Performance Analysis of WebRTC-based Video Conferencing

B.A. Jansen

Master of Science Thesis

Delft University of Technology

Network Architectures and Services

PVM 2016-086

Performance Analysis of WebRTC-based Video Conferencing

Master of Science Thesis

For the degree of Master of Science in Electrical Engineering at Delft University of Technology

B.A. Jansen
1540580

August 8, 2016
M.Sc. Thesis No: PVM 2016-086

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Network Architectures and Services Group

Abstract

Video Conferencing has been rapidly gaining attention and is becoming a more popular way to communicate due to better internet connectivity and increasing computation power. Because of this increase in demand, the dying out of proprietary plugins like Adobe Flash, and the need for easy and accessible video conferencing, a new browser technology emerged: WebRTC. WebRTC allows internet browsers to perform a peer-to-peer video conference without the need to install additional plugins or applications.

WebRTC is widely adopted for video conferencing solutions and getting more support across all browsers. WebRTC is easy to use by application developers, but unfortunately does not allow a developer to easily have insight into the video call characteristics made on the platform. This is because the way WebRTC handles network congestion is completely hidden away in the code of the browser. This work describes the media pipeline of WebRTC including all different network protocols used by WebRTC.

To have a better understanding of how WebRTC handles network congestion, the Google Congestion Control (GCC) algorithm used by WebRTC is analyzed. Since WebRTC works over UDP which does not support congestion control like e.g. TCP does, a custom made congestion control is used. GCC changes the data rate based on packet loss and latency measured at respectively the sender and the receiver side during a WebRTC video conference.

Based on the thorough WebRTC analysis including related work which some of WebRTC's pitfalls, a test bed is set up to conduct a comprehensive performance analysis of WebRTC using the latest browsers, for different scenarios, while being subject to different network effects. The effects of additional latency, packet loss and limited bandwidth are tested. The effects of cross traffic, multi-party, WebRTC on mobile devices, different browsers and different video codecs are furthermore analyzed.

This performance evaluation verifies that results shown in earlier research no longer hold due to the improved GCC algorithm. Results however do show that there is room for improvement since inter-protocol fairness is not ideal and WebRTC streams have a higher priority than competing TCP flows. It also shows the difference in WebRTC's performance for different internet browsers and mobile devices. Also the newly added support for the video codecs VP9 and H.264 are analyzed which do not perform as expected and need to improve.

Table of Contents

Preface

This document contains the thesis written as a final project for the MSc in Electrical Engineering. Most of the work is done at Liberty Global B.V., a telecommunications and television company.

I have always been fascinated by new (web) technologies and I always try to be up to date with the latest technology developments. Before I started working on this thesis, I started experimenting with the new emerging WebRTC browser technique which made it really easy for me to set up a video call between two internet browsers without any prior knowledge of the underlying protocols. I was quickly able to add browser support, build mobile applications and I then decided to create a social video chatting platform available at www.phid.io. This video chatting platform randomly connects you to another random available person and allows you to video chat together.

One thing I could not understand was how this new browser technique adapted for different network effects since I was not able to configure anything from the client side. Some quick research resulted into a custom made congestion control algorithm which is still under heavy development. Some research was done, but either incomplete or outdated. This seemed like a great opportunity to build upon this existing research with a complete and thorough performance analysis of a WebRTC-based video call.

Through my previous job, I have come in contact with Anne Bakker from Liberty Global. An international cable company also interested in the performance of WebRTC. Liberty Global is really committed to innovation to build market-leading next-generation products. The set-top boxes Liberty Global provides for their customers run on the same technology stack WebRTC is built on: JavaScript and HTML. When I told Anne about WebRTC and Phidio, he wondered about the same thing: how does WebRTC perform? This seemed like a good opportunity to explore this.

I would like to thank my supervisor, Dr.Ir. Fernando Kuipers, for allowing me to investigate WebRTC's performance and his support and advice throughout this thesis. Furthermore, I would like to thank Anne Bakker for allowing this opportunity at Liberty Global and the many brainstorm and feedback sessions we have had.

Delft, University of Technology B.A. Jansen
August 8, 2016

Chapter 1

Introduction

This chapter provides a background of video conferencing, why it became so important and how the technology evolved. After this short introduction, the new emerging standard for making video conferencing solutions is discussed: WebRTC and the relevance of this research is described. This is followed by the goals and approach of this thesis and concluded by an outline of what is discussed in the remainder of this thesis.

1-1 Background

Communication has evolved greatly in the last century; from post letters, to communication via the telephone, to Voice Over Internet Protocol (VOIP) and now Video Conferencing (VC). Video Conferencing is a conference between two or more different users using telecommunications to transmit audio and video data. Due to companies expanding internationally there has been a need to bridge communication distances without traveling. Video conferencing has evolved rapidly in the last decades and is getting more accessible to users every day. This is due to the emerging computer and smartphone industry with continuously improving capabilities and also due to better network connectivity of (mobile) devices with a built-in camera.

Video conferencing is widely supported and available through many platforms and applications (e.g. Skype, Google Hangouts, Apple Facetime). Video conferencing either works in a Server-to-Client (S/C) manner, where the video and audio streams get channeled via a server or Peer-to-Peer (P2P), where video and audio directly get exchanged between peers without the extra overhead of a centralized server. A P2P browser protocol that has been getting attention lately and which this thesis will be focusing on, is Web Real-Time Communication (WebRTC). This is an open source network protocol for the browser set up by World Wide Web Consortium (W3C) and the RTCWeb group from the Internet Engineering Task Force (IETF) and allows to transmit video, audio and data between browsers in P2P without the need to install additional software or plugins.

1-2 Research motivations

Before WebRTC was introduced, real-time communications were only available to bigger companies or via extra browser plugins like Adobe Flash. WebRTC makes it really easy for application developers to develop their own video chatting applications because it exposes a very high level Application Programing Interface (API). This makes it really approachable for developers, without the need to understand how the underlying protocols work. The problem with this high level approach is that the way congestion is being handled, is completely hidden from application developers.

In this thesis, WebRTC's congestion algorithm and the performance of WebRTC's video chatting capabilities are analyzed. Earlier research has been done, but some of these findings no longer hold because WebRTC is continuously changing and crucial aspects such as the performance of different browsers, video codecs and mobile devices have not been examined in earlier research. Even though W3C and IETF are working on standardizing the WebRTC protocol, lots of different WebRTC implementations and configurations exist which result in a different video chatting experience. To get a better grasp on how WebRTC performs, different network scenarios are simulated and the performance metrics between these scenarios are compared.

1-3 Goals of the research

The main research question for this thesis is:

"How do the different forms of WebRTC relate to each other and how do they perform under different network conditions?"

This main question is answered through 6 sub questions:

Q1 What is WebRTC, how does it work and how has it evolved?

Q2 How does WebRTC handle congestion?

Q3 How can WebRTC's performance be measured objectively?

Q4 Which different scenarios can be identified for WebRTC?

Q5 How do the different forms of WebRTC compare to each other in performance?

Q6 Which factors play important roles when improving WebRTC's performance?

1-4 Approach

The approach for the problem described in section 1-3 is as follows:

1. First the WebRTC protocol is described, how it can be used to set up a Video Conference and how the underlying protocols work

2. WebRTC is then described in more detail, covering different network topologies and its interoperability with different browsers
3. WebRTC's Google Congestion Control algorithm is analyzed which is implemented to provide fairness and adjust the data rate to handle congestion
4. Related work is studied to get a grasp on what has been researched, focusing on earlier WebRTC video conferencing performance analyses
5. An experimental setup for WebRTC performance testing is created
6. A thorough performance analysis is conducted with varying network conditions, different conference layouts, cross traffic, cross browser and the performance on mobile phones is analyzed
7. Potential ways to improve the performance are suggested and different WebRTC solutions are compared with each other

1-5 Outline

An in depth analysis of WebRTC is given in chapter 2 where the requirements are discussed, how WebRTC's underlying protocols work and an example is given how a WebRTC call can be set up. In chapter 3 different network topologies are described including different topologies that can be used for WebRTC. The WebRTC interoperability and adoption of different browsers is discussed in chapter 4. In chapter 5 WebRTC's congestion control algorithm is described in detail which adjusts the data rate during a WebRTC call when congestion occurs. After WebRTC is thoroughly analyzed, related work is studied in chapter 6 describing their performance analysis findings and how WebRTC's congestion control algorithm has improved over time. The experimental test setup is presented in chapter 7 and a thorough performance analysis of WebRTC is given in chapter 8. This thesis concludes in chapter 9 where all findings are summarized and recommendations are provided.

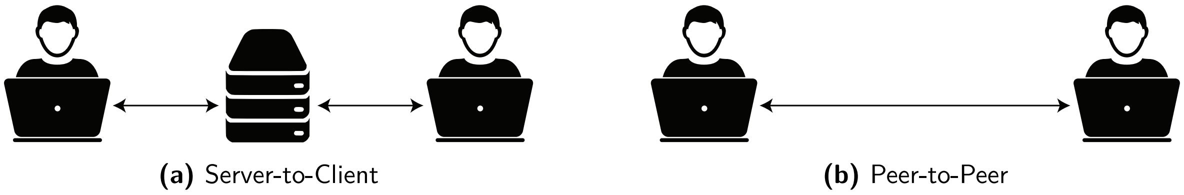
Chapter 2

Web Real-Time-Communications

In this chapter a basic overview of WebRTC is given. In section 2-1, a short introduction is given. Since WebRTC still needs a separate service for signaling, the basic procedure for setting up a call is given in section 2-2. In section 2-3, NAT traversal methods for obtaining a reachable IP address are discussed. The different protocols used by WebRTC are explained in section 2-4. WebRTC's media pipeline is discussed in section 2-5. Finally, in section 2-6 a brief overview of WebRTC's API is given with some basic examples.

2-1 Introduction

WebRTC is a new emerging, free and open source technology which provides Real-Time Communication (RTC) capabilities for the browser. WebRTC allows real time communication for audio (voice calling), video chat and data (file sharing). It allows two browsers to communicate directly with each other in peer-to-peer (figure 2-1b), which is unlike most browser communication which flows through a server (figure 2-1a). One of the biggest benefits of WebRTC is that it is integrated in the browser and runs without the need to install third-party plugins or applications. The World Wide Web Consortium (W3C) [1] set up an API (section 2-6) which is made available in the browser and allows developers to easily implement WebRTC using JavaScript. W3C collaborates with the Internet Engineering Task Force (IETF) [2] which define the WebRTC protocols and underlying formats as discussed in section 2-4. The goal of this collaboration to create a unified standard for all internet browsers.



(a) Server-to-Client **(b)** Peer-to-Peer

Figure 2-1: Difference between Server to Client and Peer to Peer communication

Before WebRTC was introduced, video chatting in the browser relied on proprietary plugins such as Adobe Flash. One of the downsides of Flash is that developers are free to define their own solutions and protocols instead of well-known standards [3]. Plus that Adobe Flash needs to be installed separately with the browser and is significantly losing adoption due to extreme battery usage and security vulnerabilities [4].

WebRTC was first introduced and made open source by Google in May 2011 [5], but not available for the masses till late 2012 in the most recent version of Google Chrome. A lot has changed since then, and the definition of the final WebRTC standard is still an ongoing process where IETF and W3C work hard on standardizing the technology [6]. Currently, not all browsers support WebRTC yet without installing external plugins as discussed in chapter 4. Also the custom-made congestion control algorithm to adapt when network congestion occurs is under heavy development (chapter 5).

In this thesis, the focus lies on the peer-to-peer WebRTC video chatting capabilities and performance under different network conditions. JavaScript is used for the communication between the peers and it routes the media to the respective HTML video elements as further explained in section 2-6. As mentioned before, WebRTC can also be used to exchange data between two peers through the DataChannel component, but this functionality is not studied in the remainder of this thesis.

2-2 Signaling server

A signaling server coordinates all communication and exchanges the necessary information to set up a connection between peers. A signaling service is not defined in WebRTC. This is because WebRTC is intended to solely specify and control the media plane and different applications may prefer different communication infrastructures.

The signaling server has to be centralized and thus not peer to peer (unlike WebRTC). It keeps track of the online users and when a peer wants to initiate a call, the required information is exchanged to set up a peer connection between both peers. Furthermore, the shared signaling server is used to exchange other control data such as: Error messages, control messages (initiate call, disconnect, receive call), network data (IP/port), media metadata (codec settings, etc).

WebRTC uses the *Session Description Protocol (SDP)* [7] to describe the parameters of the peer to peer connection. SDP does not deliver any media itself, but describes session parameters such the types of media offered (video and/or audio) and its resolutions, the media codecs used (VP8/Opus), the decoding capabilities, network connection information and more. All peers create and exchange SDPs before setting up a direct peer connection and the SDPs describe the session. Fortunately, these different parameters do not need to be interpreted and managed manually, since the *JavaScript Session Establishment Protocol (JSEP)* [8] abstracts these away in the RTCPeerConnection object.

Figure 2-2 shows how SDPs get exchanged via the signaling server between two peers, initiated by Alex:

1. The initiator does a *getUserMedia()* request to get access to its audio/video *MediaStream* and adds it to a new *RTCPeerConnection*

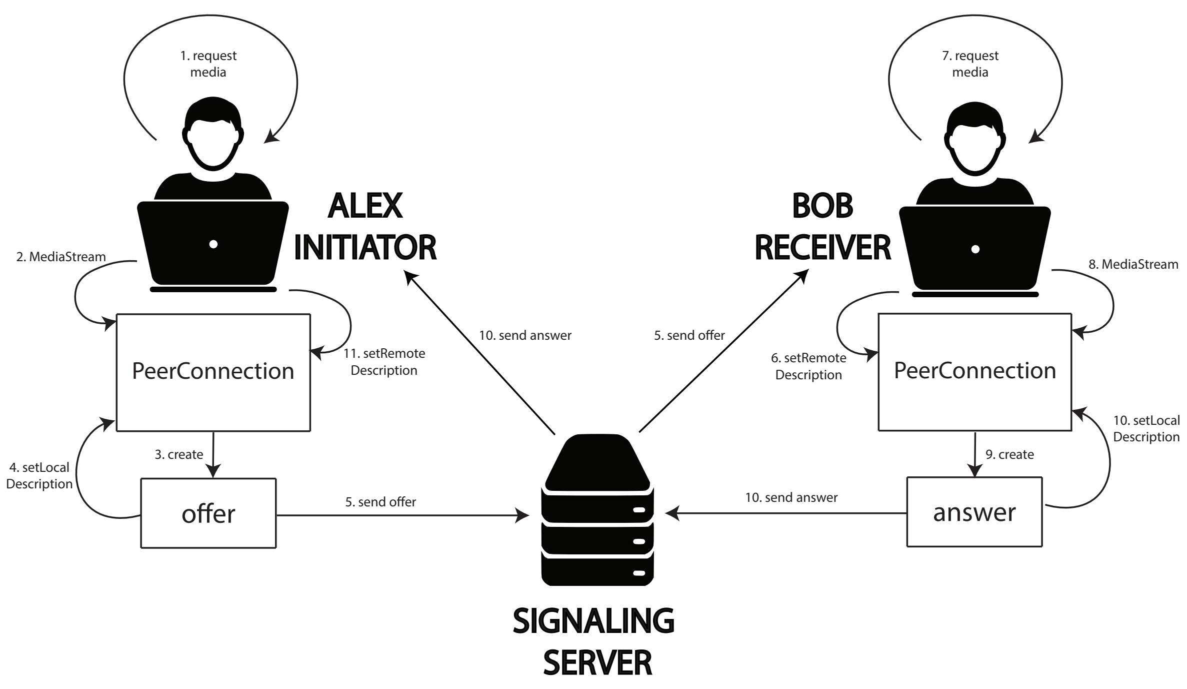


Figure 2-2: SDP exchange between initiator and receiver to prepare for peer-to-peer connection

2. An (SDP) *offer* is created using the generated *RTCPeerConnection*
3. This *offer* is set as the peer connection's *localDescription* and sent to the receiver via the *signaling server*
4. Upon receiving the offer, the receiver creates a *RTCPeerConnection* using its own *MediaStream* and adds the received offer as its *remoteDescription*
5. The receiver creates an (SDP) *answer* from its *RTCPeerConnection*, adds this as its *localDescription* and sends this answer to the initiator via the *signaling server*
6. The initiator receives the answer and sets it as its *remoteDescription*

Both peers have now exchanged each other's SDPs and negotiated the settings and type of streams to use, and a direct connection between both peers can be set up.

2-3 NAT traversal

Even though the peers have exchanged information using the signaling server, a direct route between the peers has not been set up yet. However, because peers tend to be behind one or more firewalls, NATs or antivirus protection, a direct connection is usually not possible. Network Address Translation (NAT) translates IP addresses from one address space to another. This for instance happens when computers are connected to the internet via a router. Every single computer connected to this router has the same external IP address, but are distinguished with unique local IP addresses. When the router receives a packet, it maps this incoming packet to its associated unique local IP address using a NAT table. This section will discuss how WebRTC implements NAT traversal.

WebRTC uses *Interactive Connectivity Establishment (ICE)* [9, 10] to traverse NATs and firewalls. WebRTC's built-in ICE agent tries to discover the best path to connect peers. First ICE tries to connect directly to the IP address exchanged via the signaling server. This local IP address however only works when both peers are in the same local network or not behind a NAT. When this is not the case, ICE uses a *Session Traversal Utilities for NAT (STUN)* server [11] to obtain an external IP address to set up a direct connection between both peers. Sometimes a direct route can then still not be set up and then the communication is relayed via a *Traversal Using Relays around NAT (TURN)* server [12], according to [13] this occurs in 8% of the cases, usually due to heavily protected enterprise networks. When STUN is used, video and audio are directly transmitted between the end points, whereas media is proxied via a server (as in figure 2-1a) when a TURN relay is necessary. STUN is always attempted first and when a direct connection between the peers cannot be set up, TURN is used as a last resort.

