https://www.dropbox.com/

perspective, if the structure of the JSON file should change, the only file that needs to be modified is the Bash script extracting data from the getstats.io statistics files.

No Database

As already mentioned in Section 4.1.1, the WebRTC-dashboard does not support a proper database. Even though, a database for a web application is highly recommended and important, the WebRTC-dashboard utilizes JSON, txt, and mp4 files to retrieve data. This is because there has already been done some work (Bash scripts) in order to retrieve data, and this is the main reason the WebRTC-dashboard uses files as a "database" and not a proper database (such as *MyStructured Query Language (SQL)*[3]).

Other Essential Technologies

Angular.js JS is excellent at defining functionalities and HTML is an excellent template language for static documents, but how to fill the gap between them? If information should change over time, how should this be updated? *Angular.js* solves the mismatch between dynamic applications and static documents by including a powerful library with a collection of functions that support communication between HTML and JS [3].

Angular.js is an open sourced structural framework for dynamic web applications [3] and supports data binding between HTML and JS. Briefly explained, *Angular.js* works by first by locating the *Angular.js*-specified attributes (double curly braces) in the HTML document and binds the input or output that is represented by JS variables.

Alternatives to *Angular.js* are *jQuery*[4] and *React*[5]. Both *jQuery* and *React* could been used for the WebRTC-dashboard, but because of personal experience, *Angular.js* was the most confortable choice. In retro-spect after seeing these technologies, *React* may be a better choice considering the framework size and performance. *React* requires less lines of code to accomplish the same function as *Angular.js*. For this master thesis, there was not enough time to research and learn a new data-binding tool from scratch, therefore, *Angular.js* was selected.

Node.js JS runs client side, but what about the server side? *Node.js* is an open source command line tool for developing server-side web applications [24]. It is designed to easily create fast and scalable network applications, as it is capable of handling thousands of simultaneous connections with a high throughput [25].

[3] https://www.mysql.com/
[4] https://jquery.com/
[5] https://facebook.github.io/react/

In traditional web platforms, Hypertext Transfer Protocol (HTTP) requests and responses are treated as separate streams, while *Node.js* enables the requests to be sent in parallel, which empower the application to process files faster [41]. To sum up, the WebRTC-dashboard uses *Node.js* to obtain better performance.

NPM There is no point of working on a problem that someone already has solved. Node Package Manager (NPM) makes it easy for JS developers to share their code that solves a particular problem and enables other developers to reuse the code in their own projects [26]. Both *Gulp* (presented in Paragraph *Gulp* in Section 4.1.3) and Paragraph *Chart.js* in *Chart.js* (presented in Section 4.1.3) are NPM projects that other developers have made and shared with others. In addition, for future development, NPM is an easy way to update NPM projects (also known as NPM packages) in the event some of them release a new version.

Gulp *Gulp* is a task runner [18]. Although it may go without saying, a task runner runs tasks. A task runner applies self-defined tasks on files in a software project. Usually software projects consist of many files, such as images, videos, text files, etc. Sometimes it is not necessary for all these files to be located on the server, or be readable for the human eye (due to security issues). In these cases, the task runner is put to good use! There are several tasks a task runner can perform, but the most basic tasks are moving, copying, cleaning, and minifying. Minifying a file means that it compresses the file into fewer kilobytes, such that the compressed version may no longer be readable for humans in its minified state, but the content remains the same. Web browsers will not have a problem reading the minified file and with fewer kilobytes the uploading time will be shorter.

Another task runner, which is better known than *Gulp*, is *Grunt*. *Grunt* is a *Node.js*-based task runner [16], while *Gulp* only uses *node.js* [19]. There are no significant differences between *Gulp* and *Grunt*, they both are able to perform the same tasks. Yet, a small difference separates these two, which is that *Gulp* requires less code lines when performing the same task. In addition, *Gulp* is known for having less configuration than *Grunt* [17], and as this was the first time I had experience with task runners, *Gulp* was selected for the WebRTC-dashboard.

Chart.js When developing a WebRTC-dashboard that will present a considerable amount of data, it is important that the data be presented in a good and readable format. Charts are far better for displaying data visually than for example tables. On the other hand, charts may be difficult to create and implement. One solution is to include open source projects that create charts. There are many such libraries,

but in this case only three of them were considered: *Data-Driven Documents (D3)*[6], *Highchart*[7], and *Chart.js*[8].

All of them support good-looking charts, however they support different levels of functionalities. When considering customization, *D3* stands out. *D3* supports complexity and flexibility of different types of charts (line-, bar-, pie chart etc.) but this also power the number of settings and configurations [13]. *Highcharts* and *chart.js* are considered much alike, the only difference between them is that *Highcharts* supports a higher chart veracity [21], while *Chart.js's* a more light weighted and responsive [9]. Considering the purpose of the WebRTC-dashboard and what it requires (light weighted charts and only line-charts), *Chart.js* seems to be the best alternative.

In addition, a new version of *Chart.js*, version 2 was published 9.april.2016 [10]. Even though the new version included several features that would be nice-to-have in the WebRTC-dashboard, like double Y-axis, chart titles, etc., there was no time to implement these features in this master thesis. Unfortunately, these features must be included in future work.

Bootstrap *Bootstrap* is the most popular HTML, CSS, JS framework that makes front-end web development faster and easier [8]. It contains both HTML- and CSS based design templates for such as: buttons, tables, navigation, etc., and also optional JS extension for future functionalities. Instead of defining CSS for buttons, dropdowns, panels, etc. yourself, developers can easily import the *Bootstrap* library for free and save time. For this reason, the WebRTC-dashboard uses *Bootstrap* framework's elements, such as radio buttons, checkboxes, dropdowns, input fields, etc.

Angular Material One element which is desperately needed for the WebRTC-dashboard and missing from *Bootstrap* is the slider found in the Media Player Panel. Luckily, *Angular Material* is like *Bootstrap*, a User Interface (UI) component framework which includes a well tested slider. Complementary to this *Angular Material* is specified for developers using *Angular.js* [2].

Font Awesome To enhance the usability of a web application, it is wise to add icons. Print, save, "hamburger"-menu, are just a few of many actions associated with an icon. However, the WebRTC-dashboard does not include actions as print and save, but, play, stop, and pause are covered. *Font Awesome* is a font and icon toolkit

[6] https://d3js.org/
[7] http://www.highcharts.com/
[8] http://www.chartjs.org/

that enables developers to apply "awesome" fonts and icons into their project [14]. To improve the usability of the WebRTC-dashboard, *Font Awesome* has been included.

Trello During development, it is critical to organize work. The WebRTC-dashboard includes several functionalities, and while developing the WebRTC-dashboard it is important always to know what has been done, what needs testing and, of course, what is still missing. To address these issues, *Trello* has been used. *Trello* is a web-based project management interface [33] with a design that reminds one of a scrum board.

4.1.4 Challenges and Decision Making During Implementation Process

At the beginning of the development process, it is almost impossible to identify all the challenges that will present themselves during the implementation process. Therefore, in this section, the master thesis will present the most significant challenges that appeared during the implementation of the WebRTC-dashboard and what decisions that were made to handle these challenges.

Chrome Statistics are Collected Correctly

The WebRTC-dashboard only supports Chrome statistics files that contain the correct JSON format, that is, the Chrome statistics file must contain the correct number of *Data channels*. The number of *Data channels* is dependent upon the number of participants participating in the WebRTC-based conversation. For a n-party WebRTC-based conversation, each participant's Chrome statistics file should include $n-1$ number of *Data channels*. For example, assuming a three-party conversation, the Chrome statistics file for each participant should include two *Data channels*. Assuming Chrome statistics file for participant A, this file should include two *Data channels*, one for connecting participant A and participant B, and a second *Data channel* connecting participant A and participant C. See Figure 4.3 for an illustration of a three-party conversation.

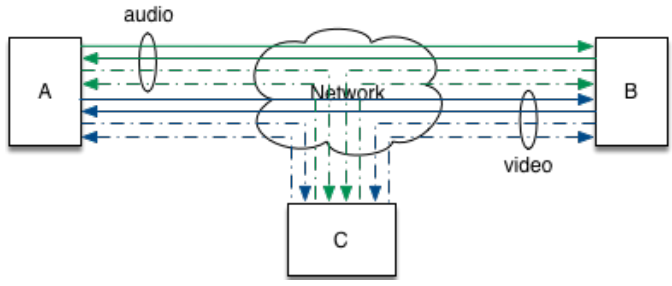


Figure 4.3: Illustration of a three-party video conversation.

However, when a participant changes the video quality or refreshes the browser window during an appear.in conversation, appear.in is developed to close all the currently connected *Data channels* to that particular participant and then creates new ones. It is important to highlight that this is only the case when utilizing appear.in, and not, for example, Google Hangouts. For example, assuming a three-party appear.in video conversation between participant A, B, and C. Each participant is connected to two *Data channels* (one for each participant, excluding themselves). If participant A adjusts his video quality, all the *Data channels* connected to participant A are closed, and new *Data channels* are opened. This means that the Chrome statistics file for participant A contains four *Data channels* (two closed, two created). However, for the other two participants' (B and C) Chrome statistics files, contain three *Data channels*. This is because the previous *Data channel* connected to participant A is closed and a new one is created, while the *Data channel* connecting participant B and C remains intact.

In order to compute the number of participants participating in the WebRTC-based conversation, the WebRTC-dashboard uses the number of *Data channels*. However, due the fact that the number of *Data channels* can vary dependent on the participants behavior, the WebRTC-dashboard uses the number of *Data channels* only when there are no other options available. That is if Chrome statistics is the only available data source for the WebRTC-based conversation that is selected to analyze.

The Number of Chrome Statistics Files are Collected Correctly

As presented previously in Section 3.3.1, each participant must remember to download the Chrome statistics file. Each Chrome statistics file contains $n-1$ *Data channels* for a n-party conversation. Each *Data channel* contains two *MediaStreams*, one for video and one for audio. Each *MediaStream* consists of two *tracks*, one for sending and one for receiving, and each *track* is identified by a unique SSRC ID. SSRC ID links the two parties in a *PeerConnection* [46]. Figure 4.4 illustrates a two-party WebRTC-based video conversation.

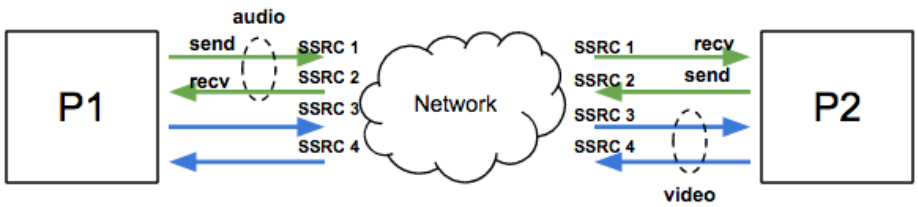


Figure 4.4: Illustration of a two-party video conversation with two Chrome statistics files.

When n is greater than *two* in a n-party video conversation, a potential challenge arises. In order to identify the *PeerConnections* between the participating parties, the WebRTC-dashboard must know which *track* is connected to who. To do so, the WebRTC-dashboard requires that all of the Chrome statistics files for a WebRTC-based video conversation are collected. Figure 4.5 illustrates how the SSRC IDs connects three participants together in a three-party video conversation. For example, if Chrome statistics files from both participant 2 (P2) and participant 3 (P3) is missing, then it is impossible for the WebRTC-dashboard to idntify which participant is receiving, for example, audio data on *track* ID SSRC *1*.

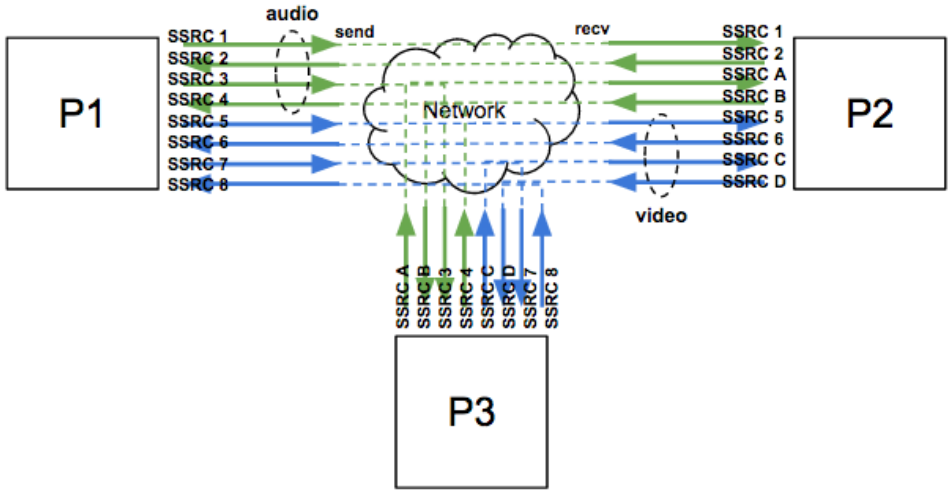


Figure 4.5: Illustration of a three-party video conversation with three Chrome statistics files.

One of Google Chrome's WebRTC internal interface limitations is that it does not support downloading Chrome statistics automatically after a session (presented in Section 3.3.1). Consequently, it can be easy for the end-user to forget to download the Chrome statistics at the end of the session. Therefore, in order to meet this challenge, the WebRTC-dashboard has implemented a mechanism to handle WebRTC-based conversations that have been collected $n - 1$ Chrome statistics files. As long as only one Chrome statistics file is missing, the WebRTC-dashboard can assume who the last participant is. Figure 4.6 illustrates how the WebRTC-dashboard assumes who is the last participant.

Unfortunately, if more than one Chrome statistics file is missing from a n-party conversation, where n is greater than *two*, it is impossible for the WebRTC-dashboard to assume the *PeerConnections*. Therefore, the WebRTC-dashboard does not support

using the Chrome statistics files for post-analysis when more than one Chrome statistics file is missing from the conversation.

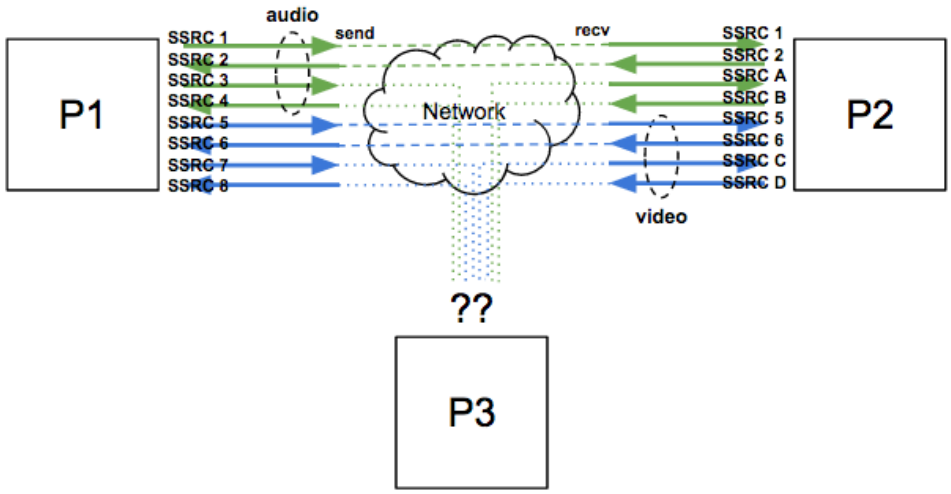


Figure 4.6: Illustration of a three-party video conversation with two Chrome statistics files.