As discussed before, a WebRTC connection is initiated via a *RTCPeerConnection* object. This RTCPeerConnection contains an ICE agent which accepts an *iceServers* object in which multiple STUN and TURN servers can optionally be added as follows:

```
var configuration = {
  'iceServers': [
    {
      'url': 'stun:stun.l.google.com:19302'
    },
    {
      'url': 'turn:192.158.29.39:3478',
      'credential': '<somepassword>',
      'username': 'user'
    },
    {
      'url': 'turn:132.12.452.39:3478',
      'credential': '<somepassword>',
      'username': 'user'
    }
  ]
};

var pc = new RTCPeerConnection(configuration);
```

Listing 2.1: Example of initializing a new PeerConnection with multiple iceServers

From this generated RTCPeerConnection the respective SDPs are generated as described in section 2-2. Once a session description (local or remote) is set for a client (figure 2-2 step 4 and 6), the ICE agent gathers all possible route candidates for the local peer:

- The ICE agent obtains the local IP address
- Configured STUN server(s) are queried which return an external IP address and port.
- Possible TURN server(s) are used if a direct connection between peers cannot be set up

Once these candidates are obtained and exchanged via the signaling server, the ICE agent does connectivity checks to see if the other peer can be reached. If STUN and TURN servers are present both candidates have a list of possible ICE candidates and the ICE agent prioritizes these. First local IP addresses are tested, then external IPs from the STUN server and lastly TURN servers if everything else fails. Once a connection between the peers is established, ICE continues to do keep-alive requests and keeps looking for alternate (faster) routes to connect peers. When for instance two TURN servers are available (as in listing 2.1) and there is no direct communication possible, the best possible path is chosen. Thus when two peers in the Netherlands try to connect and there's a TURN server available in Germany and another one in San Francisco, the ICE agent makes sure that the TURN server in Germany is used to minimize the communication delay.

2-4 WebRTC's protocol stack

In this section WebRTC's internals and underlying protocols are discussed. An overview of WebRTC's complete protocol stack is shown in figure 2-3, where different protocols are used for WebRTC's media and data-transfer capabilities. In the following sections these different layers are discussed in more detail. Section 2-4-1 discusses why UDP is chosen over TCP and section 2-4-2 explains the upper layers used by WebRTC.

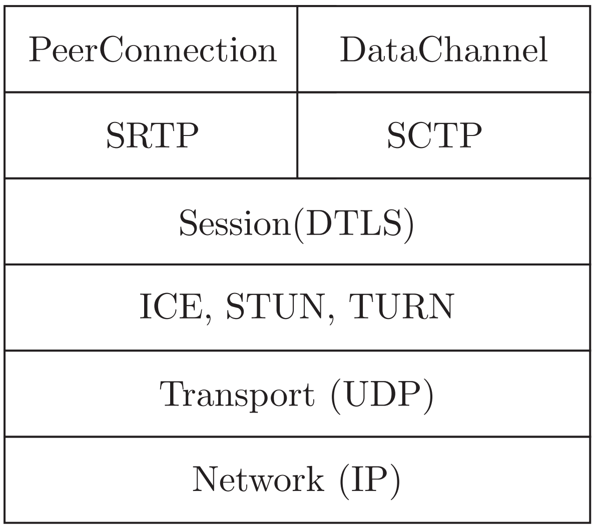


Figure 2-3: WebRTC protocol stack for media (left top) and data (right top)

2-4-1 Transport protocol

Transport of information between the browser and server (e.g. HTTP/HTTPS) is done via the Transmission Control Protocol (TCP). TCP is known for its reliability and error-correction which guarantees packets get delivered in the same order in which they are sent with retransmissions when packet loss occurs.

With real-time communications (i.e. WebRTC), speed is preferred over reliability. WebRTC (and many other multimedia applications) transmit audio, video and data between browsers over the User Datagram Protocol (UDP). WebRTC over TCP is also possible and used as a last resort, when all UDP ports are blocked which can be the case in heavily protected enterprise networks. With UDP, packets are sent to the recipient without knowing whether they arrive. Because there are no retransmissions, congestion control or acknowledgements, UDP is noticeably faster than TCP. UDP is chosen for WebRTC because low latency and high throughput are more important than reliability. Because UDP is really minimalistic, additional transport protocols are added to ensure support for changing network conditions, synchronization and more. More on this in the next section.

2-4-2 WebRTC protocols

Since WebRTC uses UDP, extra security protocols are added for encryption, reliability, flow control and congestion control. The WebRTC specification requires that all transferred data is encrypted. The Datagram Transport Layer Security (DTLS) [14] (figure 2-3) is responsible for adding security to the other-wise unsecure datagram protocol UDP. DTLS is based on Transport Layer Security (TLS) which is used to encrypt TCP traffic. DTLS prevents message forgery, eavesdropping and data tampering to UDP. DTLS is used for establishing the keys, which can then be used by SRTP to encrypt the mediastream.

Secure Real-Time Transport Protocol (SRTP) [15] is the most important protocol used by WebRTC for its media delivery (left top of figure 2-3). Once a PeerConnection is established (after offer/answer), the SRTP protocol is used to transfer the audio and video streams between the WebRTC clients. SRTP contains all information required for transporting and rendering the media. It furthermore adds sequence numbers, timestamps and unique stream IDs which can be used by the receiver for detecting e.g. out-of-order delivery and synchronization between audio and video streams. SRTP however does not provide mechanisms such as error recovery, guaranteed delivery and reliability. This is handled by the Secure Real-time Control Transport Protocol (SRTCP) [15] which flows in the opposite direction to provide feedback. It tracks packet loss, last received sequence number, jitter for each SRTP packet, etc. Data gathered by SRTCP is transferred between the peers to adjust for quality and other parameters when necessary. The browser takes care of interpreting and adjusting of these protocols using a congestion algorithm as further explained in chapter 5.

Lastly, the Stream Control Transmission Protocol (SCTP) [16] is used on top of UDP and the secure DTLS layer for the Data Channel (application data) as seen at the right top of figure 2-3. SCTP adds TCP-like features to UDP with features like multiplexing, flow control and congestion control [17]. It provides the best features of TCP and UDP with configurable reliability (i.e. acknowledgements) and delivery (i.e. unordered or ordered delibery).

2-5 Media pipeline

Now that WebRTC's connection establishment procedure and protocol stack have been explained, this section will focus on how raw media is processed through to the receiver. As

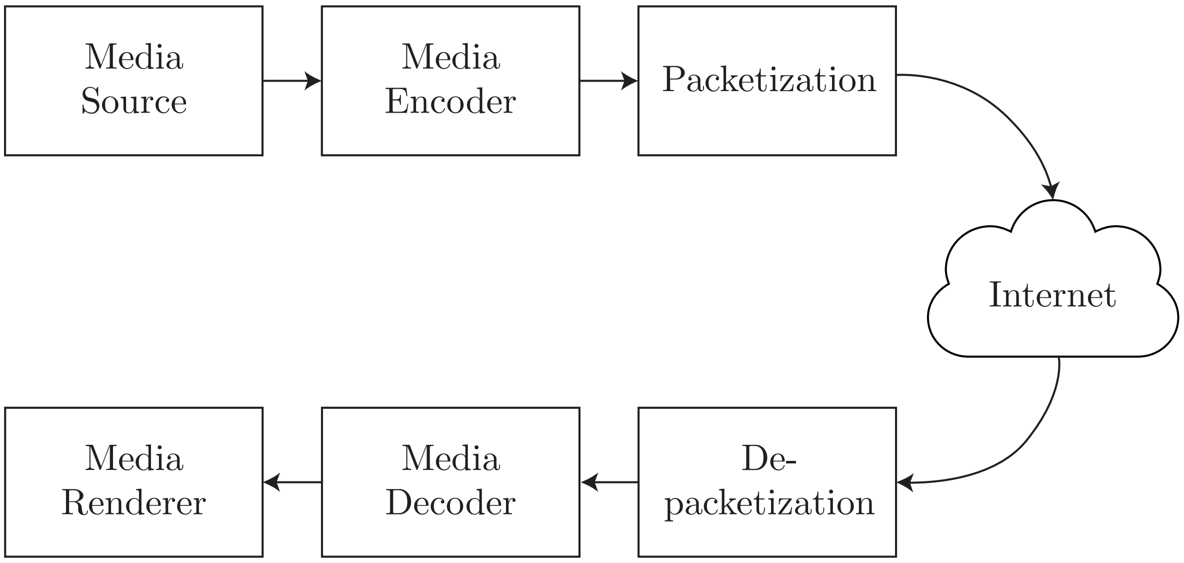


Figure 2-4: WebRTC's media processing pipeline

mentioned before, WebRTC handles all media processing as illustrated in figure 2-4. Raw audio and video data from respectively the microphone and webcam are first preprocessed and then encoded at a given target rate determined by the congestion control algorithm in chapter 5. The encoding process is further explained in section 4-2. These frames are then packetized into Maximum Transmission Unit (MTU) size and sent to the receiver over RTP/UDP as explained in section 2-4-1. Once these frames arrive at the receiver, they are subsequently depacketized and decoded which returns the raw video input which can be rendered at the receiver.

2-6 Application Programming Interface

In this section the core components that WebRTC exposes through JavaScript as defined by W3C [1] are briefly discussed. In section 2-6-1 the procedure to get access to a user's microphone and webcam are explained. The basic required components of the peer connection are described in section 2-6-2. Lastly, a brief overview WebRTC's data channel is given in section 2-6-3.

2-6-1 getUserMedia

To get access to a user's audio and/or video input device, `MediaDevices.getUserMedia()` [18] is used. Once this function is invoked, the user is prompted for permission to its media devices. If permission is provided, a `MediaStream` is returned which consists of a VideoTrack and AudioTrack (both left and right channel). This MediaStream can then be displayed in a HTML5 video tag both locally and remotely. This process is shown in the listing below:

```
// get audio and video
var getMedia = navigator.mediaDevices.getUserMedia({ audio: true, video: true
    });

// once permission is granted
getMedia.then(function(mediaStream) {
  // display local video in HTML video element
  var video = document.querySelector('video');
  video.src = window.URL.createObjectURL(mediaStream);
});

// if user blocks permission
p.catch(function(err) {
    // alert user
    alert('Permission not granted');
});
```

Listing 2.2: Example of getting media access and displaying it in a video element

This function is available in most modern browsers (except for Safari and Internet Explorer) and requires a microphone and webcam to be properly set up. As of Google Chrome v47 (december 2015), getUserMedia is only available from secure contexts (i.e. websites using HTTPS) to force encrypted usage.

2-6-2 PeerConnection

The `RTCPeerConnection` [19] is responsible for managing the peer to peer connection between the different browsers. As seen in listing 2.1 it is initialized with the available iceServers. Furthermore, it creates offers/answers as shown in figure 2-2 and it has an ICE agent which finds all possible paths (section 2-3). A basic peer connection (with a trivial signaling server) can be set up as follows:

```
var pc = new RTCPeerConnection(ice);

var getMedia = navigator.mediaDevices.getUserMedia({ audio: true, video: true
    });

getMedia.then(function(mediaStream) {
  pc.addStream(mediaStream);

  // set local video
  var local_vid = document.getElementById('local_vid');
  local_vid.src = window.URL.createObjectURL(mediaStream);

  pc.createOffer(function(offer) {
    pc.setLocalDescription(offer);
    signalingChannel.send(offer.sdp);
  });
}

pc.onicecandidate = function(evt) {
  if (evt.candidate) {
    signalingChannel.send(evt.candidate);
  }
}
```

```
signalingChannel.onmessage = function(msg) {
  if (msg.candidate) {
    pc.addIceCandidate(msg.candidate);
  }
}

pc.onaddstream = function (mediaStream) {
  // set remote video
  var remote_vid = document.getElementById('remote_vid');
  remote_vid.src = window.URL.createObjectURL(mediaStream);
}
```

Listing 2.3: Example of setting up a basic Peer Connection in Javascript

After exchanging the SDPs and a successfully established connection, RTCPeerConnection receives the recipient's stream via RTCPeerConnection's `onaddstream` and the remote stream can then be shown in the HTML video element.

2-6-3 Data Channel

Even though this thesis focuses primarily on audio and video transport, WebRTC also supports a data channel to transfer data (e.g. files or text messages) over the same RTCPeerConnection. As also shown in figure 2-3 it is not built on top of SRTP, but uses SCTP. When creating a data channel on a RTCPeerConnection, the developer can configure a lot of the options. Developers can define the retransmissions behavior, reliability and delivery (if packets have to come in-order or out-of-order) when setting up a data channel. When unordered, unreliable delivery with zero retransmissions are set, the channel characteristics are similar to UDP.

Chapter 3

Network topologies

A network topology describes how nodes are arranged in the network. WebRTC has a peer-to-peer nature where media flows directly between the participants, but sometimes the media gets relayed via an in-between server. This server can either function as a TURN server (section 2-3) because the participants cannot reach each other directly, or an extra server can be used to reduce bandwidth for multi-party conferencing. There can be a significant difference in performance when having a conversation between two clients (one to one) versus a multi-party conference. In this chapter different network topologies for one to one and multi-party solutions are first discussed in respectively section 3-1 and 3-2. Followed by section 3-3 where an overview of different WebRTC topology implementations is discussed.

3-1 One to one conversations

This is the easiest and most common way of video conferencing where two clients are having a one-to-one video conference. It can be conducted in two different manners: Peer-to-Peer (P2P) or Server-to-Client (S/C) both shown in figure 3-1.

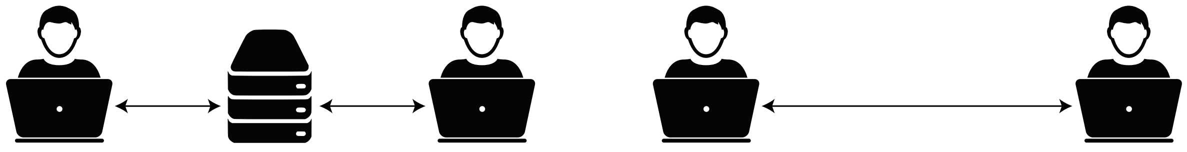

(a) Server-to-Client **(b)** Peer-to-Peer

Figure 3-1: Difference between Server to Client and Peer to Peer communication

For one to one conversations, a P2P topology (figure 3-1b) is preferred and will have the best performance since there is a direct connection (thus without a server in-between) between the two peers having the conversation.

Even though Peer-to-Peer conversations will always have a preference, Server-to-Client solutions (figure 3-1a) also exist for one-on-one conversations. With this network architecture, the server receives and transmits all video and audio data between the two clients and therefore adds extra latency varying based on where the server is located. Two scenarios are available when an extra server is necessary. Firstly, when a direct connection between two pairs cannot be set up and the data has to be relayed via a TURN server as explained in section 2-3. Another less common scenario is when two peers cannot negotiate a mutual (video) codec, the media streams can then be transcoded by an in-between (media) server which converts the media to respective formats which can be decoded at each peer.

3-2 Multi-party conferencing

When more than two nodes participate in a video call, we speak of multi-party video conferencing. Similarly as one to one conversations, a distinguish is made between peer-to-peer topologies (section 3-2-1) and topologies where an extra server is introduced (section 3-2-2). A dedicated server can improve the video call experience in certain scenarios when a particular party forms a bottleneck.

3-2-1 Peer-to-Peer

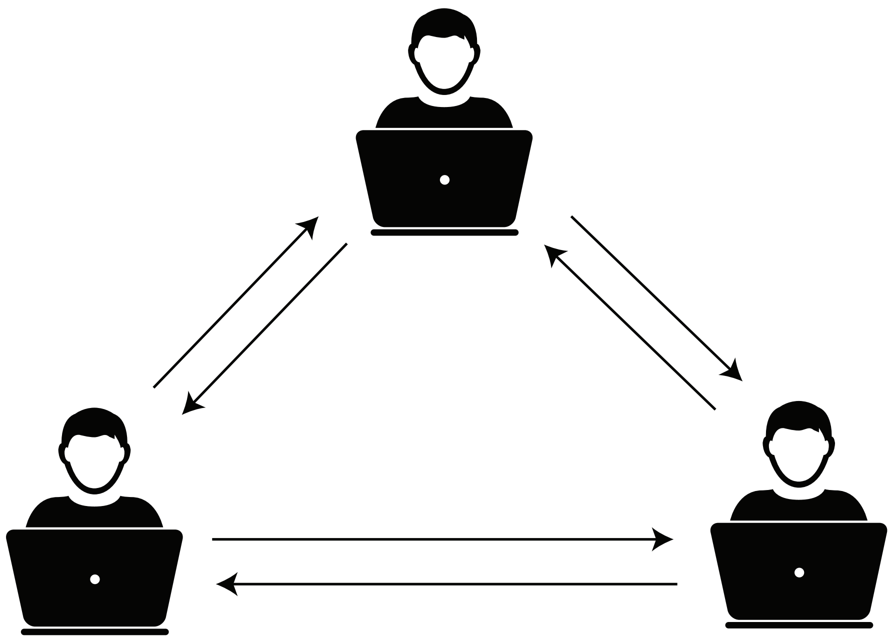


Figure 3-2: Meshed topology

Similar to one-on-one conversations, peer-to-peer conferencing transmits and receives the audio and video streams of the other peers participating in the video conference, thus doubling and tripling the necessary upload and download bandwidth for respectively 3 and 4 person parties compared to one-on-one conversations.

Besides the traditional meshed network where all nodes are connected to each other, there are also alternative topologies available. In a ring topology (figure 3-3a) all nodes are connected

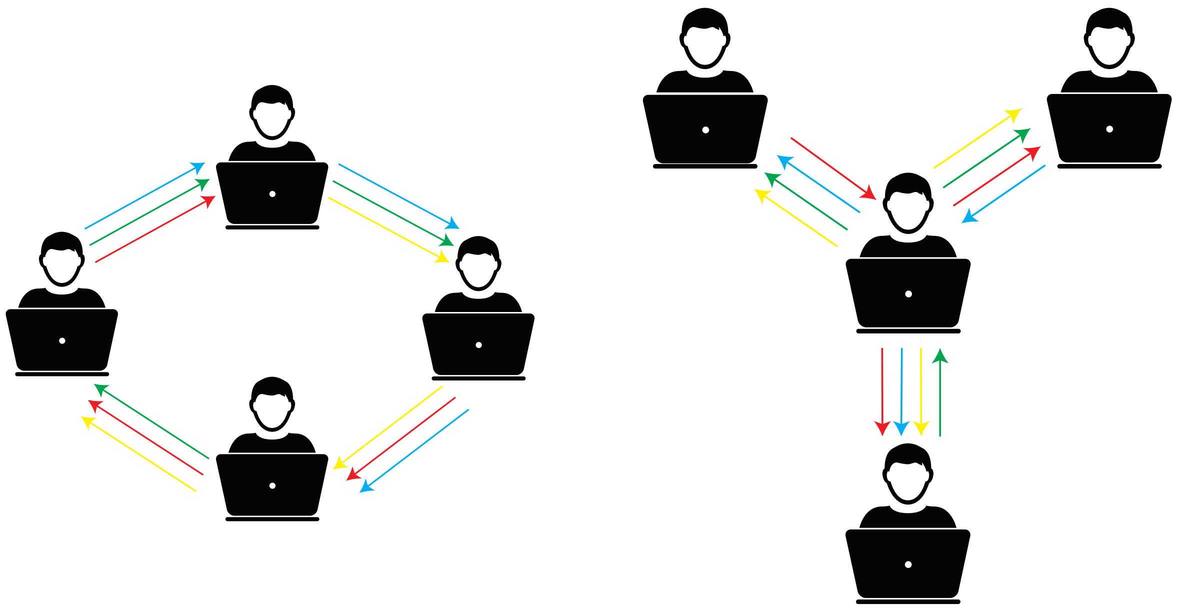

(a) Ring Topology **(b)** Star Topology

Figure 3-3: Meshed network topology alternatives

in a closed loop and every node is connected to exactly two nodes. This topology requires the same bandwidth as the meshed network, has significantly higher communication delay and is very prone to errors occurring at a single node and therefore not suitable for WebRTC. A star topology (figure 3-3b) can be interesting for WebRTC where a supernode with the highest available bandwidth receives all data and transmits this data to all other peers. The required uplink bandwidth for non-supernodes can be reduced significantly from $n-1$ to 1 while the supernode requires $(n-1)^2$ times the uplink bandwidth instead of $n-1$ where n is the amount of nodes participating in the call.

3-2-2 Server-to-Client

Having a dedicated server with large parties can be beneficial due to a significant decrease in the upload bandwidth for every node. Two alternatives are proposed in this section; a server acting simply as a forwarder and a server with advanced capabilities.

Selective Forwarding Unit

By introducing an extra server which forwards the streams, the necessary uplink bandwidth can be reduced. If a Selective Forwarding Unit (SFU) is used, all participants only have to upload their stream once and the SFU forwards these to the other participating clients. This approach introduces extra latency because streams are relayed, but significantly reduces both CPU usage (for encoding all streams $n-1$ times) and necessary uplink bandwidth (from $n-1$ times to 1 time). In figure 3-4 an example is shown, where the colored streams are forwarded (without being altered) by the SFU. One downside of this approach is that the stream is sent in the same quality to all receivers. This means that if user #1 and user #3 have a perfect

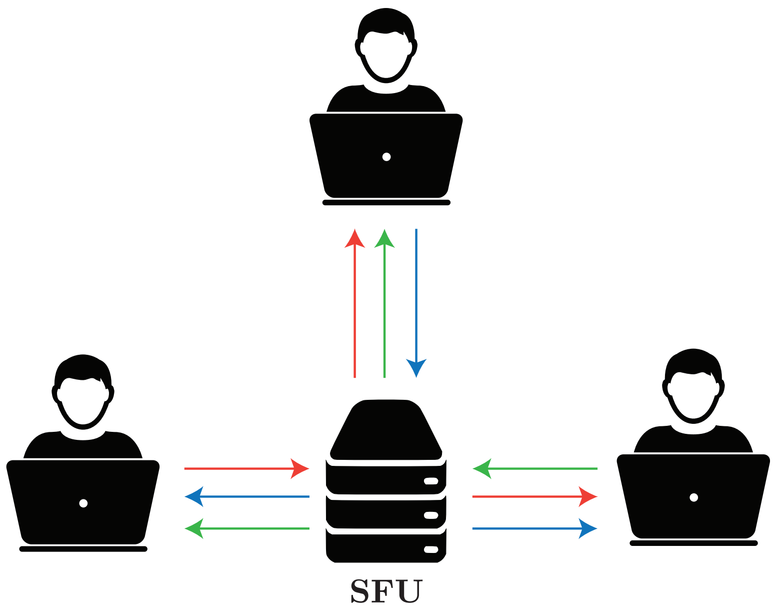


Figure 3-4: Selective Forwarding Unit

connection, but there is a bottleneck between user #1 and user #2. That user #1 will scale down its video quality for not only user #2 but also user #3 since the same video is sent to all users.

Multipoint Control Unit

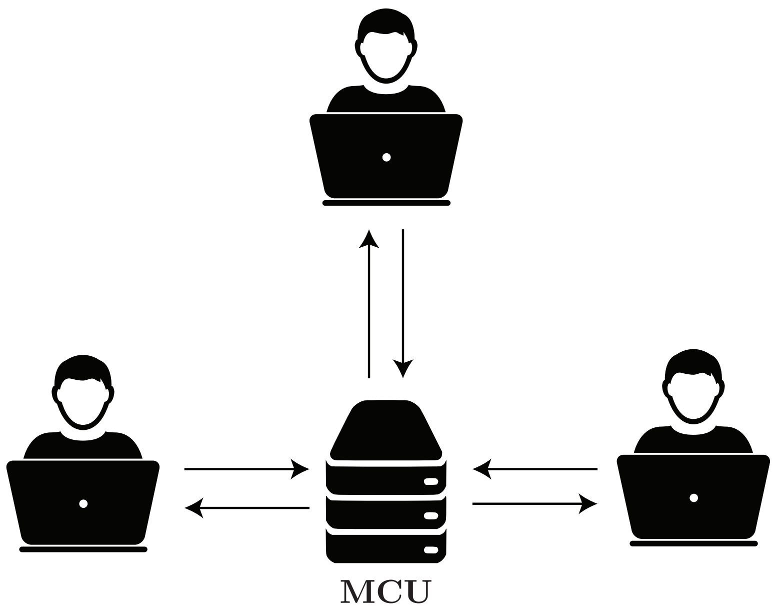


Figure 3-5: Multipoint Control Unit

The SFU only forwards the packets between the peers, whereas a Multipoint Control Unit

(MCU) also processes all incoming streams and multiplexes them before sending them to the other peers. Besides multiplexing, MCUs are also known for their more advanced media processing (e.g. transcoding, recording, etc.). This media processing does not only delay the real-time media, but also requires computation power whereas an SFU primarily requires sufficient bandwidth since packets are not altered. A comparison of SFU and MCU processing is shown below in figure 3-6.

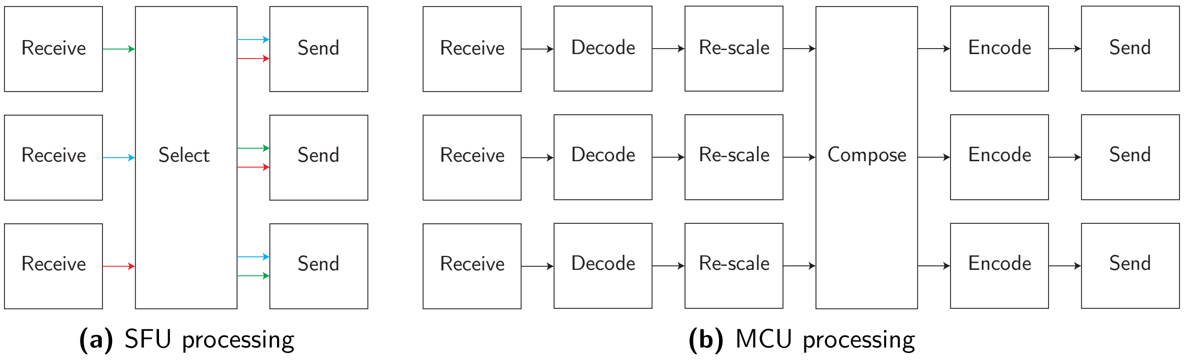


Figure 3-6: Difference between SFU and MCU processing [20]

Compared to SFUs, MCUs do have the ability to send the received video stream in different qualities and can even transcode a stream from e.g. VP8 to H264 when two peers do not support the same video codecs.

3-3 WebRTC topology solutions

In this section all available multi-party solutions for multi-party WebRTC conferences are discussed. In section 3-3-1 a decision is made on how multi-party peer-to-peer WebRTC conferences are analyzed. In section 3-3-2 an informed decision is made on which WebRTC gateway is used to analyze centralized multi-party WebRTC video chats.

3-3-1 Peer-to-peer

Internet browsers currently do not support the ability to create multi-party conference calls out of the box. Fortunately, an extra JavaScript plugin, *RTCMultiPeerConnection* [21], is made available to allow peer-to-peer full meshed video conferencing (figure 3-2) in the browser with WebRTC. It basically initiates multiple RTCPeerConnections with all other clients in the mesh. Unfortunately, it does require the outgoing video stream to be encoded (and of course decoded) for every single client which requires a lot of computation power, but this does mean that different clients can be served with different quality streams depending on their mutual connectivity.

Currently there are no WebRTC solutions available for the ring or star topology (figure 3-3). The ring topology would however not be suited for WebRTC because of the extra delay it introduces plus that bad connectivity for a single client affects all communication in the ring. If a supernode is detected with sufficient uplink bandwidth $(n-1)^2$, all traffic could theoretically be channeled through this node. Due to the complex signaling challenge in identifying this node and forwarding the streams, this option is not studied in the performance analysis. RTCMultiPeerConnection is therefore used to test peer-to-peer multi-party WebRTC video conferencing.

3-3-2 WebRTC gateway

A WebRTC gateway is used to compare with WebRTC meshed calls. As explained in section 3-2-2, two variants are available: a SFU which simply forwards the streams to all other clients and a MCU which additionally processes and multiplexes these streams before they are sent as a single stream to the receiver. Three different solutions are discussed in this section:

- *Jitsi Videobridge*, recently acquired by Atlassian, offers a SFU solution which relays the streams to all call participants.
- *Janus WebRTC Gateway*, also offers this same SFU solution and has additional support for extra custom plugins which allow e.g. MCU-like functionalities
- *Kurento*, is a media server offering MCU-like capabilities such as transcoding, multiplexing and recoding. Unfortunately, Kurento is not commercially used due to its extreme licensing which prevents this

For the multi-party performance analysis, the Janus gateway is used. This is because Janus is more widely used than Kurento due to its better licensing which allow commercial companies to freely use it. Janus is chosen instead of Jitsi because the Janus gateway outputs the necessary call characteristics for the performance analysis, while the Jitsi Videobridge does not provide this functionality by default. To avoid additional complexity and for better reproducibility, the tests in the performance analysis are limited to the Janus SFU and MCU WebRTC solutions are not analyzed.

Chapter 4

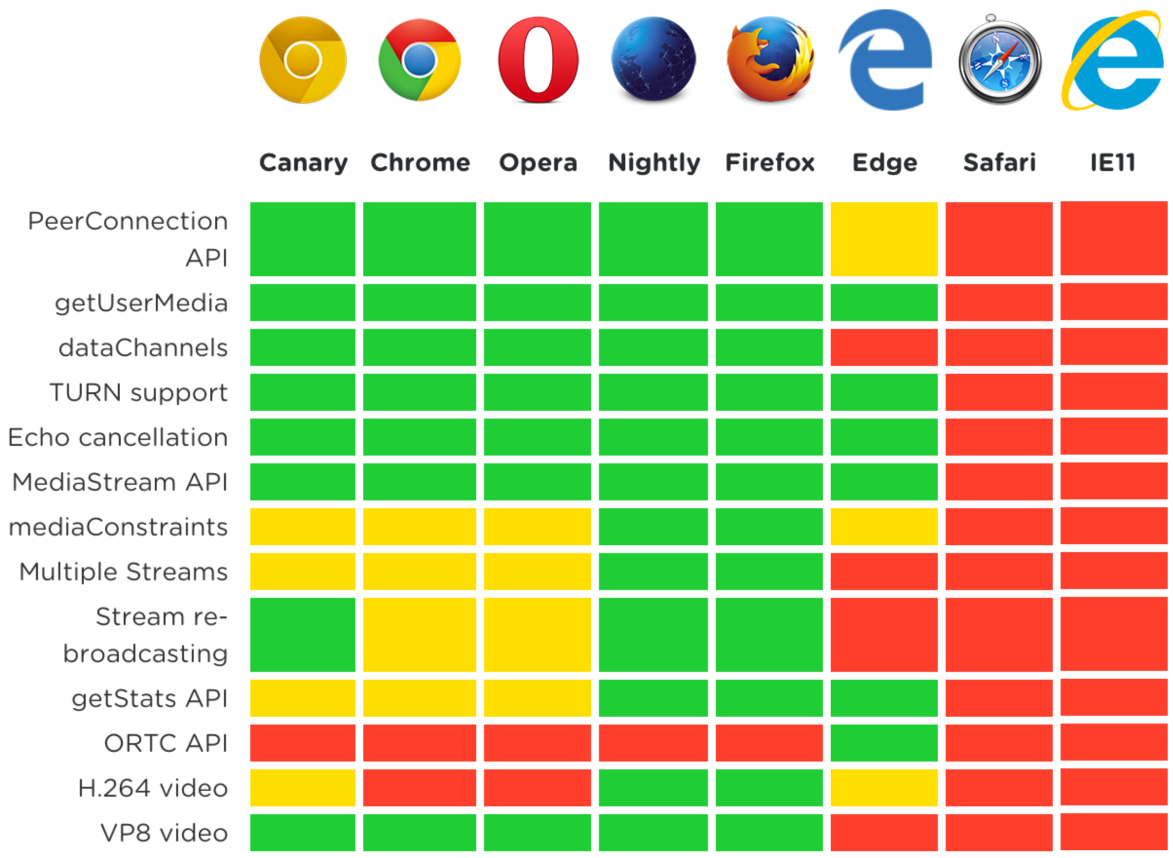
Interoperability

The World Wide Web Consortium (W3C) and Internet Engineering Task Force (IETF) are working hard on creating a standard for WebRTC which will allow different browser vendors to communicate with each other. At the time of writing not all browsers support WebRTC, different browsers support different codecs and the support on mobile devices also varies. In this chapter different implementations and interoperability of WebRTC is discussed. This chapter starts with looking into WebRTC's current coverage in section 4-1. Then different audio and video codecs are discussed and the different ways they work in section 4-2. Followed by different browser implementations in section 4-3 and lastly, mobile support is discussed in section 4-4.

4-1 Adoption

Although WebRTC is a relatively new technology, most of the major internet browsers have (partial) support for the protocol as shown in figure 4-1. Green indicates full support, yellow partial support and no support is indicated with red. This screenshot is adjusted and just shows the crucial components for setting up a WebRTC call and some of the features talked about. Figure 4-1 shows the different video codecs that browsers support and that both Internet Explorer <= 11 and Safari do not support any of WebRTC's features. This will be discussed in detail in the remainder of this chapter.

Browser support is one thing to look at, it is however more interesting to also take the usage of these browsers into consideration. This thesis therefore looks at browser statistics gathered by *Net Market Share* [23]. The browser distribution for every year is first gathered which gives a distribution of all browsers and its versions that were being used that year. Based on when WebRTC was introduced for every browser, the portion of WebRTC capable sessions for every year can be determined. This is shown for both desktop and mobile/tablet in table 4-1 which shows that browser support for WebRTC is rapidly increasing. This table illustrates the capabilities of setting up a WebRTC call but unfortunately does not take into account whether these users also have the necessary audio and video equipment.



	Canary	Chrome	Opera	Nightly	Firefox	Edge	Safari	IE11
PeerConnection API	🟩	🟩	🟩	🟩	🟩	🟨	🟥	🟥
getUserMedia	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
dataChannels	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
TURN support	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Echo cancellation	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
MediaStream API	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
mediaConstraints	🟨	🟨	🟨	🟩	🟩	🟨	🟥	🟥
Multiple Streams	🟨	🟨	🟨	🟩	🟩	🟥	🟥	🟥
Stream re-broadcasting	🟩	🟨	🟨	🟩	🟩	🟥	🟥	🟥
getStats API	🟨	🟨	🟨	🟩	🟩	🟩	🟥	🟥
ORTC API	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥
H.264 video	🟨	🟥	🟥	🟩	🟩	🟨	🟥	🟥
VP8 video	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥

Figure 4-1: Modified screenshot of [22], showing which WebRTC features are supported on different browsers

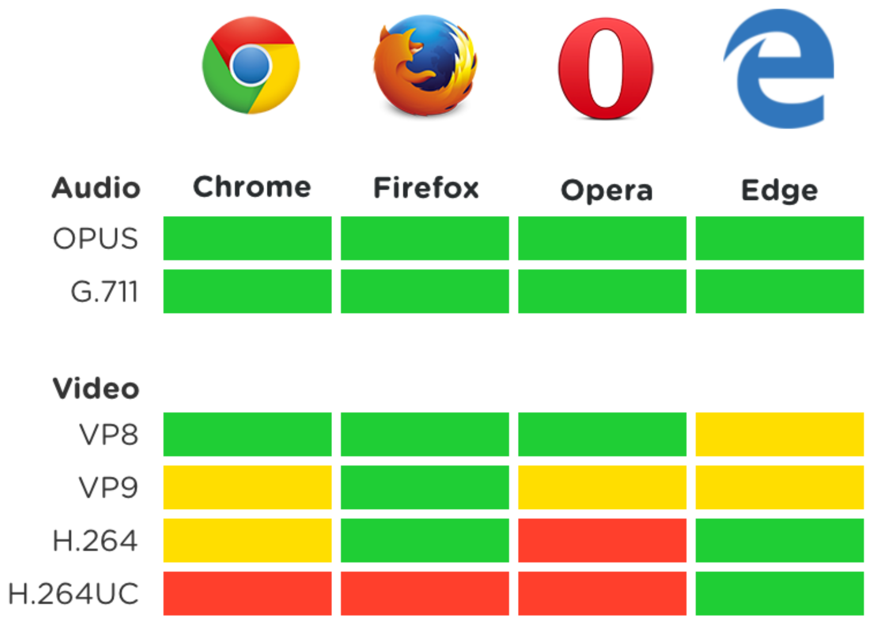
	2012	2013	2014	2015	2016
Desktop	2,01%	22,06%	30,90%	36,53%	68,51%
Mobile and Tablet	0%	1,75%	12,32%	28,94%	41,59%

Table 4-1: Support for WebRTC in the average browser session from 2011 till (April) 2016

4-2 Media codecs

WebRTC transmits and receives both audio and video streams as explained in chapter 2. A video codec takes the raw video data produced by the webcam with any kind of framerate, resolution, color depth, etc. and compresses it to reduce its size. The encoder is responsible for compressing this raw data and the decoder on the receiving end then decompresses it. Some quality is lost during this process but it does not weigh up against the reduced bandwidth that is required.