Getstats JSON Format May Change

In the context of maintenance and future use of the WebRTC-dashboard, one of the most significant challenges is that the getstats.io does not support a standardized JSON format. This implies, that the getstats.io can change the JSON format at any time (the last time getstats.io changed the JSON format was in January 2016 [6]), which can cause the WebRTC-dashboard to not further support getstats.io statistics files. It is, therefore, important that the WebRTC-dashboard is not directly dependent on the JSON format for getstats.io statistics files.

To provide easy maintenance, the WebRTC-dashboard uses Bash scripts (presented in paragraph *Bash* in Section 4.1.3) to extract data from the getstats.io statistics files, and stores the data on to .txt files. Instead of directly retrieving statistics from the getstats.io statistics file, the WebRTC-dashboard employs the .txt files to retrieve data. If getstats.io should update and launch a new JSON format, the only file that must be modified is the Bash script for extracting the data from the getstats.io statistics files. Consequently, this gives flexibility, and easily maintenance in case of changes to the JSON format should occur.

Sampling Time of getstats.io Statistics and Chrome Statistics

In order to plot the statistics retrieved from both getstats.io and Google Chrome's WebRTC internal interface, the WebRTC-dashboard needs to have the correct time stamp of each data sample. However, Chrome statistics does not include the time stamps but instead includes the start time and end time of the WebRTC-based conversation, and the data samples. Each Chrome statistics file includes several parameters (such as *PacketsLost*, *bytesSent*, *bytesReceived*, etc.) and each of them contains data samples. From this information, it is possible for the WebRTC-dashboard to calculate the duration of the WebRTC-based conversation (in seconds) and the number of data samples for each parameter. Assumptions based on knowledge gained from research findings in [45, 46] and from real-time visualization of data on Google Chrome's WebRTC internal interface, one can assume that the sampling time for Chrome statistics is equal to one sample per second. Therefore, this master thesis assumes that Chrome statistics samples data at one second intervals. On the other hand, getstats.io statistics files collect a time stamp for each data sample. However, getstats.io retrieves data and time stamps at ten second intervals, not one second intervals as in the case for Chrome WebRTC internal interface. See Figure 4.7 for an illustration.

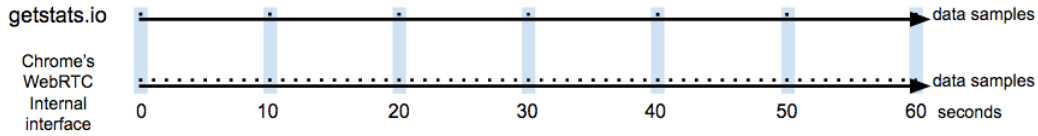


Figure 4.7: Illustration of how often getstats.io and Chrome's WebRTC internal interface retrieves data samples.

The charts included in the QoS Panel in the WebRTC-dashboard are developed with help from *Chart.js* (described in Paragraph *Chart.js* in Section 4.1.3), and since *Chart.js* requires that every plot included in the same chart should have the same sampling time, an issue arises. For example, the statistics from getstats.io and Google Chrome's WebRTC internal interface do not have the same sampling time. getstats.io statistics retrieves one data sample every ten seconds, and Chrome statistics collects one data samples each second.

To satisfy the requirement of *Chart.js* and offer the possibility of comparing statistics from both getstats.io and Chrome's WebRTC internal interface in the same chart, the WebRTC-dashboard supports a function to compute and convert the sampling times to be the same. In order to achieve this, the WebRTC-dashboard converts the sampling time of getstats.io statistics from one sample per ten seconds,

to one sample per second. This conversion is achieved by dividing each getstats.io statistics data sample by 10. Even though this computation only gives an average, this is one of the better solutions which satisfies both the requirements of *Chart.js* and gives the end-user the possibility of editing the sampling time.

By comparison, another way of supporting these requirements is to convert the Chrome statistics sampling time from one sample per second to one sample per ten seconds. This solution would have been much easier and more correct. However, if this were the case, then the WebRTC-dashboard would no longer allow the end-user the possibility of editing the sampling time to a number less than ten, or a number that is not divisible by ten.

In addition, there exists a flexibility feature in the Conversation Handler Panel in the WebRTC-dashboard, called *sample interval size*. The *sample interval size* is a term used for computing the average of a x number of data samples. For example, assuming a Chrome statistics file with 1000 data samples and a *sample interval size* $x = 10$. Then the WebRTC-dashboard will first compute the average of sample number 1 - 10, then 11 - 20, and so on. On the other hand, if the WebRTC-dashboard should use the sampling time one sample per ten seconds, the *sample interval size* would always compute the average of sample number 1 - 100, 101 - 200, and so on, when $x = 10$ (this is because each data sample is a summation of ten data samples, and the *sample interval size* computes the average of $x = 10$ data samples, which results in $10 * 10 = 100$). Therefore, in order to support as much flexibility as possible, the WebRTC-dashboard converts the getstats.io statistics sample time from one sample per ten seconds to one sample per second. Accordingly, the WebRTC-dashboard supports the end-user to select whichever *sample interval size*, as long as it is greater than zero.

Getstats.io statistics' sample data is computed to an approximate value when it is calculated from one sample per 10 seconds to one sample per second. In order to obtain the most accurate plots, the end-user is recommended to select a sampling time (*sample interval size*) that is divisible by 10. Since the getstats.io retrieves one data samples every 10 seconds; this value is the most correct to use when performing an analysis. If the end-user selects a *sample interval size* that is not divisible by 10, it is important to note that this is an approximate calculation.

Uploading WebRTC-Based Conversation Statistics Files

In order to analyze WebRTC-based conversations, the WebRTC-dashboard must support a method of uploading new statistics files. At the beginning of the implementation process, the developer had to upload manually statistics files on to the server, which was an inconvenient and demanding way of uploading new data. During the development process, Git, a version control software (presented in Paragraph *Git* in

Section 4.1.3) was used to upload updated code on to the server, which was found as an efficient procedure. However, when video files were included, these files were too big for the free version of Git. Therefore, the developer had to transfer each video file manually to the server with the *scp*[9] command. Since this process was both time consuming and an inconvenient process, finding a solution was highly prioritized. Accordingly, it was necessary to implement the Admin Modal (presented in Section 5.1.6), this modal enables the WebRTC-dashboard to add new conversations to analyze and remove old conversations.

Synchronization of Video, Charts and Slider

The slider found in Media Player Panel on the WebRTC-dashboard, contains multiple steps. The number of data samples determines the number of steps. If the Chrome statistics file contains 100 data samples, then the slider would contain 100 steps. Since Google Chrome's WebRTC internal interface retrieves one data sample per second, one can assume that a Chrome statistics file that contains 100 data samples is 100 seconds long. Each step represents a time stamp during a conversation. This means the first step represents the first time stamp in the conversation, and likewise, the last step represents the last time stamp.

However, the *sample interval size* can change the number of data samples that need to be plotted on the charts. For example, assuming a Chrome statistics file which is comprised of 100 data samples and a *sample interval size* $x = 10$, then since $100/10 = 10$, the chart will consist of 10 data samples and not 100. Then the first step would represent the time after 10 seconds, and the next step would represent the time after 20 seconds, etc. This means the draggable element must wait ten seconds before "jumping" from one step to the next one. See Figure 4.8 below which illustrates this.

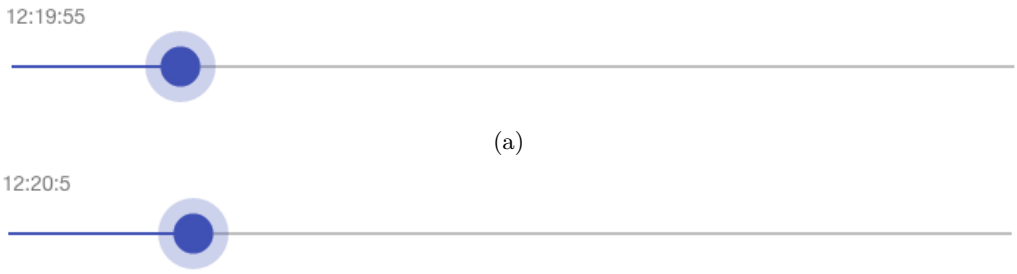


Figure 4.8: Illustration of the slider in Media Player Panel [39].

[9]scp is the command for copying files from one computer to another computer.

Each time the play button on the Media Player Panel is triggered, it starts a count down from 10 seconds (this is the same value as the *sample interval size*). Every time the countdown finishes, the draggable element on the slider "jumps" from one step to the next step, and starts the countdown all over again. However, the issues arise when the end-user triggers the pause button in the middle of a countdown. For example, if the end-user triggers the pause button when the countdown has counted down to 5 seconds, the countdown stops and gets nullified. When the end-user triggers the play button again, the countdown has to start all over again from 10 seconds.

Even though this may not sound like a problem, the real issue arises when the slider must be synchronized with the video, audio recordings, and chart(s). The charts plot a data sample every time the draggable element on the slider "jumps" and since the video and audio recordings are supported by HTML5 media functionalities, the video and audio recordings will play, pause and stop each time the respective buttons are triggered on the WebRTC-dashboard. In the case of pausing the slider, video, and charts in the middle of a countdown, the video is not affected by the seconds nullified in that action. For example, if the end-user triggers the pause button when the countdown has counted down to 5 seconds, the video, slider and charts pauses, and the countdown gets nullified. When the end-user triggers the play button to continue, the video continues to play where it paused, while the chart and slider start the countdown from 10 again. Consequently, the chart, slider, and videos will be unsynchronized by (in this example) 5 seconds.

Fortunately, there are several ways of resolving this problem. One of the solutions is that the developer can implement to store the number of seconds counted down for each step in a variable, and then continue the countdown from there. A second solution is to push the video back to the last step. Unfortunately, this problem was discovered at a late stage during the implementation process, and therefore not rectified in for the WebRTC-dashboard, but will be included in future work. There exists, however, another method of fixing the synchronization in the currently developed WebRTC-dashboard. To achieve synchronization, the end-user must drag the draggable element along the slider; it does not matter which step the element is dragged to, as long it is dragged at least one step. This action will push the video back/forward to the current step of the draggable element and synchronize the charts, video, and the slider.

4.1.5 Code Implementation

Although, it may appear relatively easy to implement a dashboard it takes a considerable amount of time. The development of this WebRTC dashboard took an average of approximately 40 hours a week during a 3 month period and consists of about

4000 lines of JS code, Bash scripts, HTML, and CSS.

4.2 Advantages of the WebRTC-Dashboard

This section will present and highlight the additional features that the WebRTC-dashboard supports. First, this section will present the replaying feature, followed by the combination and customization of Chrome and getstats.io statistics features, and finally the flexibility features.

4.2.1 Replaying Charts and Videos

One of the greatest advantages of the WebRTC-dashboard is that the WebRTC-dashboard supports the replay function for video, audio and statistics recordings. This feature is not supported by either getstats.io or Google Chrome's WebRTC internal interface. However, getstats.io supports the function to view recording statistics both during and after a WebRTC-based session has ended, but it does not support the function to replay the statistics. On the other hand, Google Chrome's WebRTC internal interface supports the function to view the statistics during the session but does not support to view the statistics in the interface after a session has ended.

The WebRTC-dashboard allows the end-users to replay video, audio and WebRTC-based conversation statistics synchronized with each other. This feature enables the end-users to perform post-processing analysis, and can easily identify a correlation between network statistics and video quality deteriorations (such as video freezes, bad or no audio, etc.). Figure 4.9 shows a video recording and a chart plotting of *PacketsLostRatio* (Equation 4.1). As illustrated, the video recordings from participant B to participant A show some negative quality deteriorations, which may be caused by the increasing packet lost which is visible in the chart below the videos.

4.2.2 Combine Chrome and getstats.io Statistics

Another great advantage of the WebRTC-dashboard is that it can plot statistics from both Chrome's WebRTC internal interface and getstats.io into the same chart. This function is not supported by either Google Chrome's WebRTC internal interface or getstats.io and gives the WebRTC-dashboard a unique advantage. This functionality gives end-users the possibility to compare different session-related statistics from different platforms. Since both Chrome's WebRTC internal interface and getstats.io use WebRTC API to retrieve data from WebRTC-based conversations, they both should support the session-related parameters to have the same values. However, because they support different sampling times and the fact that the WebRTC-

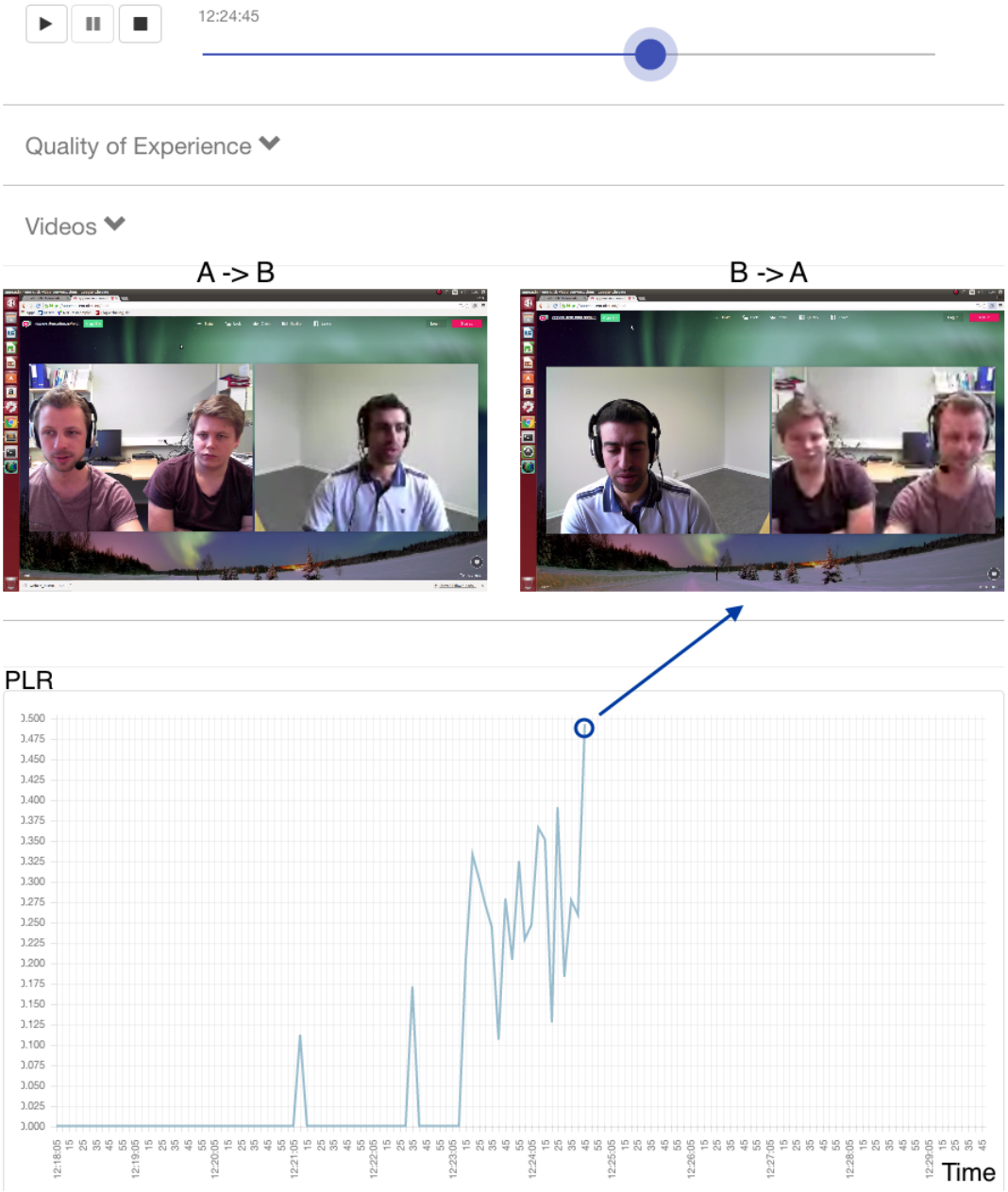


Figure 4.9: Screenshot of the WebRTC-dashboard while replaying video and chart plotting *PacketsLostRatio* [39].

dashboard computes approximate values for getstats.io, the plots of these session-related statistics may not be completely the same (see example in Figure 5.11).