Besides WebRTC's browser compatibility as explained in section 4-1, the video codec support across different browsers is a more serious issue. When two users set up a call, they exchange a Session Description Protocol as explained in section 2-2. In this SDP every user specifies which audio and video codecs they support and a new video call can only be established if both users share at least one supported audio and video codec.



Audio	Chrome	Firefox	Opera	Edge
OPUS	🟩	🟩	🟩	🟩
G.711	🟩	🟩	🟩	🟩
Video				
VP8	🟩	🟩	🟩	🟨
VP9	🟨	🟩	🟨	🟨
H.264	🟨	🟩	🟥	🟩
H.264UC	🟥	🟥	🟥	🟩

Figure 4-2: Modified screenshot of [22], showing which WebRTC features are supported on different browsers.

There has been a lot of debate on which codecs need to be supported. Both VP8 and H.264 have been potential candidates. VP8 has the benefit that developers do not have to pay license fees, while H.264 is preferred because a majority of the mobile devices support hardware accelerated encoding and decoding for H.264. The IETF RTCWEB group recently decided that both VP8 and H.264 support is required for all WebRTC capable browsers [24]. Chrome, Firefox and Opera currently support VP8 and are working on H.264 support. Microsoft Edge currently does not support either VP8 or H.264 which makes video conferencing between Edge and other browsers currently impossible unless in-between media servers (MCU) transcode the streams. Microsoft Edge supports H.264UC which is used by Skype to enable forward error correction, simulcast and scalable video coding. Microsoft Edge is not planning on supporting VP8 soon, but is planning to release a new version of Edge in 2016 with H.264 support [25] to enable cross browser video chatting.

Furthermore, VP8's successor VP9 has been getting a lot of attention. VP9 is already supported by Chrome, Firefox and Opera even though VP8 is currently preferred over VP9 when negotiating which codec will be used. VP9 requires more computational power but is able to stronger compress videos and therefore requires a lower bandwidth. VP9 is therefore great for compressing high quality (e.g. 4K) video streams and significantly reduces the bandwidth costs for both TURN servers and SFUs. On the other hand, the extra computational power will reduce the battery life for mobile phones. VP9 support is also coming to Microsoft Edge in the future [26], most likely after H.264 is introduced.

When it comes to audio codecs, there is less debate and the OPUS audio codec is preferred and supported by all browsers. It can be used for low and high bitrates and when high quality audio is required in the WebRTC application, OPUS is the only choice. G.711 is also required by all browsers according to the IETF standard, but due to the low quality this codec is rarely used. A summary of all current supported codecs for WebRTC capable browsers is shown in figure 4-2.

4-3 Browser compatibility

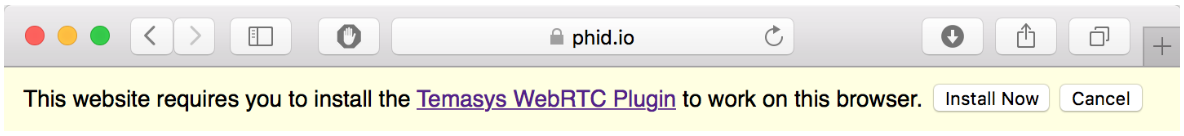
Even though WebRTC has been adopted in most major browsers as shown in figure 4-1, there are still browsers which require additional effort to be completely compatible. To support Safari and Internet Explorer, a browser plugin needs to be installed as described in section 4-3-1. Microsoft Edge does not support WebRTC, but supports Object RTC which is further explained in section 4-3-2.

4-3-1 Browser plugin

Internet Explorer and Safari with respectively 30.26% and 4.07% market share [23] do not support WebRTC. Therefore, an extra browser plugin needs to be installed to perform peer-to-peer video conferences over WebRTC. Even though this defeats the whole added benefit of plugin-less real-time communications, application developers can widen their support by building in added support for this relatively large audience (almost 35% of the browser sessions). The most common plugin is AdapterJS [27] provided by Temasys which is continuously updated to adapt to the latest WebRTC specification. The latest version of AdapterJS can be added by requiring the JavaScript library as follows:

```
<script src="https://cdn.temasys.com.sg/adapterjs/0.13.x/adapter.min.js"></
   script>
```

Once added, AdapterJS provides polyfills for the necessary WebRTC functions and shows a dialog if one of these functions are invoked and the plugin is not installed as shown in the figure 4-3. Once installed, the existing WebRTC functionalities work as expected.


phid.io

This website requires you to install the Temasys WebRTC Plugin to work on this browser. Install Now Cancel

Figure 4-3: AdapterJS prompt for Safari and Internet Explorer to install the required WebRTC plugin

4-3-2 Object RTC

Internet Explorer 11's successor is Microsoft Edge which was released together with Windows 10 in 2015. As of late 2015, Microsoft Edge supports Object Real-Time Communication (ORTC) [28]. Even though ORTC is very similar to WebRTC, there are still fundamental differences and at the time of writing a cross browser video chatting conversation between ORTC and WebRTC is not possible due to codec incompatibility (section 4-2). As already shown in figure 4-2, only audio conferencing is currently possible between WebRTC capable browsers and Microsoft Edge.

Object RTC also differs a lot from WebRTC [29] and at this the time of writing there is no official W3C specification for ORTC. ORTC implements a lower-level API which allows greater flexibility and advanced capabilities. ORTC does not require a signaling server for exchanging generated SDPs. ORTC is more flexible and allows the possibility to have a server dictate the media settings to both parties. The ORTC API also makes it easier for developers to implement advanced capabilities such as simulcast, Scalable Video Coding (SVC) and layered video coding. Currently, the WebRTC project is working on *WebRTC NV* which will be a full convergence of ORTC and WebRTC, there is however little information and no estimates available about when this new standard will be released.

4-4 Mobile support

Mobile devices can support WebRTC in two different ways; via the browser or via a custom-built application. WebRTC via the browser for mobile devices works in the same manner, before the peerconnection is established, permission is first asked to use the (front) camera of the mobile device. Unfortunately, only Android supports WebRTC via the browser via the Google Chrome app since August 2013. At the time of writing support for iOS is still unannounced.

As mentioned before, WebRTC can also be implemented in apps on mobile devices. In fact, this is the only way iOS currently has support for this technology. Because WebRTC is designed and implemented for internet browsers, a mobile app can be more complicated to build. There are two different types of apps:

1. **Hybrid Mobile Application**
   Third-party software is used to support different mobile platforms with the same code-base. One type of hybrid application is a *WebView* (i.e. internet browser) which supports hooks to run native code to, for example, get access to the user's camera and microphone and feed this back to the browser. Another form of hybrid apps are *Compiled Apps* where code is written in e.g. JavaScript or C# and compiled to native code for each platform.

2. **Native Mobile Application**
   When mobile applications are written specifically for e.g. iOS or Android in native code, we speak of native apps. This generally gives a better performance and user experience in the app, but lacks WebRTC support which is implemented in the browser. Luckily, the WebRTC project [30] provides native APIs which can be used.

In chapter 7, a choice is made on which mobile approach is chosen for the performance analysis of Apple iOS and Google Android.

Chapter 5

Congestion Control

Network congestion occurs when a network node carries more data than it can handle. Since WebRTC typically uses UDP which has no congestion control built-in, a custom-made congestion control algorithm is used. WebRTC uses the Google Congestion Control (GCC) [6] which has the ability to dynamically adjust the send and receive rates of the video streams when congestion is detected. WebRTC uses the Real-Time Transport Protocol (RTP) to send its media packets and receives feedback packets from the receiver in the form of Real-Time Control Protocol (RTCP) reports. GCC controls congestion in two ways: delay-based control (section 5-1) on the receiving end and loss-based control (section 5-2) on the sender side.

5-1 Receiver side controller

The receiver side controller is delay-based and compares the timestamps of the received frames with the timestamps these frames were generated. The receiver side controller consists of three different subsystems. The decision to increase, decrease or hold A_r is made by the rate controller (section 5-1-3), which receives a signal s from the over-use detector (section 5-1-2) which uses the estimated output m_i of the arrival-time filter (section 5-1-1) together with an adaptive threshold (γ). These different subsystems of the receiver side controller are shown on the right side of figure 5-1. Based on the outcome of the over-use detector it calculates the receive rate A_r as follows:

$$A_r(i) = \begin{cases} \eta A_r(i-1) & \text{Increase} \\ \alpha R(i) & \text{Decrease} \\ A_r(i-1) & \text{Hold} \end{cases} \tag{5-1}$$

Where $\eta = 1.05$, $\alpha = 0.85$ and $R(i)$ is the measured receive rate for the last 500ms. Lastly, the received rate can never exceed $1.5R(i)$:

$$A_r(i) = \min(A_r(i), 1.5R(i)) \tag{5-2}$$

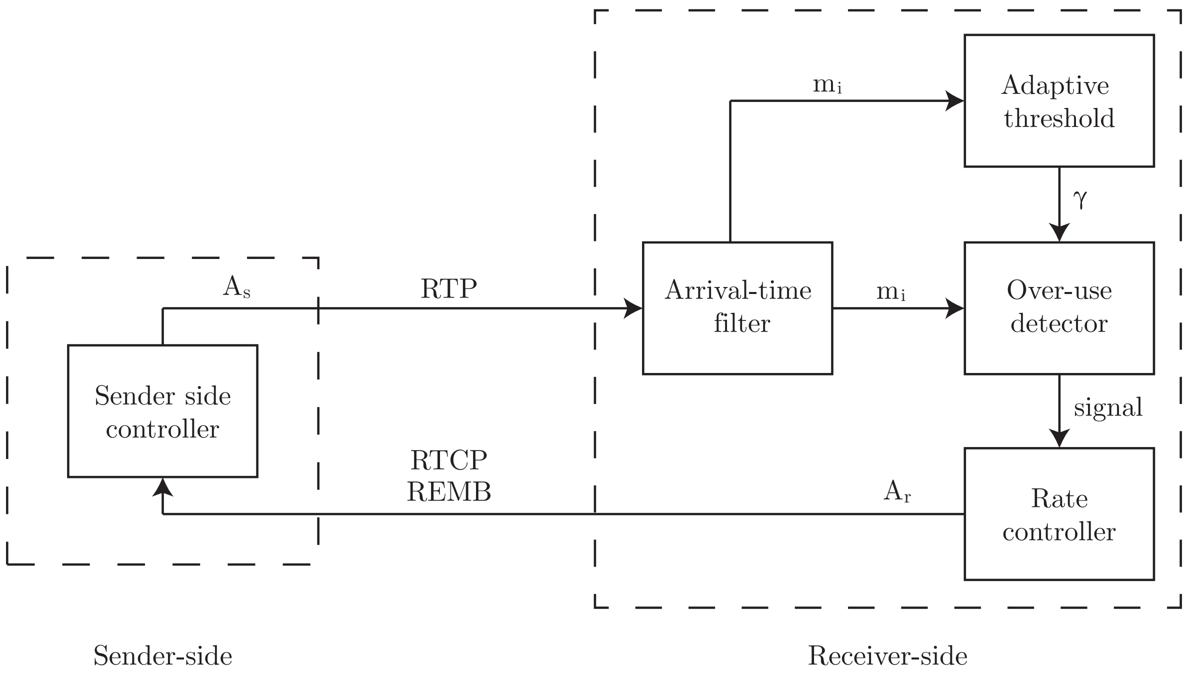


Figure 5-1: Diagram illustrating how sender and receiver-side compute and exchange their available bandwidth

5-1-1 Arrival-time filter

The arrival time filter continuously measures the delay of all packets that are received. It uses the inter-arrival time to calculate the difference in arrival time between two consecutive packets: $t_i - t_{i-1}$. And the inter-departure time to get the time difference in which both packets were sent: $T_i - T_{i-1}$. It then calculates the one-way delay variation d_i, defined as the difference between inter-arrival time and inter-departure time as follows:

$$d_i = (t_i - t_{i-1}) - (T_i - T_{i-1}) \tag{5-3}$$

This delay shows the relative increase or decrease of delay with respect to the previous packet. The one-way delay variation is larger than 0 if the inter-arrival time is larger than the inter-departure time. The arrival-time filter then uses a Kalman filter to reduce the noise and estimate the one-way queuing delay variation m_i. This is based on the measured d_i and previous state estimate m_{i-1} which are relatively weighted according to the Kalman gain:

$$m_i = (1 - K_i) \cdot m_{i-1} + K_i \cdot d_i \tag{5-4}$$

Where K_i is the Kalman gain and m_{i-1} is the previous one-way queuing delay variation. The Kalman gain is high when the error variance is low, and as shown in the equation above, with a high Kalman gain more weight is given to the measured d_i, while a low Kalman gain gives more weight to the previous state estimate m_{i-1}. The Kalman gain is computed as follows:

$$K_i = \frac{P_{i-1} + Q}{\sigma_n^2 + P_{i-1} + Q} \tag{5-5}$$

Where Q is the state noise variance set at 0.001, P_i is the system error variance and σ_n^2 is the measurement noise variance. An exponential moving average filter is used since no measurement noise is available which is proven to be representative according to [31]. The noise variance can then be computed as follows:

$$\sigma_n^2(i) = \beta * \sigma_n^2(i-1) + ((1-\beta) \cdot (d_i - m_{i-1})^2) \tag{5-6}$$

Where β is set to 0.95, d_i comes from equation 5-3 and $d_i - m_{i-1}$ represents the residual (also used to multiply with the Kalman in equation 5-4). Finally, the system error variance used in equation 5-5 is calculated as:

$$P_i = (1 - K_i) \cdot (P_{i-1} + Q) \tag{5-7}$$

Where P_0 is set to 0.1 as an initial value and serves as the input for obtaining the Kalman gain (equation 5-5) in the next iteration.

5-1-2 Over-use detector

The estimated one-way queuing delay variation (m_i) is compared to a threshold γ. Over-use is detected if the estimate is larger than this threshold. The over-use detector however does not signal this to the rate controller unless over-use is detected for a longer period of time, which is currently set to *100ms* [32]. Under-use is detected when the estimate is smaller than the negative value of this threshold and works in similar manner. A normal signal is triggered when $-\gamma \leq m_i \leq \gamma$.

$$signal_i = \begin{cases} m_i > \gamma_i & \text{Over-use} \\ m_i < -\gamma_i & \text{Under-use} \\ -\gamma_i \leq m_i \leq \gamma_i & \text{Normal} \end{cases} \tag{5-8}$$

The value of the threshold has great impact on the overall performance of the congestion algorithm. A static threshold γ can easily result in starvation in the presence of concurrent TCP flows as shown in [33]. Therefore, a dynamic threshold has been implemented:

$$\gamma_i = \gamma_{i-1} + (t_i - t_{i-1}) * K_i * (|m_i| - \gamma_{i-1}) \tag{5-9}$$

Where the value of the gain K_i depends on whether $|m_i|$ is larger or smaller than γ_{i-1}:

$$K_i = \begin{cases} K_d & |m_i| < \gamma_{i-1} \\ K_u & \text{otherwise} \end{cases} \tag{5-10}$$

Where the initial value of γ_0 is set to 0.2 ms and $K_d < K_u$. This causes the threshold γ to increase when the estimated m_i is not in the range $[-\gamma_{i-1}, \gamma_{i-1}]$ and decrease when it does fall in that specific range. This helps increasing the threshold when e.g. a concurrent TCP flow enters the bottleneck and avoids starvation of the WebRTC streams. According to [34] this adaptive threshold results in better fairness with competing traffic and is able to reduce the queueing delay with up to 50% when compared to using a static threshold.

5-1-3 Rate controller

The rate controller decides whether to increase, decrease or hold, depending on which signal is received from the over-use detector. Initially, the rate controller keeps increasing the available receive bandwidth until over-use is detected by the over-use detector. Figure 5-2 shows how the rate controller adjusts its state based on the signals received by the over-use detector.

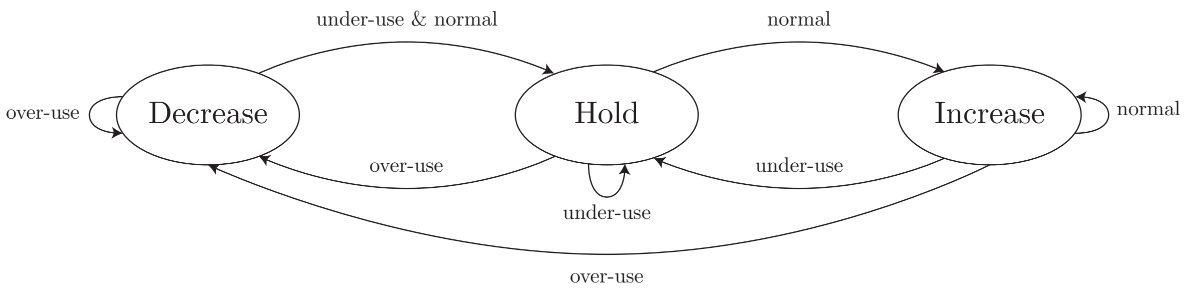


Figure 5-2: How the rate controller decides its state based on the signal output of the over-use detector

A congestion/over-use signal always results into decreasing the available bandwidth, while under-use always results into keeping the bandwidth unchanged to make sure the queues are first being drained. The state of the rate controller translates into its receive rate A_r as shown in Equation 5-1. A_r is sent back to the sender as a Receiver Estimated Maximum Bandwidth (REMB) [35] message in a RTCP report (figure 5-1).

5-2 Sender side controller

The sender-side is loss-based and computes the target sending rate A_s in kbps and is shown on the left side of figure 5-1. It runs every time (i) a RTCP report is received from receiver and is based on the fraction of lost packets $f_l(i)$:

$$A_s(i) = \begin{cases} A_s(i-1)(1-0.5f_l(i)) & f_l(i) > 0.1 \\ 1.05A_s(i-1) & f_l(i) < 0.02 \\ A_s(i-1) & \text{otherwise} \end{cases} \tag{5-11}$$

If the packet loss is between 2% and 10%, the send rate remains unchanged. If more than 10% of the packets are reported lost the rate is multiplicatively decreased, and if the packet loss is smaller than 2%, the sending rate is linearly increased. Furthermore, the sending rate is limited, so that it never exceeds the last received receiving rate $A_r(k)$ which is obtained through REMB messages from the receiver as seen in figure 5-1:

$$A_s(i) = \min(A_s(i), A_r(k)) \tag{5-12}$$

Chapter 6

Related Work

In this chapter the available literature of WebRTC based video conferencing is studied. Most related work focuses on single aspects of the protocol or use outdated versions of WebRTC in their performance analyses. Since WebRTC and its congestion control algorithm are still rapidly changing, a distinction is made between recent and older studies. Interesting to see is that most remarks and proposals to improve the congestion control algorithm made in the past, have actually helped improve and are even implemented in the Google Congestion Control algorithm used today. Older research is discussed in section 6-1. Papers focusing on single performance characteristics are studied in section 6-2. Finally, related work which propose several improvements to the congestion control algorithms are discussed in section 6-3.