4.2.3 Customized Chrome and getstats.io Statistics

In addition to the session-related statistics included in both Chrome and getstats.io statistics (presented in Table 3.1 and 3.2), the WebRTC-dashboard includes further customized statistics. These customized statistics are included to give the end-user a deeper understanding of the QoE in the context of WebRTC-based video communication application and services.

Customized Chrome Statistics

Theses next paragraphs will present the customized Chrome statistics which are included in the WebRTC-dashboard. These statistics are included in order to give the end-user additional options of how to plot the network statistics. These plots may be easier to understand and regain information from, than the ones that are already supported by the Google Chrome's WebRTC internal interface.

- **PacketsLostRatio for sender:** PacketLostRatio for sender is the number of packets lost divided by a number of packets sent for each second. i is the number of samples, and for $i = 0$, both $packetsLost_{i-1}$ and $packetsSent_{i-1}$ are equal 0. PacketsLostRatio for sender gives a better indication of how many packets are lost at the sender at each second. Currently, the Google Chrome's WebRTC internal interface supports the function to plot the total number of packets lost after i seconds, but not for each second.

$$\frac{packetsLost_i - packetsLost_{i-1}}{packetsSent_i - packetsSent_{i-1}} \tag{4.1}$$

- **PacketsLostRatio for receiver:** PacketsLostRatio for receiver is similar to PacketLostRatio for the sender. The difference is that instead of dividing by the number of $packetsSent$, PacketLostRatio for the receiver is dividing by the number of $PacketReceived$. Likewise as above, i is the number of samples, and for $i = 0$, both $packetsLost_{i-1}$ and $packetsReceived_{i-1}$ are equal 0. As for PacketsLostRatio for sender, the PacketLostRatio for receiver gives a better way of seeing how many packets are lost at the receiver at each second.

$$\frac{packetsLost_i - packetsLost_{i-1}}{packetsReceived_i - packetsReceived_{i-1}} \tag{4.2}$$

- **GoogPLIsReceivedNonCumulativ:** GoogPLIsReceivedNonCumulativ is a non cumulative version of googPLIsReceived found in the parameters for receiving

data for video. Equally as for previous statistics, i is the number of samples, and for $i = 0$, $googPlisReceived_{i-1}$ is equal 0. GoogPLIsReceivedNonCumulativ is included in order to give the end-user a better view of how many Picture Loss Indications (PLIs) are received at each second. At the present Google Chrome's WebRTC internal interface, it only supports to plot the total number of PLIs received after i seconds.

$$googPlisReceived_i - googPlisReceived_{i-1} \tag{4.3}$$

- **GoogPLIsSentNonCumulativ:** GoogPLIsSentNonCumulativ is a non cumulative version of googPLIsSent found in the parameters for sending data for video. Again, i is the number of samples, and for $i = 0$, $googPlisSent_{i-1}$ is equal 0. As similar to GoogPLIsReceivedNonCumulativ, GoogPLIsSentNonCumulativ is added as one of the customized Chrome statistics because it gives the end-users a better understanding of how many PLIs are sent at each second.

$$googPlisSent_i - googPlisSent_{i-1} \tag{4.4}$$

Customized getstats.io Statistics

The WebRTC-dashboard only includes one additional customized getstats.io statistic. This statistic is computed in the same way as the *PacketLostRatio for receiver* (Equation 4.2).

- **PacketsLostRatio:** PacketsLostRatio divides the number of packets lost by a number of packets received for each second. As before in previous examples, i is the number of samples, and for $i = 0$, both $PacketsLost_{i-1}$ and $PacketsReceived_{i-1}$ are equal 0. This ratio is included in order to get a deeper understanding of how many packets are lost at each second.

$$\frac{PacketsLost_i - PacketsLost_{i-1}}{PacketsReceived_i - PacketsReceived_{i-1}} \tag{4.5}$$

4.2.4 Flexibility Features

Flexibility features are features the WebRTC-dashboard uses in order to customize analysis. With these features, the end-user can customize an analysis to adapt to its purpose. These features are found in the Conversation Handler Panel's additional settings and are described below.

- **Selecting the number of charts to be shown on each row:** For various reasons, an end-user may want to vary the number of charts that are shown on

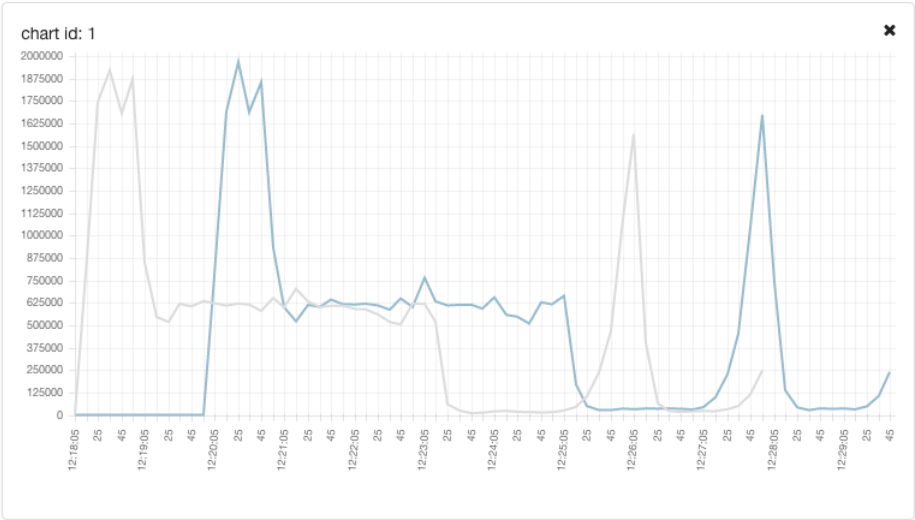
their screen. Currently, the WebRTC-dashboard supports between *one* to *four* charts for each row. If the number is greater than 4 or less than 1, the WebRTC-dashboard will not support it, and the default value (which is *one*) or the last selected value overwrite the unaccepted value.

- **Shift or no shift:** As highlighted in one of the Chrome statistics limitations in 3.3.1 and also described in [46], when statistics are collected from multiple parties, the internal clock at each of the participant's machine may not be synchronized. In this case, the data samples of each participant may not start from the same time, and be shifted when they are plotted on the charts. See Figure 4.10 for an example of this. To solve this problem, the WebRTC-dashboard supports a *no shift* option. When this option is selected, the WebRTC-dashboard does not consider the time when each data sample is collected, but, however, assumes that they all start at the same time. The WebRTC-dashboard finds the first time a data sample is collected and uses this time as the start time for the conversation.

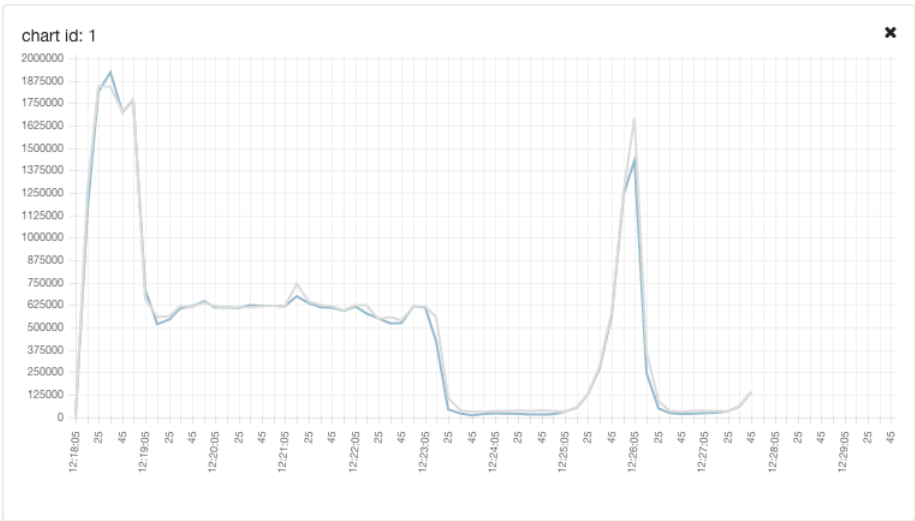
  On the other hand, if a participant should join the conversation some time after the other participants, and if the *no shift* option is selected, it will not be able to detect this. When this option is checked, all the statistics from every participant will be plotted at the first detected data sample time.

- **Sample interval size:** As already presented in Section 4.1.4, the *sample interval size* is a term used for computing the average of a x number of samples. For example: Assuming a Chrome statistics file including 100 samples, then *sample interval size* of $x = 10$, computes first the average of sample number *1 - 10*, then *11 - 20*, and so on. Accordingly, this will reduce the number of data samples in the charts.

  The *sample interval size* enables the end-user to customize the number of data samples each chart should plot. Figure 4.11 illustrates two charts, including the same network statistics, but the *sample interval size* differs. As shown, Figure 4.11a has a much smoother plot than the chart illustrated in Figure 4.11b. However, in Figure 4.11b, one can easily detect peaks, which can be used to detect quality deteriorations.

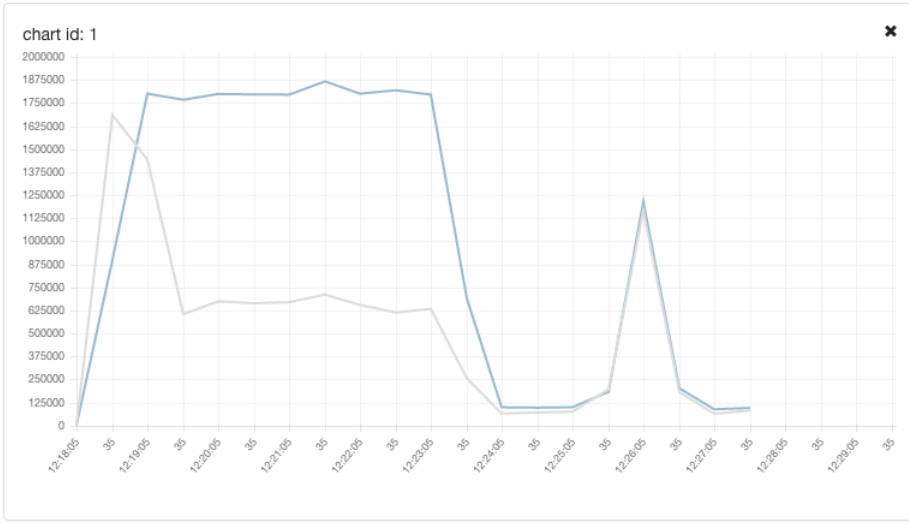


(a) *No shift* checkbox is not checked.

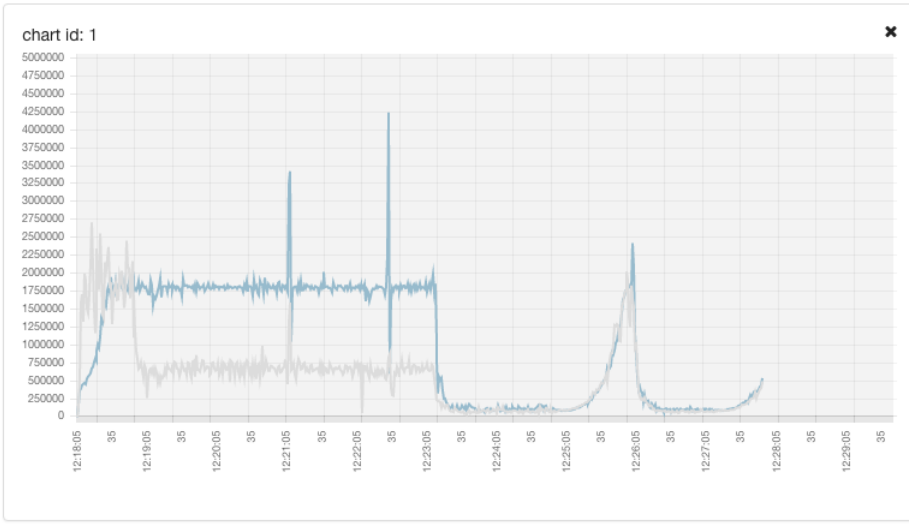


(b) *No shift* checkbox is checked.

Figure 4.10: Illustration of how the *no shift* checkbox impacts chart plotting using the same statstics [39].



(a) Illustrates a chart, and the setting for the *sample interval size* is equal to 30 seconds.



(b) Illustrates a chart, and the setting for the *sample interval size* is equal to 1 second.

Figure 4.11: Illustration of how the *sample interval size* impacts chart plotting the same statstics (*bitsSentPerSecond* and *bitsReceivedPerSecond*) [39].

Chapter 5
Description of the WebRTC-Dashboard

This chapter covers a description of the WebRTC-dashboard, which is available at the URL *http://appear01.item.ntnu.no:3000/*. As previously described in Section 3.1, the WebRTC-dashboard is divided into six parts, and each part is responsible for different functionalities. This chapter covers a description of each part of the WebRTC-dashboard and the associated functionalities. At the end of this chapter, the thesis discusses the WebRTC-dashboard's limitations and how they may be solved.

5.1 Functionalities

This section will present the functionalities of the WebRTC-dashboard. Figure 5.1 illustrates the WebRTC-dashboard and its parts, which was presented briefly in Section 3.1. In order to simplify the text, from now the term *converstion* encompasses WebRTC-based video conversation, and the term *end-user* encompasses the end-user interacting with the WebRTC-dashboard.

5.1.1 Conversation Handler Panel

As shown in Figure 5.1, the Conversation Handler Panel is located at the top of the WebRTC-dashboard and is the first part of the WebRTC-dashboard the end-user interacts with. The Conversation Handle Panel's responsibility is to manage the (WebRTC-based video) conversations.

Each conversation that is stored on the server is listed by conversation name (in alphabetical order) in the Conversation Handle Panel. In order for the end-user to start an analysis, he/she must select a conversation. To do so, the end-user must click on the radio button next to the conversation name. As illustrated in Figure 5.2, after the end-user has selected a conversation, the Conversation Handle Panel displays the accompanying statistics in respect to the selected conversation. These statistics must also be selected individually in order to use them in an analysis.

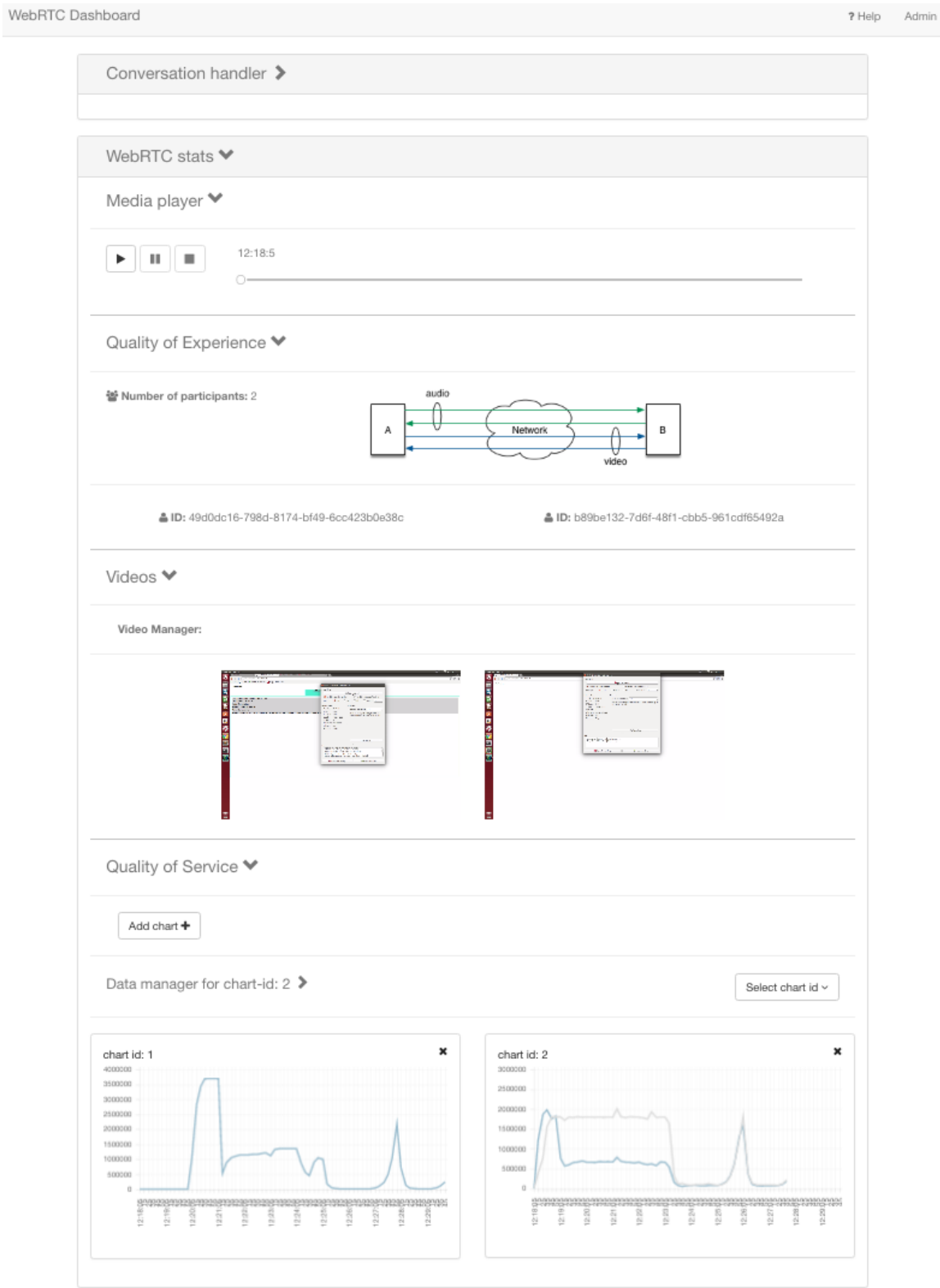


Figure 5.1: Screenshot of the WebRTC-dashboard [39].

Since there is no point of analyzing a conversation without statistics, the Conversation Handle Panel requires that the end-user select at least one Chrome statistics file or one getstats.io statistics file before proceeding with the analysis. Consequently, the Conversation Handler Panel requires that every conversation contain at least one Chrome statistics file or one getstats.io statistics file. In addition, as explained in Section 4.1.4, the Conversation Handle Panel does not allow the end-user to include Chrome statistics in the analysis if there are missing more than $n-1$ Chrome statistics files from a n-party conversation (where $n > 2$).

In addition to Chrome and getstats.io statistics files, the Conversation Handler Panel also displays the accompanying video recordings files (if any). Figure 5.2 illustrates a conversation with included video recordings. Video recording files, however, are not mandatory, and the Conversation Handler Panel does not require the end-user to select any video recordings to continue processing the analysis.

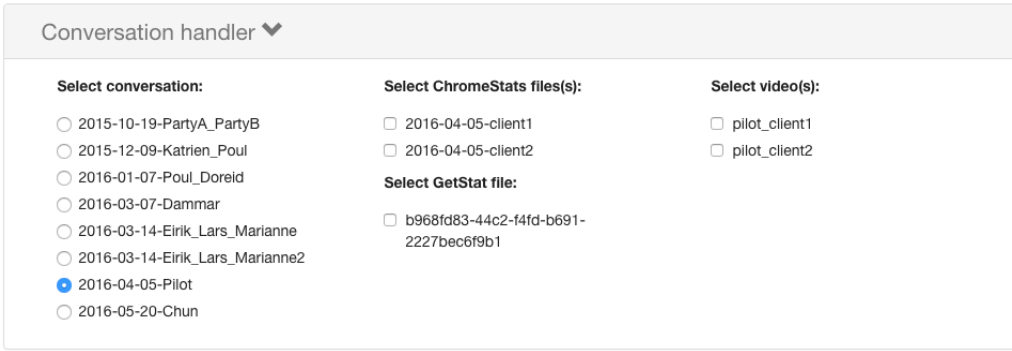


Figure 5.2: Screenshot of the Conversation Handler Panel [39].

After the end-user has selected at least one Chrome statistics file or one getstats.io statistics file, the Conversation Handler Panel displays additional settings (as illustrated in Figure 5.3). There are currently four additional settings supported in the Conversation Handle Panel. Three of them are listed as flexibility features, which were described in Section 4.2.4. The last additional setting is the x-tick location. These four settings are briefly explained as follows:

- **Computation of the *sample interval size*:** As already described previously in Section 4.1.4, and 4.2.4, the *sample interval size* computes the average of x number of samples. In the WebRTC-dashboard, x is set to 10 by default, but as long as it is greater than 0 and does not contain any decimals, the WebRTC-dashboard allows the user to modify the number to whatever he/she desires.

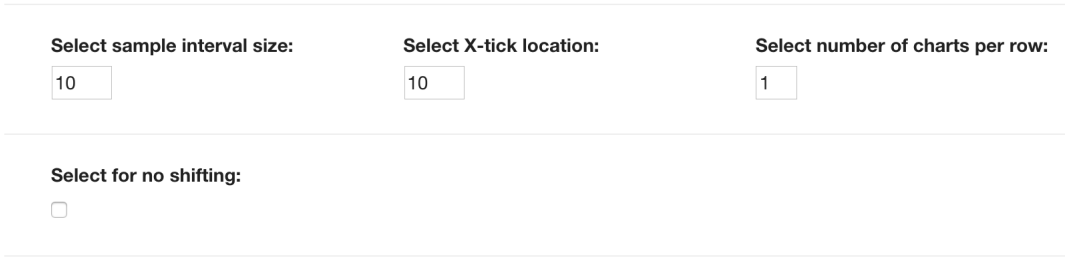


Figure 5.3: Screenshot of the Conversation Handler Panel's additional settings [39].

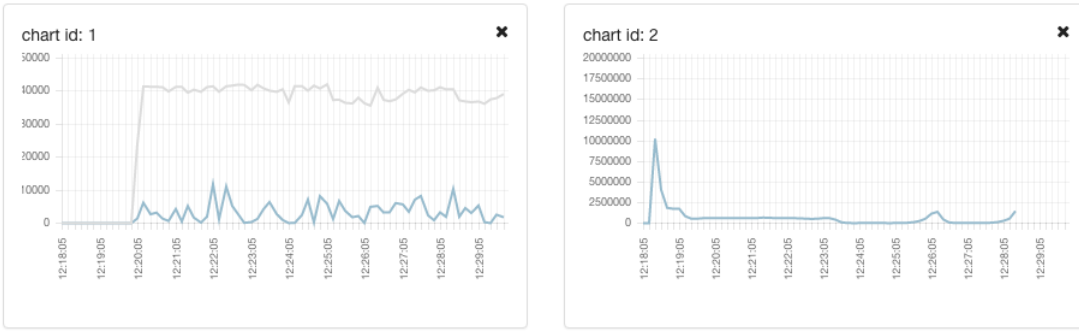
- **Shift or no shift:** Also described previously in Section 4.2.4, the WebRTC-dashboard supports a *No shift* option. When collecting statistics from multiple parties, the internal clock at each of the participant's machines may not be synchronized [46]. The *No shift* option is implemented in order to solve this problem by plotting all the statistics from the first detected sampling time.

- **Selecting the number of charts to be shown on each row:** Is also one of the flexibility features covered in the WebRTC-dashboard, and described in Section 4.2.4. The WebRTC-dashboard supports the function that the end-user can customize the number of charts to be visualized on each row. The WebRTC-dashboard currently supports from 1 to 4 charts on each row. Figure 5.4 illustrates the difference between selecting three charts per row, as opposed to two charts per row.

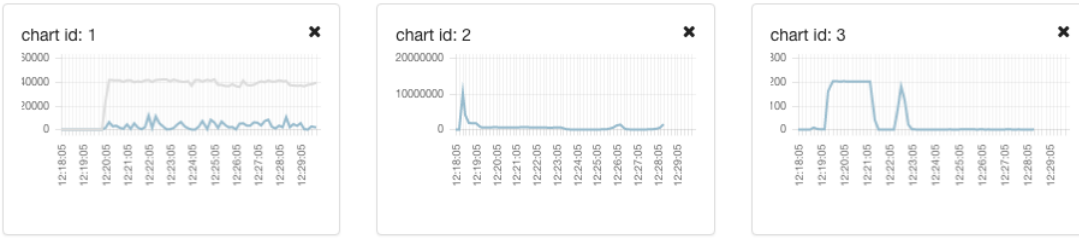
- **X-tick location:** X-tick location is how often the label along the x-axis should appear. This setting is included in order to simplify and reduce the text along the x-axis on charts. In the cases when the end-user selects multiple charts for each row, the charts become smaller and more compact. Consequently, the x-axis gets shorter in smaller charts, and this results in less space for the time stamp labels.

  To solve this issue, this setting allows the end-user to select an x-tick location number, which determines how often the timestamp labels should appear along the x-axis. For example, if the x-tick value $y = 10$, then the timestamp label would appear at every 10 seconds. Figure 5.5 illustrates two charts with different x-tick location values. Even though the timestamp labels in Figure 5.5a are more readable then timestamp labels in Figure 5.5b, Figure 5.5b has a more detailed x-axis.

  Since the timestamp label only need to appear every x time the *sample interval size* is computed, the Conversation Handler Panel requires that the end-user chooses an acceptable x-tick location number. That is, x-tick location number



(a) Two charts per row.



(b) Three charts per row.

Figure 5.4: Illustration of how the chart size is depending on how many charts are lined up in the same row [39].

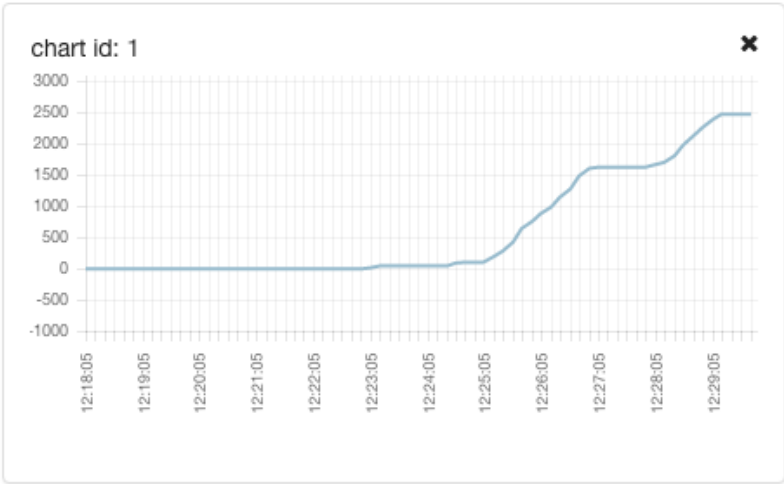
y to be either equal or greater than x, and also fulfill $y * mod(x) = 0$ in order to be accepted.

To proceed with the analysis and to save the selected statistics and settings, the end-user must click on the *Submit* button (illustrated in Figure 5.6) on the bottom of the Conversation Handler Panel. In case some of the settings do not fulfill the requirements of the WebRTC-dashboard, the *Submit* button will not be visible. Also, should an end-user want at a later stage to edit the selected statistics or settings, it is important that the end-user remembers to click the *Submit* button to store the changes.

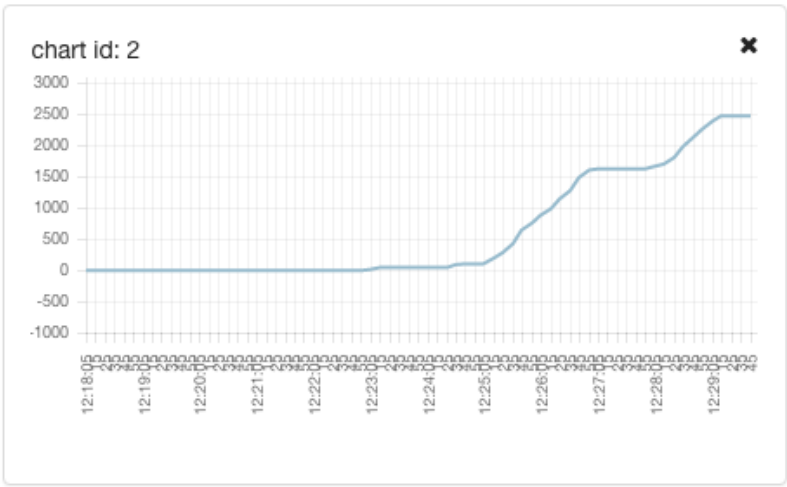
Figure 5.6: Screenshot of the *Submit* button [38].

5.1.2 Quality of Experience Panel

The QoE Panel is located between Media Player Panel and Video Panel, as illustrated in Figure 5.1. It is responsible for displaying subjective user feedback and information about the devices the participants used during the conversation (illustrated in



(a) Illustrates a chart with 60-tick location.



(b) Illustrates a chart with 10-tick location.

Figure 5.5: Illustration of two chart with two difference X-tick location [39].

Figure 5.7). Since getstats.io statistics files contain the subjective user feedback and information about the devices, as long as the conversation selected in the Conversation Handler Panel includes a getstats.io statistics file, the QoE Panel is able to show this information. Which means the end-user does not need to select the getstats.io statistic file in order for the QoE Panel to display its content.

In addition, the QoE Panel is able to visualize the number of participants in a selected conversation. Although it may appear easy to compute the number of participants in a conversation, due to the challenge described in Paragraph *Chrome Statistics are Collected Correctly* in Section 4.1.4, this is not straight forward. The QoE Panel must use the statistics files in order to compute this number. In the case when the selected conversation includes both Chrome statistics and getstats.io statistic files, the QoE Panel calculates the number of participants from the getstats.io file (by using the *Caller* attribute). However, when only Chrome statistics files are available, the QoE Panel uses the number of *Data channels* to compute the number of participants, even though, this may be unreliable due to the challenge presented in 4.1.4.