6-1 Early studies

Other papers have studied the performance of WebRTC since its introduction in late 2011. In [36] a comprehensive performance evaluation is conducted on a less mature version of WebRTC (2013). Real world conditions are simulated over a series of experiments. They argue that the RRTCC algorithm (predecessor of GCC) has the following limitations:

- Bandwidth utilization collapses when latency exceeds 200ms
- When competing with TCP cross traffic, the WebRTC flows starve
- When competing with late arriving RTP flows, the bandwidth is not distributed equally

Interesting to see is that all flaws addressed in this paper, have been fixed. The data rate no longer drops at high latencies but instead responds to latency variation. Also the competition between TCP and RTP is more fair due to the newly introduced dynamic threshold (equation 5-9) and the "late-comer" effect when bandwidth is not equally distributed when competing RTP flows are added is also solved as shown in [32]. In the performance analysis in chapter 8 the tests shown in [36] will be repeated to verify if all issues have been solved.

6-2 Performance analyses

In the following section related work will be studied which focus on aspects of WebRTC's performance which form a basis for the performance analysis conducted in chapter 8. Related cross traffic work is discussed in section 6-2-1. Followed by analyzing multi-party performance in section 6-2-2. Real network effects and their effects on WebRTC are discussed in 6-2-3. Related work is analyzed which compare Google Chrome and Firefox in section 6-2-4. Lastly, different quality measurement studies are discussed in section 6-2-5.

6-2-1 Cross traffic

In [32] the design of the Google Congestion Control algorithm as currently used in the WebRTC stack is presented. The results in this paper focus on the fairness between different concurrent RTP streams and RTP streams competing with TCP flows while being limited in bandwidth. Fairness with cross traffic is tested using the Jain Fairness index which provides a representative measure of fairness as shown in [37]. Jain's Fairness Index can be computed as follows:

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} \tag{6-1}$$

Where the fairness is computed for n users and x_i is the throughput for the ith connection. $\mathcal{J}$ indicates the fairness ranging from $\frac{1}{n}$ to 1 which respectively represent the worst and best case. According to [32], cross traffic fairness is achieved and therefore improved with respect to earlier research done in [36]. As mentioned in section 6-1, cross traffic tests will be conducted in chapter 8 to verify these findings using Jain's Fairness Index.

6-2-2 Multi-party

Unfortunately, there is limited related work available where multi-party (i.e. > 2 persons) WebRTC video conferences are tested. [38] analyzes the Janus WebRTC gateway focusing on its performance and scalability for audio conferencing. This paper shows that the gateway scales almost linearly when used as an audio bridge. Video conferencing is however not tested. In [39] both mobile devices and multi-party video conferencing via WebRTC is tested. A subjective questionnaire is conducted where three different WebRTC powered websites are tested. The study shows that the perceived Quality of Experience (QoE) is heavily depended on the CPU of the mobile device, and that even the most powerful mobile device is not able to match the perceived QoE when laptops are used.

6-2-3 Network effects

A more realistic performance study using real network effects is done in [40] where the performance of WebRTC is measured while moving around with a laptop at walking speeds in both urban and suburban areas. Even though the WebRTC implementation used is outdated (2013), this paper suggests that WebRTC relies too strong on packet loss, which seems to happen frequently while roaming around, which results into underutilization of the channel and low data rates. In [41] a performance analysis of WebRTC over LTE is done. Several tests are run with two mobile users inside an LTE simulated environment where the flows are subject to interference, packet loss and different queue sizes. The study shows that WebRTC is sensitive to these different scenarios but lacks depth in what is actually happening.

6-2-4 Cross browser

In [42] a WebRTC benchmark is presented focusing on media engine capabilities and the time it takes to establish a connection, comparing Firefox and Google Chrome. Results show that Google Chrome performs significantly faster at decoding while maintaining the original framerate and resolution plus that Google Chrome sets up a call faster than Firefox. In [43] these two browser are also tested. Similarly to [42], this paper also states that setting up a peer connection on Firefox takes significantly longer than it does with Chrome (respectively 10 seconds vs 3 seconds) which is assumed to be due to the Trickle ICE protocol which Firefox at that time did not support yet. [42] also analyzes the power consumption of WebRTC, but is unfortunately limited to the data channel. Results show that WebRTC was able to handle 50 concurrent file transfers while using less than 50% CPU on both browsers, where Firefox uses a little less CPU than Chrome.

6-2-5 Quality measurements

Quality of Experience (QoE) is a way to measure customer satisfaction when receiving a video stream. As defined in [44], this includes the complete end-to-end system effects such as client, terminal, network, video encoder/decoder, etc. and describes the overall acceptability of the service as perceived by the end user. QoE is often measured subjectively and thus results are hard to compare since different users may interpret QoE differently. [45] defines a standard for subjective QoE measurements and a lot of different conditions have to be met which can be time costly. This thesis therefore focuses on objective measurements provided by WebRTC's RTCStatsReport in which results are fully measureable and comparable in hard numbers.

In [46, 47] different methods for measuring QoE objectively are discussed. [46] measures QoE objectively based on the video quality, the audio-video synchronization and the communication delay. For the video quality, the received video is compared with the sent video and then mapped to a subjective Mean Opinion Score (MOS). And for the audio-video synchronization a custom made video is used to measure the audio-video offset accurately.

6-3 Proposed improvements

This section analyzes different recent studies which propose improvements to the current GCC algorithm. In [48] an Early Packet Loss Feedback (EPLF) algorithm is proposed in addition to the existing congestion control algorithm for a better experience when packet loss occurs due to a bad Wi-Fi link. Since the RTCP report takes at least one RTT to adjust for packet loss, EPLF uses the MAC layer of the Wi-Fi link which sends a spoofed NACK when a transmission fail is detected. This NACK triggers a retransmission immediately which reduces the amount of video freezes.

A dynamic alpha for decreasing the rate at the receiving end (equation 5-1) is proposed in [49]. Currently in GCC this value is fixed and set to 0.85 which decreases the rate with 15% when overuse is detected. [49] argues that this decrease in rate should not be fixed, but depend on how far the threshold in the over-use detector is exceeded (equation 5-8). Results show that the incoming rate is increased with 33% while receiving a 16% lower round-trip time on average while having no significant impact on the packet loss.

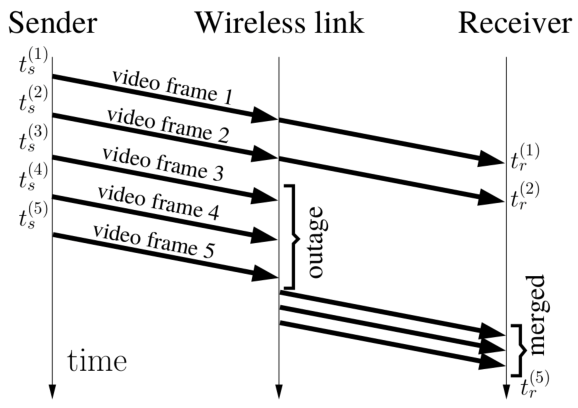


Figure 6-1: Packet grouping when channel outages occur as proposed in [50]

Lastly, an optimization for the current GCC algorithm is proposed in [50]. It shows an improvement of GCC when dealing with channel outages by introducing an extra pre-filter module at the receiving end before the arrival time filter module (figure 5-1). This is because the current implementation interprets channel outages as congestion. The approach in [50] pre-filters packets and merges packets that arrive in bursts (due to a channel outage) as shown in figure 6-1. This pre-filtering increases the throughput by 20% [50] without significantly increasing the queuing delay.

Chapter 7

Experimental Setup

This chapter describes the parameters and metrics of the testbed used to conduct the performance analysis in the next chapter. An overview of the basic setup is shown in figure 7-1. The hardware used is described in section 7-1. Information about the browser setup used is given in section 7-2. The tool used to manipulate the network characteristics is explained in section 7-3. WebRTC's statistic analysis is described in section 7-4 followed by setup details of cross traffic and mobile devices in respectively section 7-5 and section 7-6. Finally, the chapter is concluded in section 7-7 where the setup for multi-domain testing is discussed.

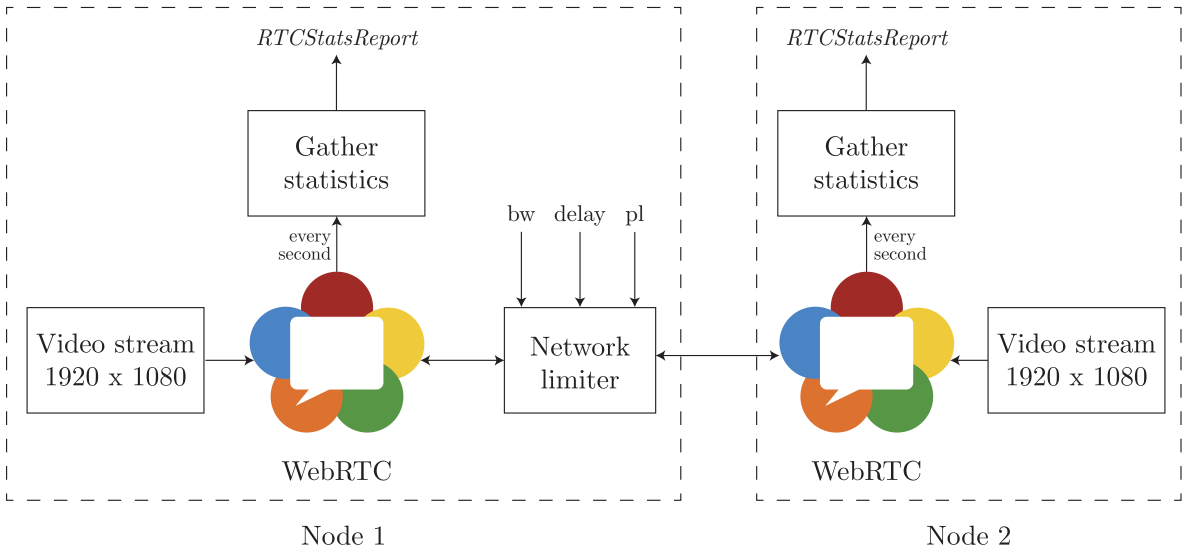


Figure 7-1: Experimental test setup used for performance analysis described in this chapter

7-1 Hardware

In the experimental setup high end computers are used to ensure that there is enough computation power for encoding and decoding the video streams. The basic setup consists of an Intel Core i7 4790K desktop running at 4.4GHz with 16GB RAM and a laptop with an Intel Core i7 4980HQ at 4.0GHz with 16GB RAM.

A separate Node.JS signaling server is also necessary to setup the call between the two peers as explained in section 2-2. This service is very lightweight and deployed locally on one of the machines to perform this initial setup.

In the performance analysis, network conditions are simulated. To avoid additional latency and network limitations we ensure all standalone PCs are on the same local network and are connected via a wire.

7-2 Browser setup

Several browsers support WebRTC as explained in chapter 4. Since Google Chrome is leading with its GCC implementation [32] and has the ability to inject a custom media feed (section 7-2-1), the Google Chrome browser is chosen for most tests. Unless mentioned otherwise, the most recent version of Google Chrome is used (version 52) at all clients with the default agreed audio codec: OPUS and video codec: VP8.

7-2-1 Inject custom media feed

When a WebRTC call is initialized, it requests permission to access your webcam and microphone which respective streams are then transmitted to the other party during the call. In order to maintain the same quality stream and therefore reduce variations that occur with e.g. different quality webcams, a workaround is necessary to provide the browser with a steady, comparable media stream.

Instead of using a webcam feed and microphone audio signal, Google Chrome's fake-device functionality is exploited to feed the browser a looping video and audio track to obtain comparable results. A custom RAW media file can be injected by starting Google Chrome with the command shown in listing 7.1. For all tests (unless mentioned otherwise) the same video is used with a resolution of 1920x1080 at 50 frames per second with constant bitrate: *in_to_tree*[1]. This video is fed to WebRTC for both nodes as shown at the far left and far right of figure 7-1.

```
open -a "Google Chrome" --args --use-file-for-fake-video-capture=/Users/
    bartjansen/in_to_tree_1080p50.y4m --use-fake-device-for-media-stream
```

Listing 7.1: How to start Google Chrome with custom media stream

[1]https://media.xiph.org/video/derf/

7-2-2 Change video codec

As explained in section 2-2, the Session Description Protocol is first exchanged between the participating parties which describes the capabilities of each client. One thing described in the SDP are all video codecs supported for each user plus that they are ordered in the preference in which they are used. Based on the different SDPs and their (different) codec preferences, a video codec is decided. Currently, VP8 is preferred in almost all browsers. But this can be changed by altering the generated SDP in the WebRTC setup code.

```
m=video 9 UDP/TLS/RTP/SAVPF 100 101 107 116 117 96 97 99 98
a=rtpmap:100 VP8/90000
a=rtcp-fb:100 ccm fir
a=rtcp-fb:100 nack
a=rtcp-fb:100 nack pli
a=rtcp-fb:100 goog-remb
a=rtcp-fb:100 transport-cc
a=rtpmap:101 VP9/90000
a=rtcp-fb:101 ccm fir
a=rtcp-fb:101 nack
a=rtcp-fb:101 nack pli
a=rtcp-fb:101 goog-remb
a=rtcp-fb:101 transport-cc
a=rtpmap:107 H264/90000
a=rtcp-fb:107 ccm fir
a=rtcp-fb:107 nack
a=rtcp-fb:107 nack pli
a=rtcp-fb:107 goog-remb
a=rtcp-fb:107 transport-cc
```

Listing 7.2: part of Session Description Protocol showing the video codec preference

An example of part of the generated SDP is shown in listing 7.2, here it can be seen that VP8, VP9 and H.264 get respectively mapped to 100, 101 and 107. The first line specifies the order in which these codecs are preferred and this order can be simply altered by putting either 101 or 107 first for respectively VP9 or H.264. This has to be done for all clients to make sure these specific video codecs are actually used. In the next chapter in section 8-7, different video codecs are analyzed using this method.

7-3 Network limiter

As mentioned before, different network characteristics are simulated to test the performance of WebRTC. *Dummynet* is used to simulate different network characteristics which is known for its accurate performance as shown in [51]. Thus before reaching the other peer, the packets are subject to different network conditions as shown at the network limiter block in figure 7-1. Dummynet makes it possible to specify the following network characteristics:

- Latency or delay in both uplink and downlink
- Percentage of packet loss in both uplink and downlink
- Link capacity or available bandwidth in both uplink and downlink

Dummynet also allows to apply these rules only to specific transport protocols (TCP/UDP) or specific ports. This is useful for the setup since the limits only need to apply to UDP traffic. An example of a simple three minute Dummynet test is shown in listing 7.3. First two pipes are generated for incoming and outgoing UDP traffic. Then every minute the conditions change: the bandwidth is limited to 1Mbps, 100ms latency is applied and 20% of all packets are dropped. It is important to note that e.g. setting the delay after 1 minute removes the earlier set bandwidth constraint.

```
# define different pipes
ipfw add pipe 1 in proto udp
ipfw add pipe 2 out proto udp


# limit available bandwidth to 1Mbps for both uplink and downlink
ipfw pipe 1 config bw 1000Kbit/s
ipfw pipe 2 config bw 1000Kbit/s

# wait 1 min
sleep 60

# add 100ms latency
ipfw pipe 1 config delay 100ms
ipfw pipe 2 config delay 100ms

# wait 1 min
sleep 60

# drop 20% of all incoming and outgoing packets
ipfw pipe 1 config plr .2
ipfw pipe 2 config plr .2
```

Listing 7.3: Dummynet example

For the simpler tests in section 8-1, the network conditions do not change during the call. In the subsequent section, the characteristics change continuously, especially the latency which gets changed every half second to provide a gradual increase (instead of incremental as with other properties). Dummynet is luckily able to handle these fast changes. An example of a linearly changing latency from 0ms to 250ms is given in the listing below.

```
# linearly change from 0ms to 250ms
for ((i=1; i<=120; i++))
do
    ipfw pipe 1 config delay $(((250*i)/120))ms
    ipfw pipe 2 config delay $(((250*i)/120))ms
    sleep .5
done
```

Listing 7.4: Dummynet linearly changing latency from 0ms to 250ms

7-4 Statistics

To gather all necessary performance analysis statistics WebRTC's built in *RTCStatsReport* can be exploited to obtain detailed statistics of all transmitted and received data. Once a call is established *getStats*() can be invoked on the active *RTCPeerConnection* which returns its current statistics as drafted by W3C [52]. These statistics are gathered at 1*second* intervals at both end points as also shown in figure 7-1. These statistics are however not directly usable and usually not in the form needed to process them. A Javascript library named *getStats.js* [53] is used together with a custom-made framework to obtain the necessary call characteristics. All data rates produced by the PeerConnection's *getStats* are for instance accumulated and need the previous captured *RTCStatsReport* to provide the data rate for that particular second.

These statistics provide basic insight about the call characteristics in the performance analysis. For a more in depth comparison of the perceived quality of the streams, a form of QoE should be used as explained in section 6-2-5. Most tests run at least 5 minutes and are averaged out over 5 different runs to avoid statistical errors. The custom-made stats framework then outputs tailored arrays which are put into Matlab which is used to produce the shown graphs.

7-5 Cross traffic

In section 8-4 WebRTC's performance is tested while competing with cross traffic. Cross traffic is other network traffic sharing the same bottleneck which could lead to network congestion. When cross traffic is present, it is important that the bandwidth is spread equally to provide fairness. WebRTC uses UDP which is unresponsive when it comes to fairness. A plain UDP flow will therefore always take up all available bandwidth if required and starve potential other flows. Even though WebRTC uses UDP, it uses GCC on top of it which provides better fairness when competing with cross traffic.