Included in the QoE Panel is an illustration of how the participants are connected to each other. This illustration is included in order to give the end-user a better understanding of how the *PeerConnections* are connected to each other. However, the current QoE Panel only supports showing this illustration when either a two- or three-party conversation is selected, Illustrative examples of two and three party conversations are shown in figures 3.2 and 4.3 respectively.

Since there is no need to show the device information included in the QoE Panel at all times, the QoE Panel supports a function to collapse and expand the device information. In order to view/expand the device information, the end-user must click on the participant's ID, and likewise, click the participant's ID again to hide/collapse the device information.

Finally, as described in Section 3.3.2, NTNU have conducted a study in order to retrieve more detailed user feedback from the participants in appear.in conversations. These user feedbacks are included in QoE Panel. This feedback is illustrated using a five star rating scale. The QoE Panel supports a questionnaire containing three questions which is shown in Figure 3.5. The questions have five possible answers from 5 (excellent) to 1 (bad), and accordingly an answer of 5 (excellent) corresponds to 5 stars, and an answer of 1 (bad) is equal to one star.

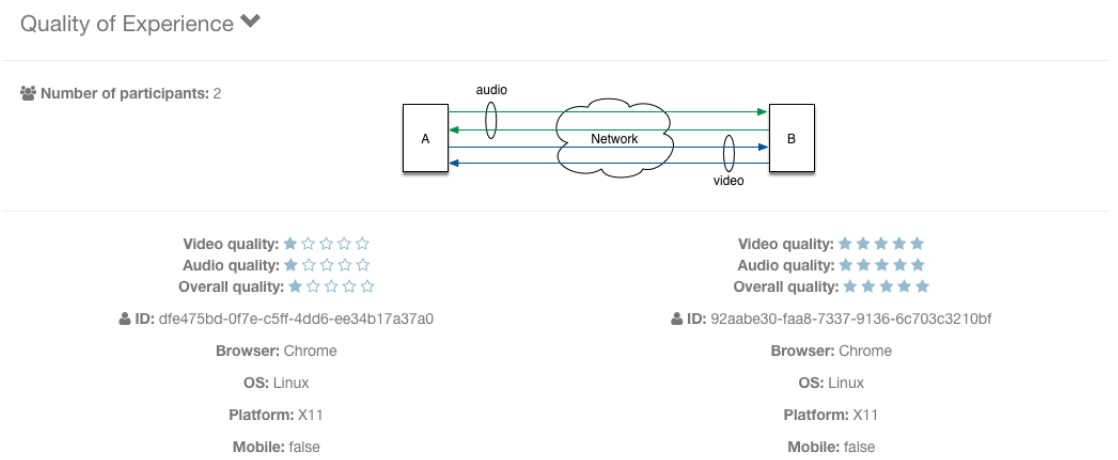


Figure 5.7: Screenshot of the QoE Panel [39].

5.1.3 Video Panel

One of the greatest advantages of the WebRTC-dashboard is that it supports to include videos. As illustrated in Figure 5.1, the Video Panel is located below the QoE Panel. The Video Panel is responsible for replaying videos and handling some video settings. There are currently three video settings included in the Video Panel, and they are: adjust the size of a video, mute a video, and hide a video.

The Video Panel includes these settings in order to have the flexibility to adapt videos for the purpose of analysis. For example, when replaying multiple videos at the same time, it can become difficult to distinguish which audio belongs to which video. Also, if the purpose of the analysis is to analyze the video quality, and not the audio quality, the mute videos option will come to good use. In addition, the Video Panel supports to hide and show the video by clicking on the checkbox next to the name of the video file. This allows the end-user to focus on only specific videos.

Depending on the number of videos selected from the Conversation Handler Panel, the Video Panel limits the number of settings. When more than one video is selected, the Video Panel supports settings such as *mute all* and *resize all*, as shown in Figure 5.8. On the other hand, when only one video is selected, these settings are not included. However, when there is no video selected, the Video panel is hidden.

In order to fit all of the selected videos together onto the same screen window, the end-user may adjust the size of each of the videos. Currently, the Video Panel supports five different sizes, which are: *xsmall, small, medium, large*, and *xlarge*. The video size determines how many videos that can appear on the same row. Because

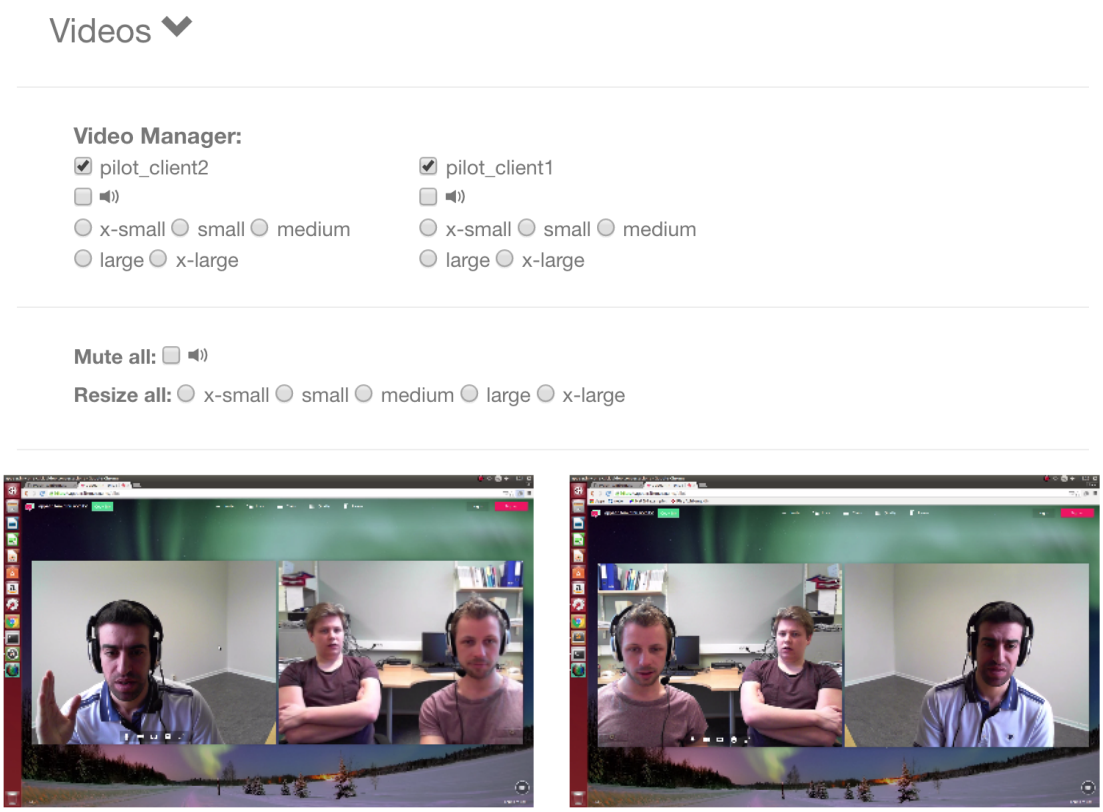


Figure 5.8: Screenshot of the Video Panel [39].

most of the conversations in the WebRTC-dashboard are two-party conversations, the current default size is *medium*.

5.1.4 Quality of Service Panel

In the context of functionalities, the QoS Panel is the most complicated part of the WebRTC-dashboard. As illustrated in Figure 5.1, the QoS Panel is located at the bottom of the WebRTC-dashboard. It is responsible for adding, removing, and plotting charts with network statistics retrieved from Chrome's WebRTC internal interface and getstats.io. These actions are discussed in the following:

- **Adding charts:** It is simple to add a chart to the WebRTC-dashboard. To add a new chart, the end-user must click on the *Add chart* button (which is illustrated in Figure 5.9), which instructs the QoS Panel to create a new empty chart. Every generated chart holds an ID, which is used to distinguish a chart

from another when modifying them. When the end-user clicks on the *Submit* button in Conversation Handler Panel, the chart ID resets to one, and each time the end-user adds a chart to the QoS Panel, the ID increases with one.

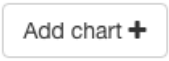


Figure 5.9: Screenshot of the *Add chart* button [39].

- **Modifying charts:** In order to modify the content of a chart, the end-user must first find the associated ID for that chart (the ID is located in the top left corner of each chart). After the end-user has identified the ID, the end-user must select the associated ID using the drop down button *Select chart id* illustrated in the upper right corner of Figure 5.10. This action sets the chart with the selected ID as the *current editing chart*, and its ID will show at the upper part of the panel (as illustrated in Figure 5.10: *Data manager for chart-id: 1*).

  Each time the end-user creates a new chart, the chart automatically becomes the *current editing chart*. This means that the end-user does not need to select the ID of the last created chart in order to modify the chart, this will be automatically set by the QoS Panel itself.

  Furthermore, to plot the charts, the end-user must navigate to the applicable network statistics by expanding and collapsing the statistics titles (such as *ChromeStats*, *GetStats*, *From: X* and *To: Y* as illustrated in Figure 5.10) and then select the associated checkbox.

- **Removing charts:** It is also simple to remove a chart. To remove a chart, the end-user must click on the × symbol at the upper right corner. However, when the end-user removes the *current editing chart* some logic must be added in order to prevent the end-user from editing a non-existing chart.

  At times when the end-user removes the *current editing chart*, the QoS Panel automatically changes the *current editing chart*. Depending on the number of charts existing in the QoS Panel, and the number of the ID of the removed chart, the QoS panel alternates between changing the *current editing chart* to a chart with either a higher or lower chart ID.

  In the cases when the QoS Panel includes of multiple charts, the QoS Panel will always select the chart with the closest ID that is lower than the *current editing chart's* ID. For example, assuming that QoS Panel holds three charts with respectfully IDs *1, 2,* and *3*, then, if the *current editing chart* is the chart with ID equal to *2*, and is removed, then the new *current editing chart* will be the chart with ID *1*.

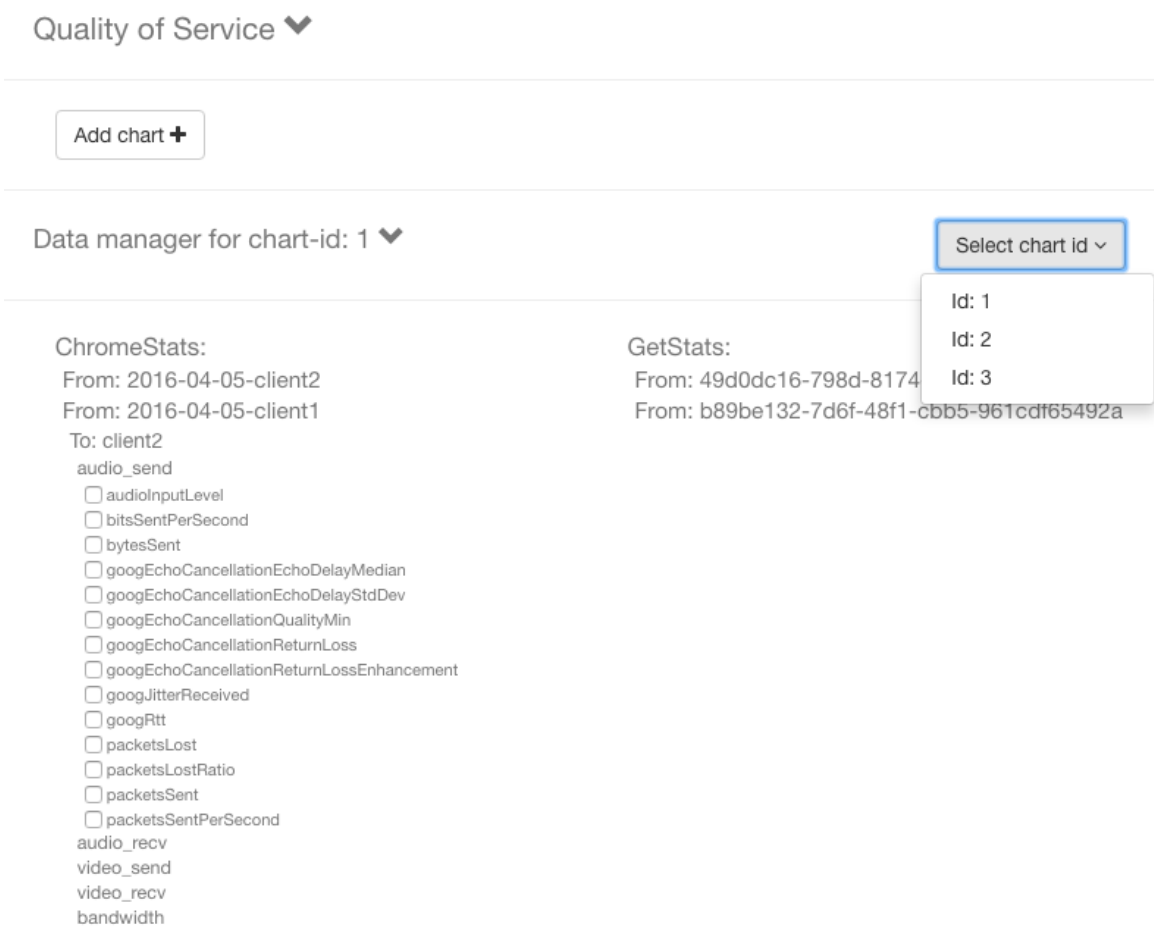


Figure 5.10: Screenshot of the section in which the end-user can modify the content of the charts on the QoS Panel [39].

On the other hand, if the *current editing chart* should have the lowest ID, the QoS Panel will select the chart with the closest ID that is higher than the *current editing chart's* ID. Using the same example as described above: if the chart with an ID equal to *1* is removed, then the new *current editing chart* will be chart with an ID equal to *2*.

Finally, if the last chart is removed from the analysis, the QoS Panel automatically hides the section where the end-user can edit the charts. It is hidden until the end-user creates a new chart by clicking on the *Add chart* button.

The main purpose of the QoS Panel is to give the end-user flexibility to add, remove, and modify charts, but equally important; to view the charts. Each chart may have as many plots as the end-user wishes to, however, the *chart.js* (presented in Section 4.1.3) which is used to generate the charts in the QoS Panel, uses seven default colors to plot statistics. When more than seven plots are included in a chart, *chart.js* random generates a color for each the additional plot, which means that there is a possibility of two plots having the same color [9].

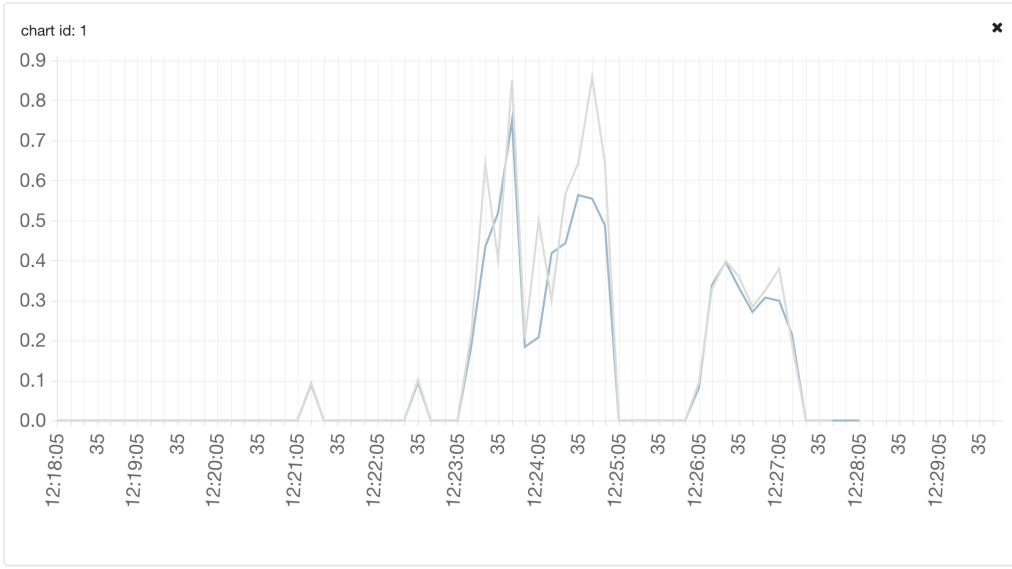


Figure 5.11: Illustration of charts that plot the *PacketsLostRatio* (Equation 4.2 and 4.5) retrieved from Chrome statistics file and getstats.io statistics file [39].