Two forms of cross traffic are tested. First, fairness is studied when multiple WebRTC flows compete with the same bottleneck. In this test, additional machines are added which conduct concurrent video calls while connected through the same network limiter. In the second scenario, a single WebRTC flow competes with TCP flows which are generated via *iperf* [54] as shown below.

```
iperf3 -s
# start server (-s) on default port (5201)


iperf3 -c 192.168.178.17 -p 5201 -t 600 -J
# start client (-c) and transmit TCP flow (default)
# to server @ 192.168.178.17
# port 5201
# for 600 seconds (10 minutes)
# output to JSON
```

Listing 7.5: iperf3 usage to start server and client

The listing above shows how easy it is to start the server at one end and transmit TCP packets from the client to the server on the other end. iperf runs together with Dummynet which limits the link capacity and allows the inter-protocol fairness to be computed.

7-6 Mobile

In section 8-5 the mobile performance of WebRTC is analyzed and compared to the performance on desktops. Since Apple iOS and Google Android account for more than 96% of the total smartphone market share [55], the focus is limited to these mobile operating systems. In section 7-6-1 the setup for Google Android is described followed by Apple iOS in section 7-6-2.

7-6-1 Google Android

For the Android performance analysis, a Samsung Galaxy S6 running Android 5.1.1 is used. As discussed in 4-4, Android's browser has had support for WebRTC since August 2013 and therefore the same codebase can be used as with the desktop tests. Even though the default browser supports WebRTC, the Google Chrome app (v52) is used which functionalities are kept in sync with its desktop counterpart. Even though Google Chrome is used as its browser, it is unfortunately not possible to inject a custom video stream and therefore the camera of the device is used for the performance analysis. By default, it is only possible to use the front facing camera for WebRTC calls in a mobile browser. Luckily, there is a workaround which allows access to the rear camera which provides a better quality stream and is more comparable to the Full HD test sequence used elsewhere.

7-6-2 Apple iOS

The test setup for Apple iOS is unfortunately more complicated. At the time of writing browsers on iOS have no support for WebRTC yet. WebRTC capabilities are only possible through native code and therefore an application needs to be created to conduct WebRTC video chats. Fortunately, Cordova[2] can be used with the existing HTML/JavaScript codebase and only requires the WebRTC functionalities to be built (i.e. the signaling, exchanging SDPs etc. is taken care of by the existing application). Cordova wraps the existing codebase in a hybrid iOS application and allows JavaScript to invoke native (Objective C/Swift) functions.

Fortunately, there is an actively maintained Cordova plugin *cordova-plugin-iosrtc* [56] available. This plugin exposes the full WebRTC JavaScript API according to the W3C standards and allows JavaScript to gain access to the iOS camera, set up a RTCPeerConnection and display both the local and remote video streams in native VideoViews.

The implemented application is installed on an Apple iPhone 6 with iOS 9.3.1 which is used the conduct the experiments. The iPhone 6 has similar performance compared to the Samsung Galaxy S6 [57] and should therefore provide a fair comparison. Similarly as with Android, it is not possible to feed the peer connection with a custom video and therefore the native $getUserMedia()$ code is modified so that the rear camera is used.

[2]https://cordova.apache.org/

7-7 Multi-domain testing

All network effects are simulated by Dummynet as explained in section 7-3. With these tests, the WebRTC streams are not subject to the effects that occur when data is transferred over the internet. As part of the performance analysis these actual effects are also studied to verify the use of the network limiter. Therefore, several virtual machines are used across the world and tested against the host machine which is situated in Utrecht, The Netherlands. Since WebRTC runs via the internet browser, it is not possible to use virtual machines running in headless mode (i.e. via the console/terminal). To fulfill this requirement, Microsoft Azure [58] is used to set up several virtual machines running Windows 10. With Windows 10, Google Chrome can be easily installed together with the injected video stream as explained in section 7-2-1. To ensure that the performance analysis is not limited to the necessary CPU power for encoding and decoding the streams, a 720p variant of the used stream is used. Microsoft Azure offers its services in a wide range of locations where the three following locations are used to connect to the host in Utrecht, The Netherlands:

- Amsterdam, The Netherlands, with a latency of approximately 8ms
- San Francisco, United States, with a latency of approximately 165ms
- Singapore, Singapore, with a latency of approximately 305ms

The performance analysis for these different locations is given in section 8-8.

Chapter 8

Performance Analysis

Based on the test setup explained in chapter 7 several tests are conducted to see how the performance in specific WebRTC video conferences are affected. In section 8-1 the effects of additional latency, packet loss or limited bandwidth during a WebRTC based video chat are analyzed. In section 8-2, the network adaptability capabilities are tested of WebRTC by changing these different network parameters during the call. Section 8-3 describes the performance of WebRTC for different multi-party solutions up to 4 persons. In section 8-4 the fairness with competing cross traffic is tested. The performance on mobile devices is analyzed in section 8-5 followed by a cross browser comparison in section 8-6. Different video codecs are compared in section 8-7 and the chapter is concluded in section 8-8 where WebRTC calls over the actual internet are analyzed.

8-1 Network characteristics

As seen in chapter 5, a lot of WebRTC's performance depends on the connection characteristics between the two participating parties. The receiver and sender side respectively adjust the data rate based on latency variation and packet loss while the effects of limited bandwidth will also affect the call. In this section a reference point is made on an unconstrained link in section 8-1-1 followed by tests how changing latency (section 8-1-2), packet loss (section 8-1-3) and available bandwidth (section 8-1-4) impact the performance of WebRTC in a simple two person WebRTC video chatting session.

8-1-1 Unconstrained link

As a reference, the performance of WebRTC without limitations to the network is tested first. This can be used as a reference for subsequent tests. The average results for a five minute unconstrained test are shown in table 8-1.

	Client 1	**Client 2**	**Average**
Audio rate (kbps)	66,5 ± 10,7	71,55 ± 6,15	69,0 ± 8,4
Video rate (kbps)	2449,7 ± 322.2	2462,7 ± 267,9	2456,2 ± 295,0
RTT (ms)	1,146 ± 0,315	1,122 ± 0,265	1,134 ± 0,29
Packet loss(%)	0.00	0.00	0.00
Avg received resolution (px)	1920 x 1080	1920 x 1080	1920 x 1080
Avg received framerate (FPS)	49.95	49.88	49.92

Table 8-1: Average results for unconstrained test

From the table above it can be seen that WebRTC is currently limited to sending at 2.5Mbps as set in the browser[1]. Furthermore, there is negligible latency (around 1ms) and packet loss, and WebRTC is perfectly capable of transmitting the video at its original resolution (1920 x 1080) and framerate (50 FPS) using the testbed described in chapter 7. These results above are used in the subsequent sections to compare different network effects with.

8-1-2 Latency

First, the effects of communication delay is tested by subjecting a five minute WebRTC call to additional latency. Three different latencies are tested in both uplink and downlink directions: 100ms, 250ms and 500ms, making the respective round-trip-times twice these values. Results are shown in table 8-2 and the resulting data rate over time is shown in figure 8-1.

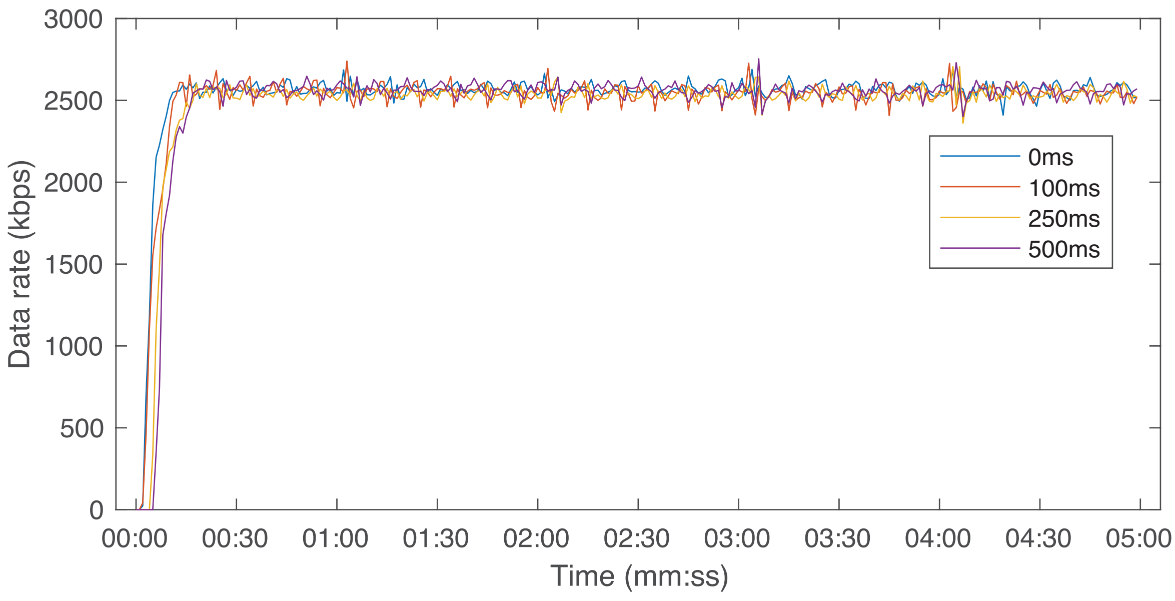


Figure 8-1: Data rate while being subject to different latencies

As expected, extra latency does not affect the data rate or quality of the stream since the Google Congestion Control algorithm only responds to latency variation, but does add the

[1]https://chromium.googlesource.com/external/webrtc

	Audio (kbps)	Video (kbps)	RTT (ms)	P.L.	Avg resolution	FPS
100ms	62,1 ± 13,5	2449,5 ± 307,3	200,39 ± 0,65	0%	1920 x 1080	49.95
250ms	60,6 ± 10,5	2433,0 ± 372,8	500,38 ± 0,54	0%	1920 x 1080	49.96
500ms	61,7 ± 14,6	2423,4 ± 397,3	1000,3 ± 0,78	0%	1920 x 1080	49.89

Table 8-2: Average results of limiting latency

specified delay to the conversation. ITU-T Recommendation G.114 [59] specifies that one-way transmission delay should preferably be kept below 150ms, and delays above 400ms are considered unacceptable. The only other effect when adding delay, is that it takes longer to set up that call and for data to flow between both end points (figure 8-1). Once data flows it takes approximately 10 seconds to reach its maximum data rate, regardless of what delay is added. This startup delay is less than expected from the data rate increase factors in the GCC equations, because WebRTC uses a ramp up function[2] to get to the highest possible data rate as soon as possible once a connection is established.

8-1-3 Packet loss

Packet loss occurs when transmitted data fails to reach its destination for example due to network congestion or limited wireless connectivity. For the next series of tests, a certain percentage of all packets are dropped for both sent and received packets. The characteristics of calls are compared with 0%, 5%, 10% and 20% packet loss. The results are shown in table 8-3 and figure 8-2.

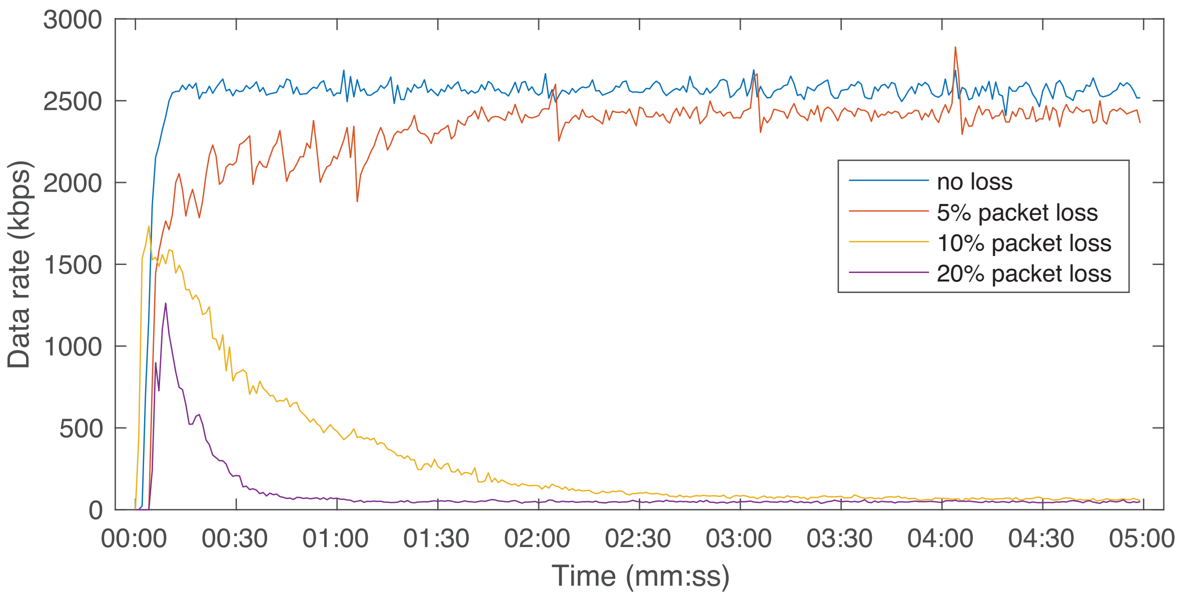


Figure 8-2: Effects of different percentage of packet loss on the data rate

The resulting packet loss is really close to the set values as seen in the table. The results further match the expectations from equation 5-11. Google Congestion Control only decreases

[2]https://bugs.chromium.org/p/webrtc/issues/detail?id=1327

	Audio (kbps)	Video (kbps)	RTT (ms)	P.L.	Avg resolution	FPS
5%	47,4 ± 9,0	2241,9 ± 374,8	1,13 ± 0,29	4,96%	1920x1080	49.71
10%	40,0 ± 6,0	255,9 ± 386,7	1,08 ± 0,26	10,18%	578,7 x 325,5 (30%)	24.26
20%	32,3± 7,0	65,6 ± 173,6	1,034 ± 0,118	20,53%	337,2 x 189,7 (18%)	10.8

Table 8-3: Average results of adding packet loss

the sending rate when more than 10% of packet loss is detected. The sending rate remains unchanged when the packet loss is between 2% and 10% and the rate is increased when less than 2% of the packets are lost. Therefore, 5% packet loss slowly converges to the maximum data rate and at 10% packet loss, the data rate converges to a minimum of 50kbps which almost completely consists of audio data which is not affected by GCC since it is considered negligible [60]. The received video at this point is played at a minimum 240x135 resolution and unacceptable ~1FPS.

8-1-4 Limited bandwidth

In this section the bandwidth is limited in both uplink and downlink to 1500kbps, 750kbps and 250kbps. Results are shown in table 8-4 and figure 8-3.

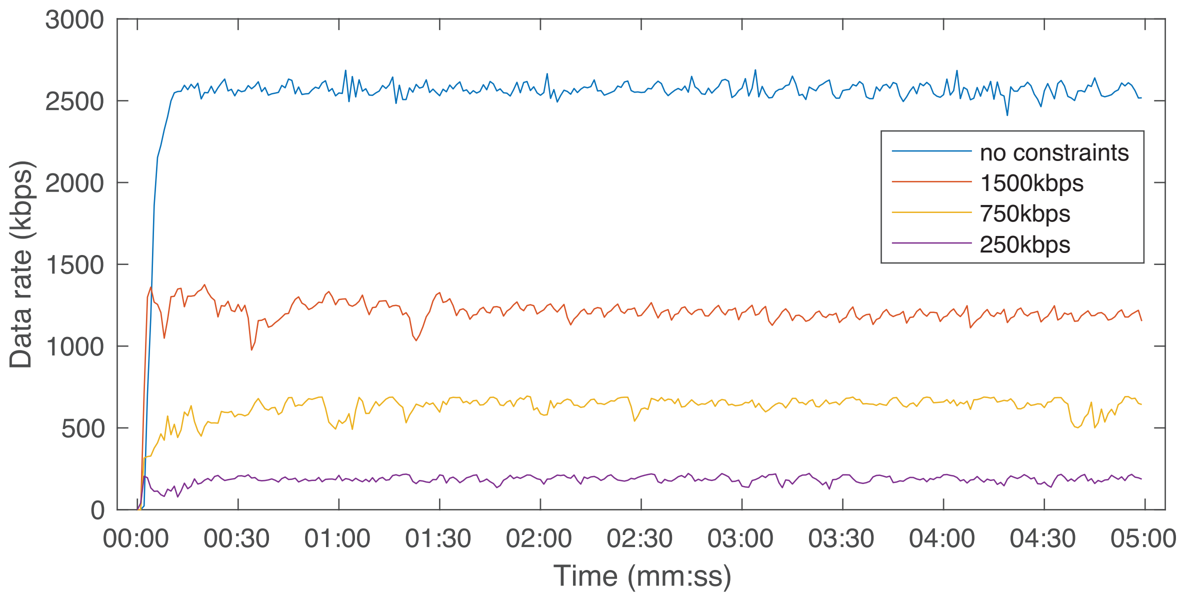


Figure 8-3: Data rate under limited bandwidth

	Audio (kbps)	Video (kbps)	RTT (ms)	P.L.	Avg resolution	FPS
1500kbps	62,1 ± 15,4	1149,9 ± 128,8	23,0 ± 15,1	0%	1019,6 x 573,5 (53%)	49.6
750kbps	65,6 ± 14,1	558,9 ± 89,3	105,62 ± 72,56	0.38%	773,2 x 434,9 (40%)	46.95
250kbps	50,5 ± 13,4	138,6 ± 40,9	360,95 ± 182,2	4%	277,6 x 156,2 (14%)	36.75

Table 8-4: Average results of limiting bandwidth

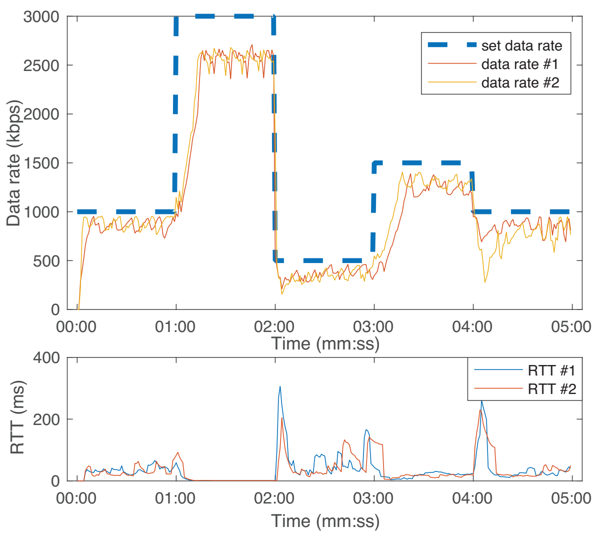
When the bandwidth is limited, WebRTC uses 80% of the available bandwidth and is able to maintain a constant data rate. What is interesting to see is that capping the bandwidth also results into packet loss and higher round-trip times, especially at 250kbps. Plus, that the video resolution is more heavily affected than the framerate when compared with the additional packet loss tests in section 8-1-3 which means that packet loss results into a lower framerate. Lowering the bandwidth further in additional tests, shows that a minimum of 20kbps is necessary to establish a video call between two parties. At least 250kbps is however necessary to obtain a somewhat acceptable framerate (25 FPS) at the lowest possible resolution (240 x 135).

8-2 Network adaptability

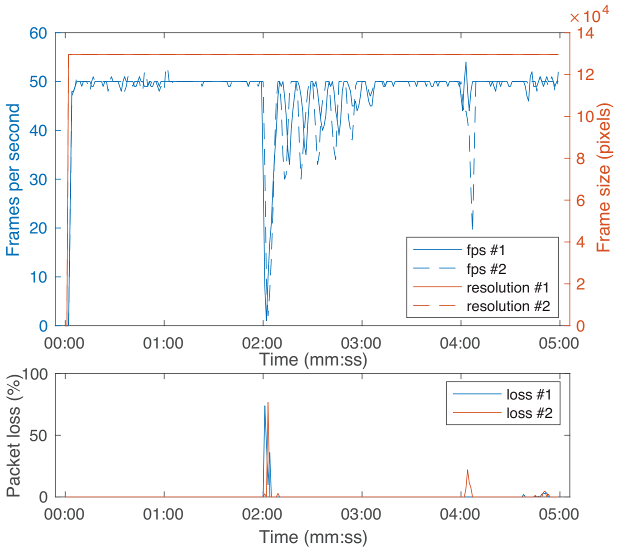
Experiencing a constant delay or being limited by a constant bandwidth as discussed in the previous section is one thing. A more common scenario is that these network characteristics change during a call. How WebRTC adapts to changing network conditions during the call is studied in this section. For each test, the network constraints are changed every minute according to a predefined schema. In section 8-2-1, 8-2-2 and 8-2-3 a single network characteristic is continuously changed and in section 8-2-4 both the available bandwidth and the delay are changed.

8-2-1 Changing bandwidth

The available bandwidth in both uplink and downlink can change during the course of a call. In this section this phenomenon is simulated by capping the available bandwidth every minute consecutively at 1 Mbit, 3 Mbit, 500 Kbit, 1500 Kbit and 1 Mbit as shown in the top graph of figure 8-4a.



(a) Data rate (top) and RTT (bottom)



(b) Framerate (top) and loss (bottom)

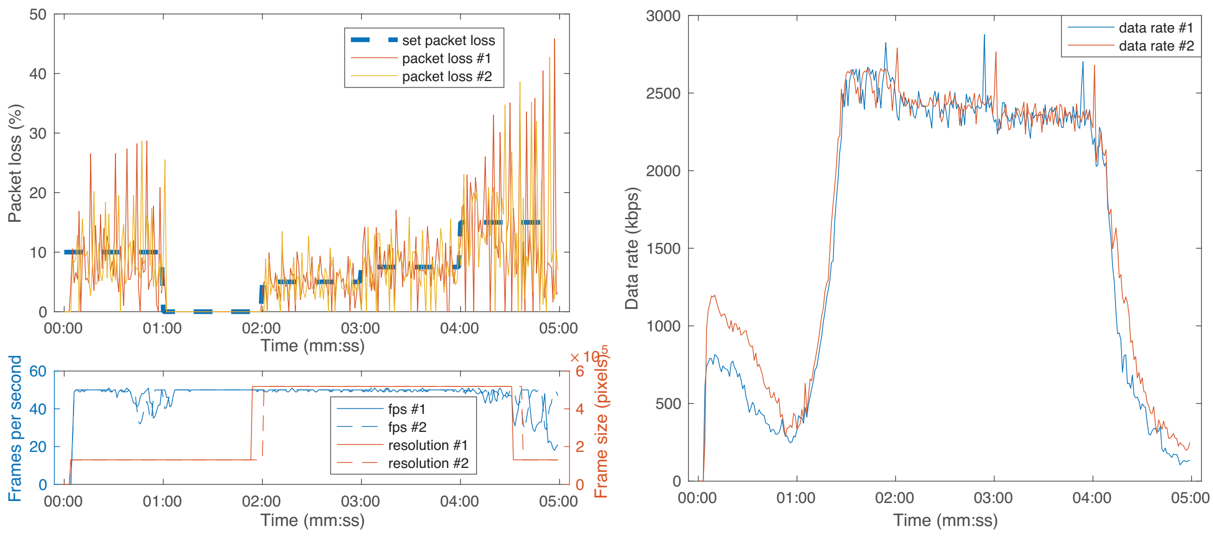
Figure 8-4: Data rate, RTT, framerate, resolution and packet loss for changing bandwidth

In this scenario, the bandwidth utilization is 77%, which is slightly less than the 80% bandwidth utilization when the available bandwidth is not changed in section 8-1-4. This is mostly due to the delay it takes to reach a steady bandwidth when more bandwidth becomes available at minute 1:00 and 3:00 where respectively 16 and 18 seconds are needed. As seen in equation 5-1, this delay confirms what can be expected from GCC since the bandwidth increases linearly with factor 1.05 when under-use is detected (equation 5-1). This is because REMB messages are sent every second which increase the bandwidth with 5% every second. Theoretically a rate of $1000kbps * 1.05^{16} \approx 2200kbps$ would be expected after the first minute and $500kbps * 1.05^{18} \approx 1200kbps$ after bandwidth is freed at the third minute, both close to the respectively reached 2500 kbps and 1350 kbps.

Furthermore, decreasing the bandwidth abruptly results into RTT and packet loss spikes at minute 2:00 and 4:00 and the round trip time is also proportionally affected when the bandwidth is limited as also seen in section 8-1-4. The framerate momentarily decreases when the available bandwidth is reduced at minute 2 but quickly climbs back up to the original 50 FPS. More interestingly is that the received resolution remains low at 480 x 270 even when more bandwidth becomes available.

8-2-2 Variable packet loss

Even though the effect of different packet losses is already studied in section 8-1-3, it can be interesting to see how packet loss that changes during the lifespan of a call affects the call characteristics. In this section the packet loss is changed every minute from 10%, 0%, 5%, 7.5% and 15% as shown by the dashed blue line in figure 8-5a.



(a) Loss (top) and framerate and resolution (bottom) **(b)** Data rate

Figure 8-5: Actual loss vs set loss, data rate and FPS and resolution for variable packet loss

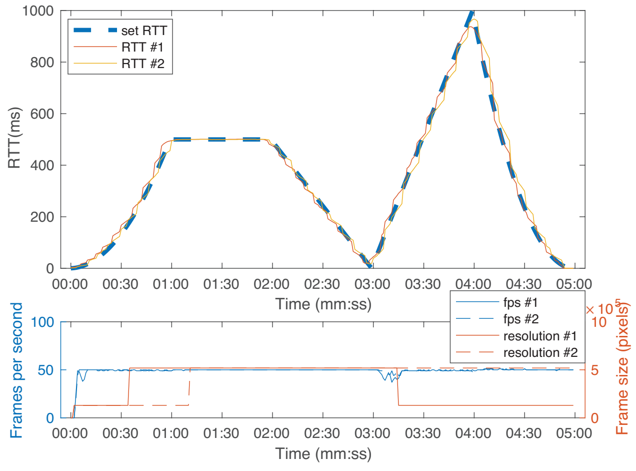
The results are comparable to what is seen in section 8-1-3. A packet loss of 5% and 7.5% only slightly lowers the data rate (minute 2 and 3), where a packet loss >= 10% reduces the data rate significantly. Also interesting to see is that it takes approximately 30 seconds to reach the maximum data rate when packet loss is removed after the first minute, this is in line with our expectations according to the 5% increase in data rate when packet loss is less than 2% (equation 5-11) set by GCC. The data rate increases with 5% every second for 30 seconds, resulting to $550kbps * 1.05^{30} \approx 2400kbps$, which is closed to the reached 2500 kbps shown at minute 1. It takes almost 1 minute to increase the resolution after heavy packet loss (minute 02:00). Decreasing the resolution when packet loss occurs goes significantly faster at around 30 seconds (4:30). The graphs also show that the framerate is only affected when packet loss is >= 10%.

8-2-3 Latency variation

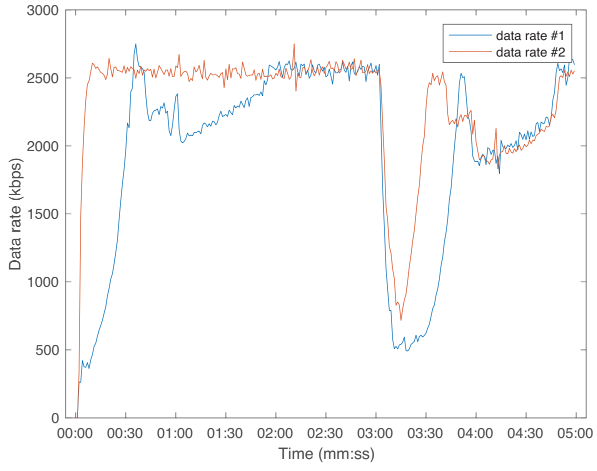
As shown in section 8-1-2 and section 5-1, WebRTC's congestion algorithm does not respond directly to different latencies, but changes its data rate based on latency variation. Instead of changing the latency with a step function as done with the packet loss and bandwidth, the latency is gradually changed at 0.5 seconds intervals in the following experiments with both linear and exponential functions as shown in table 8-5 and the upper graph of figure 8-6a.

Minute	Latency change (from - to)	Steepness
0 - 1	0ms - 250ms	exponential
1 - 2	250ms	N.A.
2 - 3	250ms - 0ms	linear
3 - 4	0ms - 500ms	linear
4 - 5	500ms - 0ms	exponential

Table 8-5: Changing latency sequence



(a) Loss (top) and framerate and resolution (bottom)



(b) Data rate

Figure 8-6: Actual RTT vs set RTT, data rate and FPS and resolution for changing latency

As can be seen, the actual round trip time is really close to the set round trip time. Unlike other tests, the data rate is slightly different for both parties even though additional latency is added in both directions. The data rate for node #2 is high which could be because the network limiter is set on node #1 and the effects on node #2 take longer to set in. As expected, the data rate climbs up to the maximum data rate when latency is decreased (at minute 2 and 4) or kept constant (minute 1).

More unexpectedly, the congestion control algorithm does not seem to kick in until after 40 seconds even though the RTT is increasing exponentially in the first minute. This is presumably due to the earlier mentioned ramp up function as seen in section 8-1-2 which allows WebRTC to reach the maximum data rate more quickly instead of the 5% additive increase set by GCC. It also seems that GCC responds heavily to the RTT transition at minute 3, where a decreasing to increasing RTT results in a massive data rate drop. An odd observation is that the data rate still increases after minute 3 even though the latency also increases.

The received framerate does not seem to be affected by the additional latency but the resolution does suffer under it. The maximum received resolution is 960 x 540 and goes as low as 480 x 270.

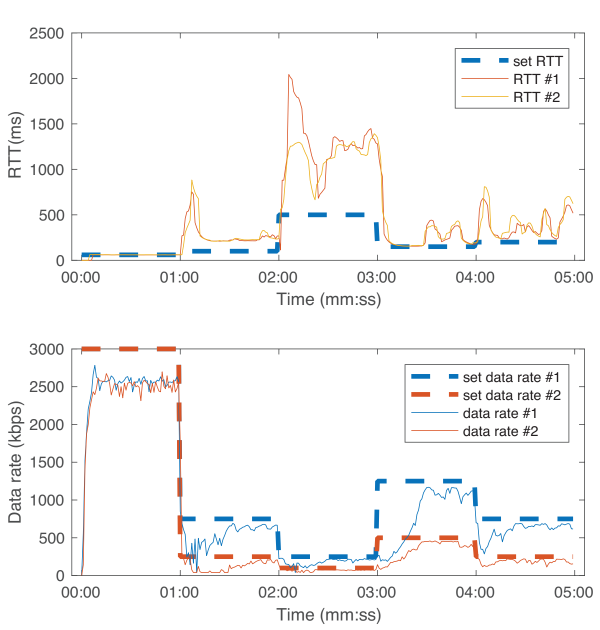
8-2-4 Latency and bandwidth variation

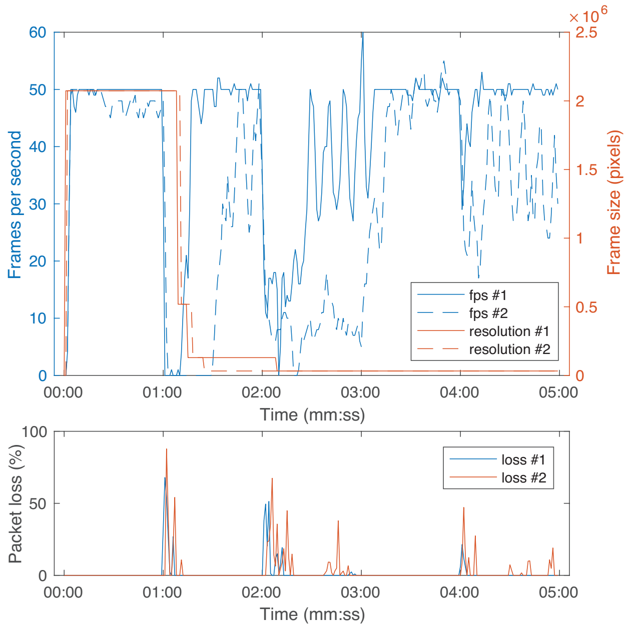
Lastly, the effects of changing both the latency and the available bandwidth are analyzed. Different network characteristics are changed to simulate the effect of handoff which for instance occurs when roaming around. In this section a simulation is made to change between 4G, HDSPA, 3G and Edge network characteristics while analyzing how fast WebRTC adapts to the new network conditions. For this test the available uplink and downlink bandwidth are also limited differently, since it is common for capable uplink rate to be lower than its downlink counterpart. The test procedure is shown in table 8-6.

Minute	**Round trip time**	**Downlink**	**Uplink**
0 - 1	60ms	3Mbps	3Mbps
1 - 2	200ms	750Kbps	250Kbps
2 - 3	500ms	250Kbps	100Kbps
3 - 4	150ms	1250Kbps	500Kbps
4 - 5	200ms	750Kbps	250Kbps

Table 8-6: Changing latency, uplink and downlink bandwidth

The set data rates and latencies together with the resulting values are shown in figure 8-7a. The bandwidth utilization is 69%, which is significantly lower than the 77% bandwidth utilization (figure 8-4) when there is no additional latency. The limited bandwidth also results in additional latency as seen before, especially when the bandwidth is extremely limited (250kbps downlink / 100kbps uplink) at minute 2 when the round trip time increases to more than two times the value it was set. Every time latency is increased or available bandwidth is reduced, packet loss occurs as seen before. As expected, the framerate and resolution received by client #1 are higher than client #2 due to the higher downlink than uplink rates which form the bottleneck at client #1. The resolution furthermore starts at its original 1920 x 1080 but does not get back up from 240 x 135 after minute 2.





(a) RTT (top) and data rate (bottom)

(b) Framerate (top) and Loss (bottom)

Figure 8-7: Actual RTT vs set RTT, actual data rate vs set data rate, FPS and resolution and packet loss when limiting both RTT and data rate

8-3 Multi party

In this section, the performance of several techniques which can be used for multi-party video conferencing are tested as explained in chapter 3. To avoid CPU limitations, this comparison is limited to a maximum of 4 participants. For this experiment, 2 person, 3 person and 4 person video conferencing are compared for different topologies. The meshed topology is first tested in section 8-3-1. An extra server which forwards the streams is then introduced to reduce the throughput in section 8-3-2.

8-3-1 Meshed network

In a meshed network, every participant uploads its stream $n-1$ times and downloads the other $n-1$ streams directly from the other peers. Where n equals the amount of participants. The results for 2p, 3p and 4 person meshed calls are shown in figure 8-8. The rates in this graph apply to both the average uplink and download data rates. The throughput for 3 person calls are close to two times the throughput of 2 person calls (factor 2.03). Surprisingly, 4 person calls have less than 3 times the throughput compared to 2 person calls (factor 2.77), mostly due the long startup delay. The throughput is also very volatile compared to the other calls which maintain a constant data rate even though several 4 person calls are averaged out. This volatile behavior is due to CPU limitations, because every person needs to both encode its own video stream three times and decode the three incoming streams simultaneously.

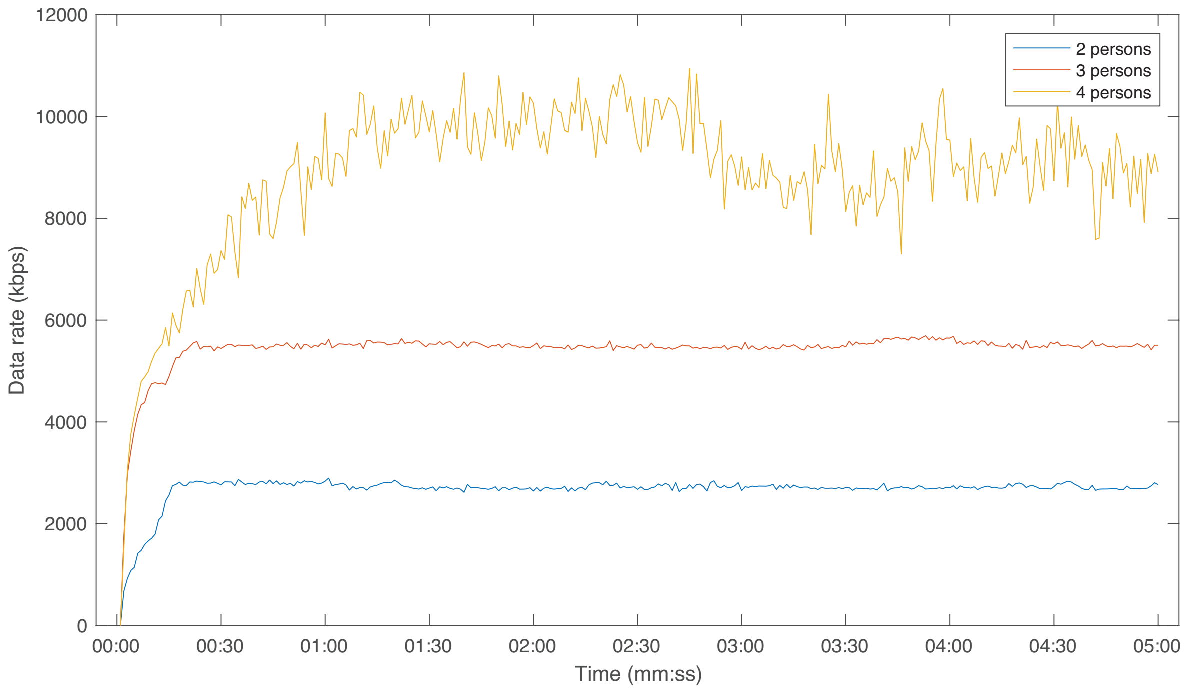


Figure 8-8: Average data rates for 2p, 3p and 4p meshed calls

8-3-2 Selective Forwarding Unit

By introducing an extra server which forwards the streams, the necessary uplink bandwidth can be reduced. If a Selective Forwarding Unit (SFU) is used, all participants only have to upload their stream once and the SFU forwards these to the other participating clients as previously shown in section 3-2-2. This extra server does not only reduce the required uplink bandwidth, but also significantly reduces CPU usage because the stream only has to be encoded once. This approach does however introduce extra latency because streams are relayed. The results with the Janus SFU Gateway are shown in figure 8-9. Compared to meshed calls, it takes significantly longer to reach a stable data rate (30 seconds vs 15 seconds) because Janus manipulates the REMB messages to create an artificial ramp up of the data rate as explained in [61]. With three participants, the average downlink rate is 2.00 times higher than the uplink rate. With four participants, the downlink rate is 2.95 times higher.