One of the greatest advantages of the QoS Panel is that it can plot statistics from both Chrome statistics and getstats.io statistics files into the same chart. Figure 5.11 illustrates two plots, one of the *PacketsLostRatio* from Chrome statistics (grey),

and the other one of the *PacketsLostRatio* from getstats.io statistics file (blue). This merging-feature is not supported in either Chrome's WebRTC internal interface or getstats.io, which is a great advantages for the WebRTC-dashboard.

One of the flexibility features described in 4.2.4, allows the end-user to select the number of charts to be shown on each row. In addition to this, in order to save space and fit as many charts onto the screen size as possible, the charts on the QoS Panel hide the chart's legends. Legends are the text that tells the end-user what the chart is plotting. However, it is important to note that they are not removed, but only hidden! In order to view the legends, the end-user must hold his/hers computer mouse over the plots. As illustrate in Figure 5.12, when the end-user places his/hers mouse over the plots, a tooltip appears. The tooltip includes the legend information.

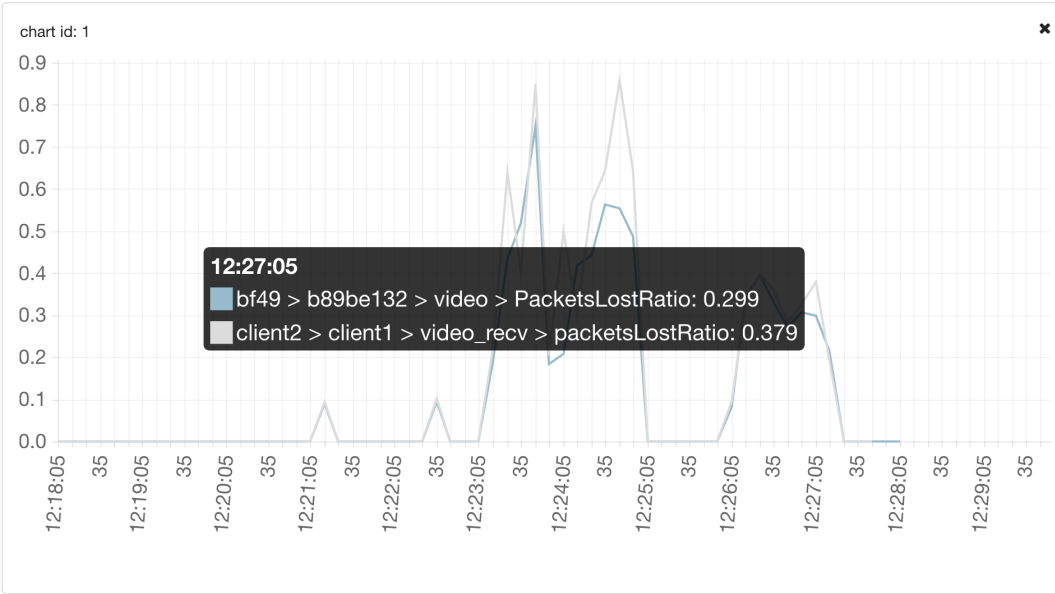


Figure 5.12: Illustration of a chart that plots the *PacketsLostRatio* (Equation 4.2 and 4.5) retrieved from Chrome statistics and getstats.io statistics files, and included the tooltip with legends information [39].

5.1.5 Media Player Panel

As illustrated in Figure 5.1, the Media Player Panel is located above QoE Panel. The Media Player Panel is responsible for play, pause, stop and drag (slider) actions. These actions enable the end-user replay, pause and stop the video and audio recordings, and statistics in charts. The drag action is included in order for the end-user to

"drag" to a specific time in the video and audio recording. These actions are described in the following.

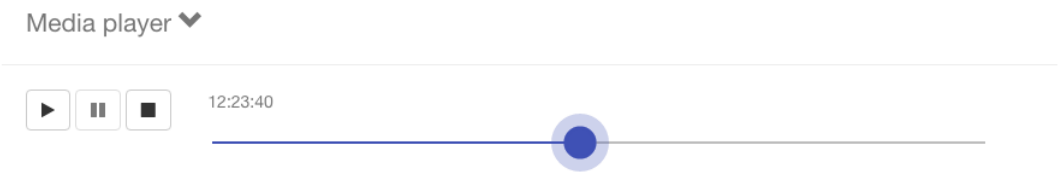


Figure 5.13: Screenshot of the Media Player Panel [39].

- **Play:** The play action is triggered when the end-user clicks on the *Play* button, which is illustrated in Figure 5.13. When the *Play* button is triggered, the video and audio recordings, and statistics in the generated charts start playing. Even though there are no charts generated, the Media Player Panel can still play video and audio recordings. This means that the Media Player Panel does not require charts in order to replay video and audio recordings.
- **Pause:** The pause action is enabled when the end-user clicks on the *Pause* button, which is illustrated in Figure 5.13. This action pauses the video and audio recordings, and the statistics in the generated charts.
- **Stop:** The stop action is triggered when the end-user clicks on the *Stop* button, which is also illustrated in Figure 5.13. When the *Stop* button is clicked, the video and audio recordings stops and jumps back to the beginning of the recording, while the statistics complete its plots.
- **Drag:** The drag action allows the end-user to drag a draggable element along the slider, which is illustrated as the round blue element in Figure 5.13. The slider represents the length of the video and audio recording, and by, for example, dragging the draggable element to the middle of the slider, the video recordings will "jump" to the middle of the conversation, and the statistics will plot halfway of the plot. The draggable element can be dragged both back and forth along the slider.

  In addition, to achieve a better user interface, the end-user can use the arrow keys on the keyboard to move the draggable element along the slider. The two keys enabled is the right arrow and the left arrow. By pressing these keys, the draggable element "jumps" along the slider in respect of the direction of the key. To avoid user confusion, these keys are only activated when the WebRTC-dashboard is playing or paused the video and audio recordings and the statistics in the charts. Also, when one of these keys are triggered while playing, it triggers the pause action, and will not continue to play until the end-user clicks on the *play* button.

Another important fact worth mentioning is the case when the video and audio recording durations are shorter than the duration of the conversation (statistics in the charts). In this case, the Media Player Panel stops the video and audio recordings at the end and continues to plot the statistics in the charts. Also, similarly, the Video Panel and the Media Player Panel are hidden when there is no video selected in the Conversation Handler Panel.

5.1.6 Additional Functionalities

During the development process, it was found that two more functionalities were needed to be added to the WebRTC-dashboard in order to achieve a more satisfactory result. Additional functionalities are functionalities that were not included in the original plan of the WebRTC-dashboard, but due to solving important problems, were included. Both of the additional functionalities are located on the navigation bar (as illustrated at the upper right corner in Figure 5.1) and are called the Help Modal, and Admin Modal, and are explained below.

Help Modal

The Help Modal is a user guide for how to use the WebRTC-dashboard. It includes a short explanation of all the different parts of the WebRTC-dashboard and their functionalities. The Help Modal tells the end-user how to use the WebRTC-dashboard and to utilize its full potential.

Admin Modal

As presented in Paragraph *Uploading WebRTC-Based Conversation Statistics Files* in Section 4.1.4, due to the inconvenient process of uploading WebRTC-based conversation statistics, having an Admin Modal was highly prioritized. The Admin Modal supports both adding and removing conversations to and from the WebRTC-dashboard. In order for the end-user to add a conversation, the end-user completes an add-form. As illustrated in Figure 5.14, this add-form consists of a mandatory text input field, and three optional uploading buttons, two for statistics files (Chrome statistics, getstats.io statistics,) and one uploading button for video recordings. The required text input field should include the name of the conversation. Even though the format of the name is not required, it is recommended to have the format: *YYYY-MM-DD-name*.

Currently, the Admin Modal only supports to upload Chrome statistics files, getstats.io statistics files and video recordings files. In order for the Admin Modal to accept the files, it is important that they have the right extension. Which are Chrome statistics files and getstats.io statistics files must end with a .JSON extension, and video recordings files must end with .mp4.

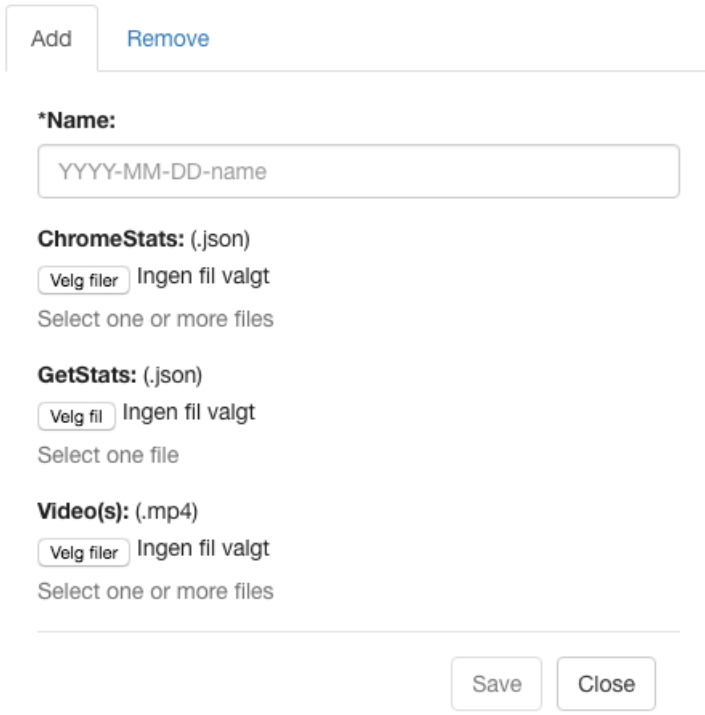


Figure 5.14: Screenshot of the add-form in the Admin Modal [39].

In addition, the Admin Modal allows the end-user to upload multiple files within the same uploading button. This means that the end-user can add multiple Chrome statistics files and video recordings into the same upload. This is achieved by selecting several files inside the end-user directory by clicking the files while holding the *ctrl* key on the keyboard. However, since each conversation only has one associated getstats.io statistics file, the Admin Modal only supports to upload one file in the case for getstats.io statistics files.

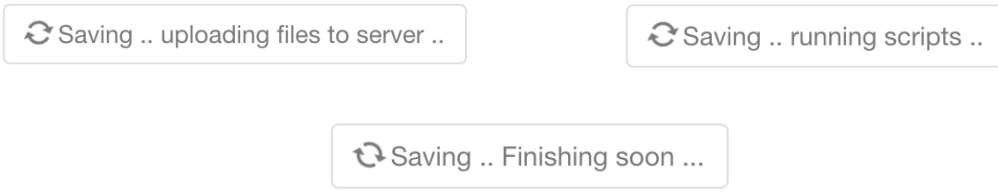


Figure 5.15: Screenshot of the saving button with feedback [39].

As soon as the *Save* button is triggered, the Admin Modal starts to transfer the files from the client to the server. If the files are large, which most video recordings files are, this transfer can take some time. In order to minimize waiting time and irritation, the Admin Modal gives the end-user feedback of how far in the uploading process it has come. See Figure 5.15 for illustration.

The second functionality the Admin Modal supports is to remove the conversation from the WebRTC-dashboard. To remove a conversation, the end-user must select the conversation(s) he/she wishes to remove, which is done by selecting the checkbox next the conversation name. In order to complete the removal, the end-user must click on the *Save* button. Figure 5.16 illustrates the how the remove-form in the Admin Modal.

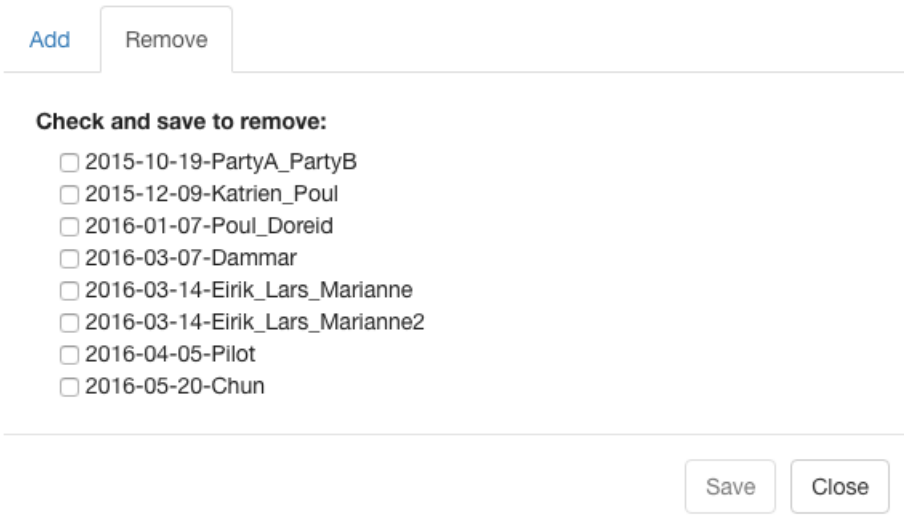


Figure 5.16: Screenshot of the remove-form in the Admin Modal [39].

5.2 Limitations

In this section, the limitations of the WebRTC-dashboard will be presented. Because the WebRTC-dashboard depends on retrieving network statistics from both getstats.io and Google Chrome's WebRTC internal interface, the WebRTC-dashboard also suffers from the same limitations which these interfaces have, which will be covered in this section. Furthermore, this section will also cover the GUI limitations of the WebRTC-dashboard.

5.2.1 Limited Number of Sample Points in Chrome Statistics

One of the Google Chrome's WebRTC internal interface limitations is that it only stores the last 1000 data samples. As highlighted in Section 3.3.1, if the conversation last longer than 1000 seconds, only the last 1000 data samples are recorded. However, getstats.io does not suffer from the same limitation. When the end-user plots a chart with statistics retrieved from both platforms, the lost data samples from Google Chrome's WebRTC internal interface are added as zero in the charts.

5.2.2 Chrome Statistics Sampling Time

Previously presented in Section 3.3.1 and described in [46], Chrome statistics files are collected per browser, which implies that the internal clock at each machine may not be synchronized with the other participant's machines. In order to perform an analysis for a multi-party conversation, it requires that these clocks are synchronized. However, this is not always the case, and when the statistics are plotted in the charts, they look shifted. As illustrated in Figure 4.10 and explained in Section 4.2.4, the WebRTC-dashboard supports a *No shift* option to handle these cases.

5.2.3 Getstats JSON Format may Change

As presented previously in Section 4.1.4, the getstats.io does not support a standardized JSON format, which implies that the format can change. If the getstats.io JSON format changes drastically, the WebRTC-dashboard will not be able to support to retrieve data from getstats.io statistics files. It is impossible for the WebRTC-dashboard to predict the future JSON format. Until getstats.io launches a standardization of their JSON file, this will be a limitation of the WebRTC-dashboard.

However, WebRTC-dashboard supports to minimize the problem when format changes occur, by implementing a Bash script for retrieving getstats.io statistics. As highlighted in Section 4.1.4, the WebRTC-dashboard is not directly dependent on the JSON format of getstats.io statistics files. Instead, it uses a Bash script in order to retrieve the information and store the data into .txt files. These .txt files are used by the WebRTC-dashboard.

If the WebRTC-dashboard had been directly dependent on the JSON format of getstats.io, there would be much more work and many more files to modify in the case of a getstats.io JSON file update. The current WebRTC-dashboard only requires the Bash script to be modified each time getstats.io updates its JSON format, which means that the one maintaining the WebRTC-dashboard only needs to know how to write a script in Bash. Even though, the WebRTC-dashboard still requires the Bash script to be modified each time getstats.io updates its JSON format, using Bash

script was one of the best available solutions when facing this limitation, and was, therefore, selected for this WebRTC-dashboard.

5.2.4 GUI Limitations

GUI limitations are included in this section in order to highlight the limitations of the UI of the WebRTC-dashboard. In addition, this section will also cover how the WebRTC-dashboard is trying to meet these limitations.

View Charts and Video Within the Size of the Screen

In the context of conducting an analysis using the WebRTC-dashboard, it is important that the end-user can see both the charts (in the QoS Panel) and the video recordings (in the Video Panel) within the size of the screen. In order to achieve this, the WebRTC-dashboard hides all the unnecessary elements from view, while the video recordings are playing and the statistics in the charts are plotting, such as; the option for the user to modify the video recordings and the content of the charts. Although, the size of the end-user's computer screen has the greatest impact on how many charts and video recordings that can be seen at the same time, the WebRTC-dashboard is trying to solve that by including settings that allow end-user to customize the size of their charts and videos.

Limited Screen Size

Generally, when considering a dashboard, one may think that all the elements of a dashboard should be visualized at all times, like a dashboard inside a car. However, as illustrated in Figure 5.1, the WebRTC-dashboard is a vertical dashboard, which means that all the included parts are stacked on top of each other vertically. Consequently, the user must scroll up and down the WebRTC-dashboard in order to retrieve the wanted information. In order to help solve this problem, the WebRTC-dashboard supports a functionality to hide and view the different parts of the WebRTC-dashboard by collapsing and expanding them, which gives the end-user the ability to customize the space of the WebRTC-dashboard.

Chapter 6
Conclusion and Future work

In this chapter, a conclusion of this master thesis will be presented. The conclusion will present a summary of the WebRTC-dashboard and highlight what it has been accomplished. This is followed by a section introducing the future work associated with the WebRTC-dashboard. The future work will include suggestions on how to remedy some of the WebRTC-dashboard's limitations and some nice-to-have features.

6.1 Conclusion

As WebRTC continues to evolve with new supporting technologies, such as appear.in, Google Hangouts, and multiple other similar applications, it becomes necessary to identify the factors which impact the QoE when utilizing these applications. This master thesis has presented the implementation of a WebRTC-dashboard. The WebRTC-dashboard utilizes session-related data from analyzing platforms in order to analyze n-party WebRTC-based video conversations. In addition, the WebRTC-dashboard can replay session-related data to find correlations between technical and non-technical factors and can identify the factors which impact the QoE. Furthermore, the WebRTC-dashboard allow the end-users to interact and customize the analysis for his/hers purpose.

One of the greatest advantages of the WebRTC-dashboard over the other analytic interfaces is that it supports to combine network parameters, subjective user feedback from different analytic platforms (getstas.io and Chrome's WebRTC internal interface), and video recordings and replay them. Due to the numerous network parameters found in Chrome statistics and getstats.io statistics files, and additional data, it has become necessary to obtain an analyzing tool to analyze all the parameters in an efficient way. The WebRTC-dashboard supports an easy and efficient way of analyzing statistics and opens the possibility to identify new correlations between impacting parameters.

To prevent users from getting unsatisfied with a WebRTC-based service, it is important to gain an in-depth understanding of the numerous technical and non-technical factors that may influence the QoE. Without an analyzing tool, such as the WebRTC-dashboard, gaining this knowledge is almost impossible. Even though both Google Chrome's WebRTC internal interface and getstats.io supports their dashboards with visual representations of their data, the WebRTC-dashboard is still needed. This is because the WebRTC-dashboard solves problems that neither getstats.io and Google Chrome's WebRTC internal interface does. For example, the WebRTC-dashboard solves one of the problems described in [46], *Imprecise sampling time*, by supporting the *No shift* option. In addition the WebRTC-dashboard supports to replay video and audio recordings together with network statistics and supports the end-user to customize its analysis with flexibility features. Also because JSON files contain numerous of lines of data, it is impossible for analysts to process all the samples without an efficient tool. Even though it is possible to plot all the parameters with help from other plotting tools, due to the high number of parameters, this cannot be done in an efficient way. Because of this, there has been a real need for an analyzing tool, such as the WebRTC-dashboard.

This master thesis has tried to highlight the great potential the WebRTC-dashboard has and what it can accomplish. It has clarified and discussed what challenges that may be encountered when developing such a solution, and how to handle these challenges. Even though the WebRTC-dashboard includes many excellent functionalities that help solve many limitations, from a realistic point of view, since the WebRTC-dashboard was developed during a limited period, it only contains the most significant and necessary functionalities. With more time and research, the WebRTC-dashboard could have contained more functionalities, such as the functionalities that are described in Future Work (Section 6.2). In the future, hopefully, the developer who continues to work on the WebRTC-dashboard or implements a similar analyzing tool from scratch can learn from the challenges and limitations this master thesis has presented.

6.2 Future Work

This section covers suggestions on how to remedy some of the WebRTC-dashboard's limitations and some nice-to-have features. Due to prioritizing and time limitations, these implementations were not included into the current WebRTC-dashboard.

6.2.1 Better Synchronization of Video Recordings, Slider, and Charts

An issue arises when trying to synchronize the video recordings, charts, and slider together. This is because video recordings are not handled in the same way as the

slider and the charts. Video recordings are supported by HTML5 media functionalities, which plays, stops, and pauses the video recordings each time the respectful buttons are triggered. On the other hand, each time the defined countdown period is finished, the chart plots a new data sample, and the draggable element “jumps” one step along the slider. The countdown starts each time the end-user triggers play and gets nullified when the end-user clicks pause, stops or drags the draggable element. As previously described in Section 4.1.4, this can cause the synchronization to be out of sync between the video recordings, slider, and charts.

Even though the slider and charts will be synchronized, the problem is to figure out how to synchronize the video recordings to the charts and the slider. Section 4.1.4 has already suggested two solutions, which are to store the number of seconds counted down for each "jump" in a variable, or push back the video recordings back to the last step. Both of these implementations are possible to accomplish, but from an implementation perspective, the second solution may be the easiest one to implement. Each time the end-user triggers the *play*-button it sends the step value as a parameter and pushes the video back to the step. However, this solution may cause the video recordings to be less smooth than when implementing the first solution.

6.2.2 Admin Access

In order to limit the access in respect to who can add and remove conversations in the WebRTC-dashboard, an admin access is required. This means that the people who have admin privileges are the only persons that can modify the number of conversations included in the WebRTC-dashboard. Also, this prevents the issue of someone who is "not welcome" to change the WebRTC-dashboard without "permission". Even though the URL to the WebRTC-dashboard is intricate than most URLs, one never knows when this can happen. Accordingly, an admin access features could solve this issue and may be appreciated.

To achieve the admin access functionality, the WebRTC-dashboard must implement a type of login process, such as an admin login page with a password. To accomplish a secure login page, the WebRTC-dashboard must require security settings and these security settings must be handled carefully and correctly. Whoever has the right password can access the admin privileges. One solution would have been to implement the WebRTC-dashboard with *Django*[1]. *Django* is a web platform, which includes a built-in admin functionality, however, this should have been included at the beginning of the implementing process of the WebRTC-dashboard.

[1]Address: https://www.djangoproject.com/

6.2.3 Support for Other WebRTC-Based Applications

In order to support more flexibility, the WebRTC-dashboard be able to support to utilize session-related data from other analytic platforms. In the beginning of this thesis, a WebRTC-based application, *Hello* was presented in Section 2.1.4. *Hello* is Mozilla Firefox's WebRTC solution, which means that it is only accessible when using Mozilla Firefox web browser. For the WebRTC-dashboard to support and analyze conversations performed on *Hello*, it is dependent upon that there is an analytic platform it can retrieve data from.

However, similarly Google Chrome, the web browser Opera supports a WebRTC internal platform[2]. If the WebRTC-dashboard implements to support session-related data retrieved from Opera's WebRTC internal interface, it can analyze WebRTC-based conversation that has been processed in Opera web browser.

6.2.4 Testing

As already presented in Section 4.1.2, testing was conducted during the implementation process. However, due to limited time, only a limited number of the functional requirements were tested. As a consequence, to ensure the reliability of all the functional requirements, more testing should be preformed. Since the Video Panel and Admin Modal was the last included functionalities, these require additional testing.

6.2.5 Search Function in QoS Panel

Due to the great number of network parameters covered by Chrome statistics and getstats.io statistics files, a nice-to-have feature is a search function in the QoS Panel. In the case when the end-user does not know exactly where to locate the network parameters he/she wants to plot, a search function can come in quite handy. Using a search function, the end-user can easily search for the network parameters, as he/she wants without knowing where to find them. Also, a search function would also minimize the number of clicks the end-user must perform to find the wanted parameters, which enables better UI. Since the WebRTC-dashboard already utilize *Bootstrap* (presented in Section 4.1.3), the WebRTC-dashboard can use *Bootstraps* search function [8].

6.2.6 Support Dual Y-axis

Another nice-to-have feature would be dual Y-axis charts. Dual Y-axis charts are used to identify quickly and to validate the relationship between two variables with different magnitudes and scales of measurement. Since there are network parameters

[2]Address: opera://webrtc-internals

that operate on different scales, they are difficult to measure against each other in the same chart. However, since the QoS Panel supports a function that the end-user can plot as many network statistics into the same chart as he/she wants, it may be a challenge to identify which statistics should depend on which Y-axis.

Chart.js, which is used to generate the charts in the QoS Panel, launched a second version April 9th 2016 [9]. Until its release, *Chart.js* did not support dual Y-axis. Since dual Y-axis is now supported by *Chart.js*, the WebRTC-dashboard can easily include this feature in the future. However, there remains the problem of how to determine which network statistic should belong to which Y-axis.

6.2.7 Store the Charts

When performing an analysis, the end-user most likely writes at least one report based on what was found (or not found) in the analysis. Most likely, a report also consists of figures to explain the findings. Since the purpose of the WebRTC-dashboard is to use it for analysis, the WebRTC-dashboard should include a way of storing the information the WebRTC-dashboard is presenting, such as charts. Currently, there are no such functionalities, and the user must manually screenshot each of the relevant charts (and other interesting findings), give it a proper name and store the information for use in a report. For the future version of the WebRTC-dashboard, a storing functionality may be desirable.

References

[1] (2016). About. https://git-scm.com/about. [Online; accessed 16-April-2016].

[2] (2016a). Angular material. https://material.angularjs.org. [Online; accessed 18-April-2016].

[3] (2016b). Angularjs. https://angularjs.org/. [Online; accessed 15-April-2016].

[4] (2016). Anymeeting enables you to host webinars or meetings, easily and affordably. https://www.anymeeting.com/. [Online; accessed 23-April-2016].

[5] (2016). Appear.in - create a room and invite up to 8 friends. https://appear.in/. [Online; accessed 11-April-2016].

[6] (2016). *Based on conversation with responssible professor, and supervisor.*

[7] (2016). Bash reference manual. https://www.gnu.org/software/bash/manual/bashref.html. [Online; accessed 26-April-2016].

[8] (2016). Bootstrap is the most popular html, css, and js framework for developing responsive, mobile first projects on the web. http://getbootstrap.com/. [Online; accessed 16-April-2016].

[9] (2016a). Chart.js - simple, clean and engaging charts for designers and developers. http://www.chartjs.org/. [Online; accessed 16-April-2016].

[10] (2016b). Chart.js - version 2.0.0. https://github.com/nnnick/Chart.js/releases. [Online; accessed 16-April-2016].

[11] (2016). Css developer guide. https://developer.mozilla.org/en-US/docs/Web/Guide/CSS. [Online; accessed 16-April-2016].

[12] (2016). Css with superpowers. http://sass-lang.com/. [Online; accessed 16-April-2016].

[13] (2016). Data-driven documents. https://d3js.org/. [Online; accessed 16-April-2016].

[14] (2016). Font awesome - the iconic font and css toolkit. https://fortawesome.github.io/Font-Awesome/. [Online; accessed 17-April-2016].

[15] (2016). Google hangouts. https://hangouts.google.com/. [Online; accessed 11-April-2016].

[16] (2016). Grunt- the javascript task runner. http://gruntjs.com/. [Online; accessed 16-April-2016].

[17] (2016a). Gulp vs grunt. why one? why the other? https://medium.com/@preslavrachev/gulp-vs-grunt-why-one-why-the-other-f5d3b398edc4#.pk45hlscy. [Online; accessed 16-April-2016].

[18] (2016b). Gulp.js. http://gulpjs.com/. [Online; accessed 15-April-2016].

[19] (2016). An introduction to gulp.js. http://www.sitepoint.com/introduction-gulp-js/. [Online; accessed 16-April-2016].

[20] (2016). Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction. [Online; accessed 15-April-2016].

[21] (2016). Make your data come alive - highcharts makes it easy for developers to set up interactive charts in their web pages. http://www.highcharts.com/. [Online; accessed 16-April-2016].

[22] (2016). Messenger. https://www.messenger.com. [Online; accessed 11-April-2016].

[23] (2016). Mvc architecture. https://developer.chrome.com/apps/app_frameworks. [Online; accessed 07-May-2016].

[24] (2016a). node.js. https://nodejs.org/en/. [Online; accessed 15-April-2016].

[25] (2016b). Node.js - introduction. http://www.tutorialspoint.com/nodejs/nodejs_introduction.htm. [Online; accessed 18-April-2016].

[26] (2016). npm - build amazing things. https://www.npmjs.com/. [Online; accessed 18-April-2016].

[27] (2016). php.net. http://php.net/. [Online; accessed 15-April-2016].

[28] (2016). Qualinet. https://www.qualinet.eu. [Online; accessed 25-April-2016].

[29] (2016). Quality of service networking. http://docwiki.cisco.com/wiki/Quality_of_Service_Networking. [Online; accessed 24-April-2016].

[30] (2016). Say hello to facetime. http://www.apple.com/no/mac/facetime/. [Online; accessed 11-April-2016].

[31] (2016). Skype. http://www.skype.com/. [Online; accessed 11-April-2016].

[32] (2016). Telefonica - in brief. https://www.telefonica.com/en/web/about_telefonica/in-brief. [Online; accessed 26-April-2016].

[33] (2016). Trello. https://trello.com/. [Online; accessed 19-April-2016].

[34] (2016). W3c. https://www.w3.org/. [Online; accessed 16-April-2016].

[35] (2016a). Webex - work where you are. https://www.webex.com/. [Online; accessed 23-April-2016].

[36] (2016b). Webrtc. https://webrtc.org/. [Online; accessed 11-April-2016].

[37] (2016). Webrtc. https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC. [Online; accessed 23-April-2016].

[38] (2016c). Webrtc 1.0: Real-time communication between browsers. https://www.w3.org/TR/webrtc/. [Online; accessed 23-April-2016].

[39] (2016d, may). *WebRTC-Dashboard*. Address: appear01.item.ntnu.no:3000.

[40] (2016). Webrtc faqs. https://tokbox.com/about-webrtc. [Online; accessed 23-April-2016].

[41] (2016c). Why the hell would i use node.js? https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js. [Online; accessed 18-April-2016].

[42] Albing, C., J. Vossen, and C. Newham (2007). *bash Cookbook: Solutions and Examples for bash Users*. " O'Reilly Media, Inc.".

[43] Ammar, D., J. Brochet, T. Begin, I. Guerin-Lassous, and L. Noirie (2012, October). Knowledge-Based Admission Control: A Real-Time Performance Analysis. In *37th IEEE Conference on Local Computer Networks (LCN 2012)*, Clearwater (Florida), United States. Demo - http://www.ieeelcn.org/prior/LCN37/lcn37demos/LCNDemos12_Ammar.pdf.

[44] Ammar, D., K. De Moor, and P. Heegaard (2016). Quality of experience - assessment of webrtc based video communication. ERCIM News 2016(105) (2016).

[45] Ammar, D., K. De Moor, M. Xie, and P. Heegaard (2016). Video qoe killer and performance statistics in webrtc-based video communication. in 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), Jul. 2016.

[46] Ammar, D., P. Heegaard, M. Xie, K. De Moor, and M. Fiedler (2016). Revealing the dark side of webrtc statistics collected by google chrome. Quality of Multimedia Experience (QoMEX), 2016 Eighth International Conference on, Lisbon.

[47] Berndtsson, G., M. Folkesson, and V. Kulyk (2012). Subjective quality assessment of video conferences and telemeetings. In *Packet Video Workshop (PV), 2012 19th International*, pp. 25–30. IEEE.

[48] Bourque, P., R. E. Fairley, et al. (2014). *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

[49] Brunnström, K., S. A. Beker, K. De Moor, A. Dooms, S. Egger, M.-N. Garcia, T. Hossfeld, S. Jumisko-Pyykkö, C. Keimel, M.-C. Larabi, et al. (2013). Qualinet white paper on definitions of quality of experience.

[50] Committee, I. C. S. S. E. S. and I.-S. S. Board (1998). Ieee recommended practice for software requirements specifications. Institute of Electrical and Electronics Engineers.

[51] De Moor, K., F. Mazza, I. Hupont, M. R. Quintero, T. Mäki, and M. Varela (2014). Chamber qoe: a multi-instrumental approach to explore affective aspects in relation to quality of experience. In *IS&T/SPIE Electronic Imaging*, pp. 90140U–90140U. International Society for Optics and Photonics.

[52] Emstad, P. J., P. E. Heegaard, B. E. Helvik, and L. Paquereau (2011). *Dependability and performance in information and communication systems.*

[53] Flanagan, D. (2011). *JavaScript - The definitive guide.*

[54] Gunkel, S., M. Schmitt, P. Cesar, and P. Hughes (2013). A qoe testbed for socially-aware video-mediated group communication.

[55] Gunkel, S. N., M. Schmitt, and P. Cesar (2008, September). Series e: Overall network operation, telephone service, service operation and human factors - quality of telecommunication services: concepts, models,objectives and dependability planning – terms anddefinitions related to the quality of telecommunication services.

[56] Gunkel, S. N., M. Schmitt, and P. Cesar (2015, May). A qoe study of different stream and layout configurations in video conferencing under limited network conditions.

[57] Khirman, S. and P. Henriksen (2002). Relationship between quality-of-service and quality-of-experience for public internet service.

[58] Le Callet, P., S. Möller, A. Perkis, et al. (2012). Qualinet white paper on definitions of quality of experience. *European Network on Quality of Experience in Multimedia Systems and Services (COST Action IC 1003).*

[59] McCarthy, J. and P. Wright (2004). Technology as experience. *interactions 11*(5), 42–43.

[60] Schmitt, M., S. Gunkel, P. Cesar, and D. Bulterman (2014). Asymmetric delay in video-mediated group discussions. In *Quality of Multimedia Experience (QoMEX), 2014 Sixth International Workshop on*, pp. 19–24. IEEE.

[61] Vučić, D. and L. Skorin-Kapov (2015). The impact of mobile device factors on qoe for multi-party video conferencing via webrtc. In *13th International Conference on Telecommunications.*

Appendix A
Software Requirements Specification

A.1 Functional Requirements

A.1.1 System

Table A.1: Functional requirements for the System.

Req. ID	Requirement Description	Priority
1.1	The system shall support to analyze n-party WebRTC-based conversations	High
1.2	The system shall be able to use the data retrieved from getstats.io	High
1.3	The system shall be able to use the data retrieved from Chrome's WebRTC internal interface	High
1.4	The system shall support an easy way of handling if the getstats.io JSON format should change	High

A.1.2 Conversation Handler Panel

Table A.2: Functional requirements for the Conversation Handler Panel.

Req. ID	Requirement Description	Priority
2.1	The end-user shall be able to select WebRTC-based conversation he/she wants to analyze	High
2.2	The end-user shall be able to select the statistics he/she wants to include into the analyze	High
2.3	The end-user shall be able to select x seconds to compute sample interval size	High
2.4	The user shall be able to select no shifting for chart	High
2.5	The end-user shall be able to select z number of charts to show for each row	Medium
2.6	The end-user shall be able to select X-tick location	Medium
2.7	The X-tick location y must be either equal ot greather than x, and fulfill $y * mod(x) = 0$	Medium
2.8	The x seconds to compute the sample interval size must be greather than 0	Medium
2.9	The z number of charts to show for each row bus be either euqual or between one and four	Medium

A.1.3 QoE Panel

Table A.3: Functional requirements for the QoE Panel.

Req. ID	Requirement Description	Priority
3.1	The end-user shall be able to view the feedback from participant	High
3.2	The end-user shall be able to view the information about the participants device	High
3.3	The end-user shall be able to see additional browser information	Medium
3.4	The end-user shall be able to see star-rating when NTNU's appear.in test server is used	Medium

A.1.4 Media Player Panel

Table A.4: Functional requirements for the Multimedia Panel.

Req. ID	Requirement Description	Priority
4.1	The end-user shall be able to click play button	High
4.2	The end-user shall be able to click pause button	High
4.3	The end-user shall be able to click stop button	High
4.4	The end-user shall be able to drag a slider back and forth	High
4.5	The end-user shall only access multimedia panel if video is included	Medium
4.6	The end-user shall be able to use arrow keys on the keyboard to move the slider forth and back	Low

A.1.5 QoS Panel

Table A.5: Functional requirements for the QoS Panel.

Req. ID	Requirement Description	Priority
5.1	The end-user shall be able to add charts	High
5.2	The end-user shall be able to close charts	High
5.3	The end-user shall be able to plot chart with data	High
5.4	The end-user shall be able to select data from the statistics chosen in conversation handler	High
5.5	The end-user shall not be able to modify a non-existing chart	High

A.1.6 Video Panel

Table A.6: Functional requirements for the Video Panel.

Req. ID	Requirement Description	Priority
6.1	The end-user shall be able to show selected videos	High
6.2	The end-user shall be able to mute all videos	Medium
6.3	The end-user shall be able to hide videos	Medium
6.4	The end-user shall be able to resize all the videos to the same size	Medium
6.5	The end-user shall be able to mute one video at the time	Medium
6.6	The end-user shall be able to resize one video at the time	Medium

A.1.7 Navigation bar

Table A.7: Functional requirements for the Navigation bar.

Req. ID	Requirement Description	Priority
7.1	The end-user shall be able to see admin modal	High
7.2	The end-user shall be able to remove old conversations	High
7.3	The end-user shall be able to add new conversation to analyze	Medium
7.4	The end-user shall be able to see help modal	Medium
7.5	The end-user shall be given feedback of how far in the uploading process it has come	Medium

A.2 Non-Functional Requirements

Table A.8: Non-functional requirements.

Req. ID	**Requirement Description**	**Priority**
8.1	The system shall be supported by desktop Google Chrome web browser	High
8.2	The system shall give the end-user a deeper understanding of what kind of technical- and non-technical factors can influence QoE in a graphical and interactive way	High
8.3	The text in the system should be written in English	High
8.4	The system shall support PC and Macintoshes	High
8.5	The text in the system should be understandable	Medium
8.6	The system shall have a good design, so that the user can easily interact with the web interface	Medium

Appendix B
Statistics

B.1 Google Chrome's WebRTC Internal Interface Statistics

Table B.1: Complete list of statistics supported by Google Chrome's WebRTC internal interface.

Parameter	**Value**	**Media**	**Source**
audioInputLevel	Integer	audio	send
audioOutputLevel	Integer	audio	receive
bitsReceivedPerSecond	Integer	audio, video	receive
bitsSentPerSecond	Integer	audio, video	send
bytesReceived	Integer	audio, video	receive
bytesSent	Integer	audio, video	send
candidateType	String	bandwidth	
dtlsCipher	String	audio	
googAccelerateRate	Integer	audio	receive
googActiveConnection	Integer	audio	
googActualEncBitrate	Integer	bandwidth for video	
googAdaptationChanges	Integer	video	send
googAvailableReceiveBandwidth	Integer	bandwidth for video	
googAvailableSendBandwidth	Integer	bandwidth for video	
googAvgEncodeMs	Integer	video	send

googBandwidthLimitedResolution	Boolean	video	send
googBucketDelay	Integer	bandwidth for video	
googCaptureStartNtpTimeMs	Integer	audio, video	receive
googChannelId	String	bandwidth	
googCodecName	String	audio, video	receive, send
googComponent	Integer	audio,	send, receive
googCpuLimitedResolution	Boolean	video	send
googCurrentDelayMs	Integer	audio, video	receive
googDecodeMs	Integer	video	receive
googDecodingCNG	Integer	audio	receive
googDecodingCTN	Integer	audio	receive
googDecodingCTSG	Integer	audio	receive
googDecodingNormal	Integer	audio	receive
googDecodingPLC	Integer	audio	receive
googDecodingPLCCNG	Integer	audio	receive
googDerBase64	String		
googEchoCancellation-EchoDelayMedian	Integer	audio	send
googEchoCancellation-EchoDelayStdDev	Integer	audio	send
googEchoCancellation-ReturnLoss	Integer	audio	send
googEchoCancellation-ReturnLossEnhancement	Integer	audio	send
googEchoCancellation-QualityMin	Integer	audio	send
googEncodeUsagePercent	Integer	video	send
googExpandRate	Integer	audio	receive
googFingerprint	String		
googFingerprintAlgorithm	String		
googFirsReceived	Integer	video	send
googFirsSent	Integer	video	receive

googFrameHeightInput	Integer	video	send
googFrameHeightReceived	Integer	video	receive
googFrameHeightSent	Integer	video	send
googFrameRateDecoded	Integer	video	receive
googFrameRateInput	Integer	video	send
googFrameRateOutput	Integer	video	receive
googFrameRateReceived	Integer	video	receive
googFrameRateSent	Integer	video	send
googFrameWidthInput	Integer	video	send
googFrameWidthReceived	Integer	video	receive
googFrameWidthSent	Integer	video	send
googInitiator	Boolean		
googJitterBufferMs	Integer	audio, video	receive
googJitterReceived	Integer	audio	send , receive
googLocalAddress	String	bandwidth	
googLocalCandidateType	String	bandwidth	
googMaxDecodeMs	Integer	video	receive
googMinPlayoutDelayMs	Integer	video	receive
googNacksReceived	Integer	video	send
googNacksSent	Integer	video	receive
googPlisReceived	Integer	video	send
googPlisSent	Integer	video	receive
googPreemptiveExpandRate	Integer	audio	receive
googPreferredJitterBufferMs	Integer	audio	receive
googReadable	Integer	bandwidth	
googRemoteAddress	Integer	bandwidth	
googRemoteCandidateType	String	bandwidth	
googRenderDelayMs	Integer	video	receive
googRetransmitBitrate	Integer	bandwidth for video	

googRtt	Integer	audio, video	send
googSecondaryDecodedRate	Integer	audio	receive
googSpeechExpandRate	Integer	audio	receive
googTargetEncBitrate	Integer	bandwidth for video	
googTargetEncBitrateCorrected	Integer	bandwidth for video	
googTargetDelayMs	Integer	video	receive
googTrackId	String	audio, video	send , receive
googTransmitBitrate	Integer	bandwidth for video	
googTransportType	String	bandwidth	
googTypingNoiseState	Boolean	audio	send
googViewLimitedResolution	Boolean	video	send
googWritable	Boolean	bandwidth	
ipAddress	Integer		
localCandidateId	String	bandwidth	
localCertificateId	String	audio,	send, receive
networkType	String		
packetsDiscardedOnSend	Integer	bandwidth	
packetsLost	Integer	audio, video	send, receive
packetsReceived	Integer	audio, video	receiver
packetsReceivedPerSecond	Integer	audio, video	receiver
packetsSent	Integer	audio, video	send
packetsSentPerSecond	Integer	audio, video	send
priority	Integer		
remoteCandidateId	String	bandwidth	
remoteCertificateId	String	audio	
selectedCandidatePairId	String	audio	
srtpCipher	String	audio	
ssrc	Integer	audio, video	receive, send
transport	String		
transportId	String	audio, video	receive, send

B.2 getstats.io Statistics

B.2.1 Network Statistics

Table B.2: Complete list of network statistics supported by getstats.io.

Parameter	**Value**	**Media**	**Bound**
BytesSent	Integer	Video, audio	Outbond
PacketsSent	Integer	Video, audio	Outbond
RoundTripTime	Integer	Video, audio	Outbond
EncodeCPUUsage	Integer	Video, audio	Outbond
CPULimitedResolution	Boolean	Video	Outbond
BandwidthLimitedResolution	Boolean	Video	Outbond
BytesReceived	Integer	Video, audio	Inbound
PacketsReceived	Integer	Video, audio	Inbound
PacketsLost	Integer	Video, audio	Inbound
Jitter	Integer	Video, audio	Inbound

B.2.2 Participant Statistics

Table B.3: Complete list of participant statistics supported by getstats.io.

Parameter	**Value**
ID	String
UserID	String
UserName	String
Browser	String
BrowserVersion	String
BrowserEngine	String
BrowserEngineVersion	String
OS	String
Platform	String
Mobile	Boolean

B.3 getstats.io Subjective User Feedback Form

Table B.4: Complete list of subjective user feedback supported by getstats.io.

Question	**Alternatives**
How would you rate the overall audio-visual quality of the session (the overall combined audio and vido quality)?	5(Excellent), 4(Good), 3(Fair), 2(Poor), 1(Bad)
How would you rate the video quality of the session?	5(Excellent), 4(Good), 3(Fair), 2(Poor), 1(Bad)
How would you rate the audio quality of the session?	5(Excellent), 4(Good), 3(Fair), 2(Poor), 1(Bad)
Which quality-related issues have you experienced during the session?	Audio problems: bad audio or no audio at all. Video problems: bad video or no video at all. Bad synchronization between audio and video. Not applicable (never experienced any problem. Other, please specify (text box).
Did you considered quitting the session because of quality-related issues?	Yes/No
Did you perceive any reduction in your ability to interact with the other party (parties) during the session?	Yes/No
If yes, specify the problem if you could:	(text box)