8-4 Cross traffic

WebRTC has to compete with cross traffic when there are other TCP/UDP flows active which share the same bottleneck. This is something WebRTC's congestion control algorithm has had troubles with in the past. When concurrent TCP flows were active, WebRTC's UDP streams would starve. As described in section 5-1-3, Google Congestion Control now has an adaptive threshold (γ) which provides better fairness when competing with concurrent flows. According to [32], GCC nowadays competes better with concurrent UDP and TCP flows. In this section this observation is verified. First, a single WebRTC stream competes with other

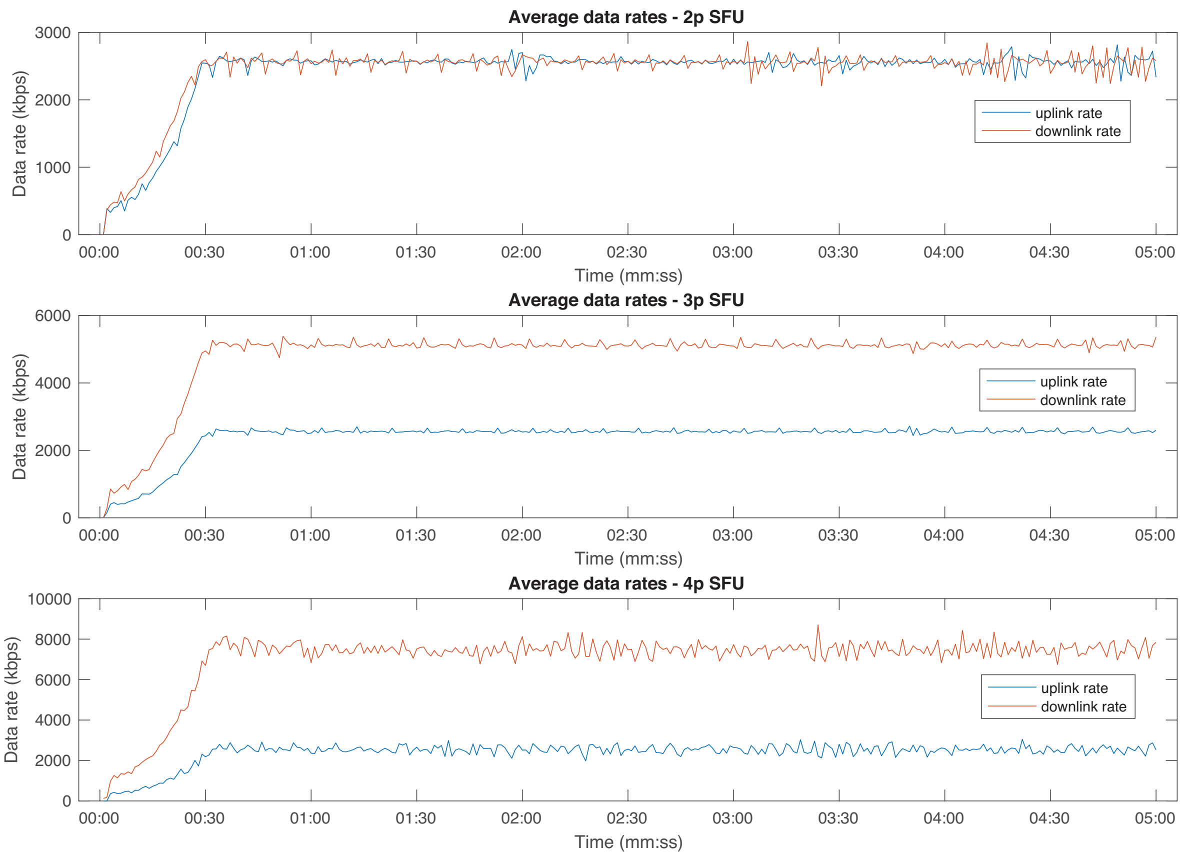


Figure 8-9: Average data rates for 2p, 3p and 4p meshed calls

WebRTC streams while sharing the same bottleneck in section 8-4-1. And in section 8-4-2, the same test is conducted with competing TCP flows.

8-4-1 RTP cross traffic

In this section the available overall bandwidth is limited to $2Mbps$ and the bandwidth distribution is analyzed when three WebRTC calls share this bottleneck. To test how fast the congestion control algorithm adapts, not all calls start at the same time. Instead the experiment starts with one call, a second call is added a minute later and a third call is added a minute after that. To see how fast WebRTC adapts once bandwidth is freed, the third call is dropped in minute 4. The results of this test can be seen in figure 8-10.

The cumulative data rate is 78% which is comparable to the earlier measured bandwidth utilizations (figure 8-3 and figure 8-4). The data rate momentarily drops when a new stream enters or leaves the bottleneck (minute 01:00, 02:00 and 04:00). It does eventually converge to equally distributed data rates, but it does take almost a minute when two streams compete and even longer when 3 streams have to compete. The Jain Fairness Index (equation 6-1) when two streams compete is 0.98, while three streams score 0.94 on Jain's Fairness Index which are both comparable to the findings in [32]. Since both scores are close to 1, fairness is achieved for intra protocol competition.

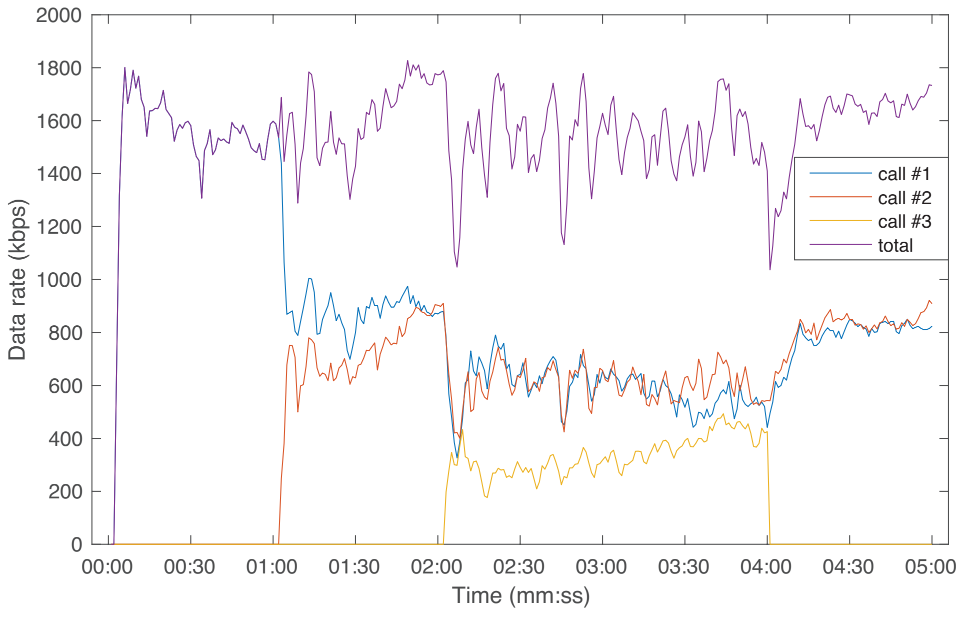


Figure 8-10: Distribution of bandwidth across three WebRTC calls

8-4-2 TCP cross traffic

As explained in section 7-5, *iperf* [54] is used to generate TCP flows to compete with the RTP flows generated by WebRTC. For this experiment the bandwidth is limited to $2Mbps$ and a 12 minute video call between two peers is set up. To test the fairness with TCP cross traffic, a ten minute competing TCP flow is introduced at minute 01:00. The results are shown in figure 8-11.

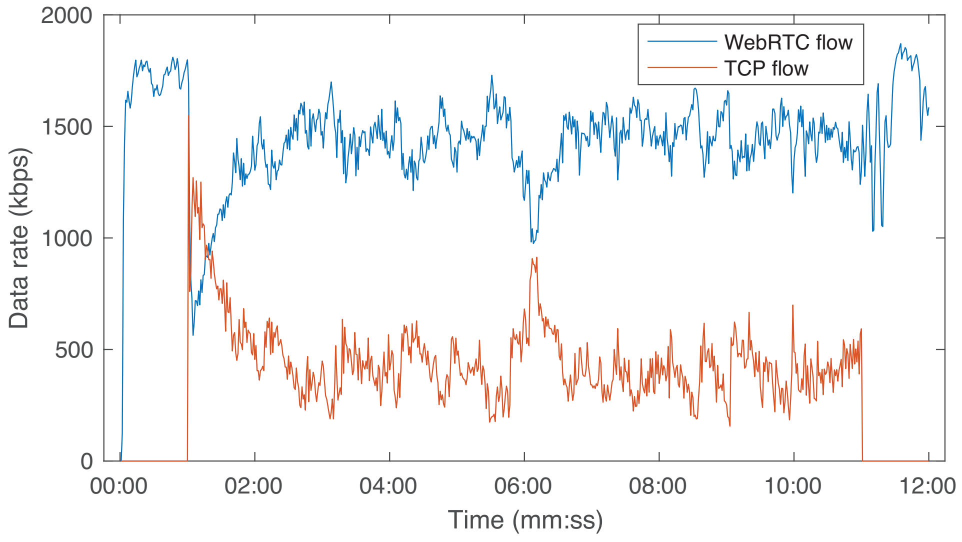


Figure 8-11: Distribution of bandwidth when a single RTP flow competes with a TCP flow

Surprisingly, WebRTC's RTP flow has a significantly higher average data rate from 01:00 - 11:00 compared to the TCP flow (on average 1408kbps vs 451kbps) with a resulting Jain

Fairness Index of 0.79. The newly introduced adaptive threshold does seem to provide better fairness and WebRTC's RTP flows no longer starve when competing with TCP flows. Optimal fairness is however not reached and it does seem that the adaptive threshold is too effective in favor of prioritizing RTP flows.

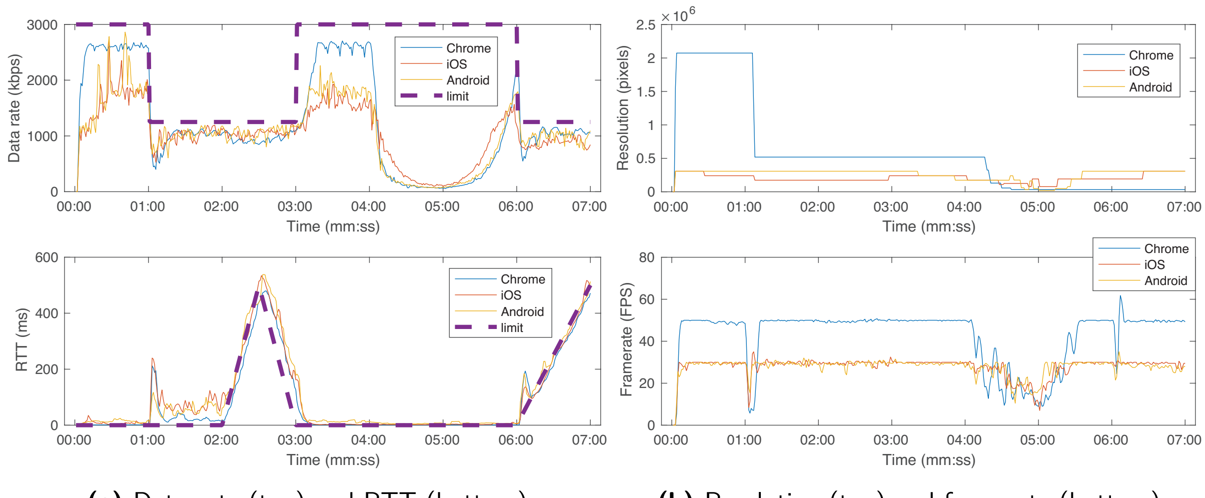
8-5 Mobile performance

For the mobile performance test, a seven-minute test is performed covering all different network variations. Unfortunately, it is not possible to inject a custom video stream for mobile devices so this test falls back on using the camera of the mobile devices as explained in section 7-6. The rear camera is forced to generate a higher quality stream. Unfortunately, Safari does not support WebRTC on iOS, a *Cordova* plugin [56] is therefore used to handle the WebRTC-part natively while using the same codebase as explained in section 7-6-2. The test procedure is shown in table 8-7 where the tests are limited to changing only one parameter each minute.

Minute	Round trip time	Data rate	Packet loss
0 - 1	0ms	∞	0%
1 - 2	0ms	1250Kbps	0%
2 - 3	0ms - 500ms - 0ms	1250Kbps	0%
3 - 4	0ms	∞	0%
4 - 5	0ms	∞	15%
5 - 6	0ms	∞	0%
6 - 7	0ms - 500ms	1250Kbps	0%

Table 8-7: Mobile performance test procedure

The different call characteristics of the test described in table 8-7 are shown in figure 8-12 and table 8-8.



(a) Data rate (top) and RTT (bottom) **(b)** Resolution (top) and framerate (bottom)

Figure 8-12: Call characteristics for iOS, Android and Chrome network limited by table 8-7

Interesting to see, is that iOS and Android seem to be limited to a resolution of 640x480 at 30FPS, even though their rear cameras are able to handle higher resolutions. Furthermore, iOS and Android behave really similar across all characteristics. Their maximum data rates are both significantly less than Chrome when there is no congestion (1750kbps vs 2500kbps) and their average RTT is much higher. It also takes longer for both mobile platforms to reach the maximum data rate when compared to Chrome (20 seconds vs 10 seconds). Even though the results show that mobile devices are inferior compared to non-mobile devices, there are barely differences in how Chrome's video data is received (columns iOS, Android and Chrome 2). The only difference is a significant increase in round-trip-time for the mobile scenarios.

	iOS		**Android**		**Chrome**	
	Chrome	**iOS**	**Chrome**	**Android**	**Chrome 1**	**Chrome 2**
Data rate (kbps)	1022,5	1302,2	1047,3	1220,5	1226,7	1248,8
RTT (ms)	95,4	105,4	100,0	108,4	79,9	80,0
Packet loss(%)	2,18%	2,57%	2,20%	2,26%	2,27%	2,32%
Resolution (px)	602 x 339	810 x 456	675 x 380	1001 x 563	1008 x 567	1004 x 565
Framerate (FPS)	27,86	44,98	27,75	42,61	42,52	42,96

Table 8-8: Average receiving characteristics during 7 minute session

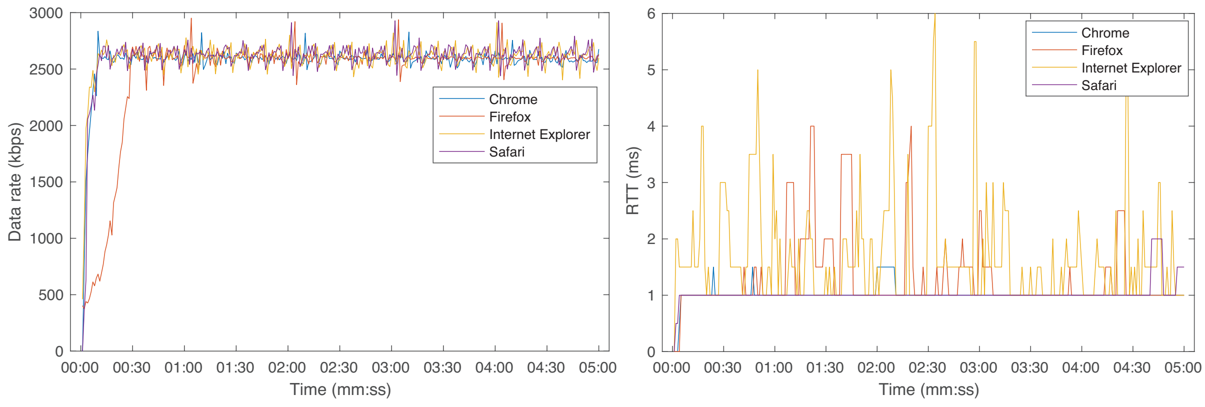
8-6 Cross browser

As explained in section 4-3, different browsers support different implementations of WebRTC. While some browsers support WebRTC natively, other browsers such as Internet Explorer and Safari need an external plugin to perform WebRTC calls which can lead to a different resulted performance. To generate comparable results, one of the clients uses Google Chrome which has the ability to inject the custom 1080p video stream (section 7-2-1). For the other client, Google Chrome, Mozilla Firefox, Internet Explorer and Safari are used and the way they receive and decode the injected video stream is analyzed. To support both Internet Explorer and Safari, the AdapterJS plugin is used (section 4-3-1) which unfortunately does not provide the received framerate and resolution. Since Microsoft Edge and Google Chrome currently do not support a mutual video codec, Microsoft Edge is not tested. Results for a standard five minute video chat on the local setup are shown in figure 8-13 and table 8-9.

	Chrome	**Firefox**	**Internet Explorer**	**Safari**
Data rate (kbps)	2568.5	2456.5	2605.5	2597.5
RTT (ms)	1.0	1.3	1.7	1.0
Framerate (FPS)	50.1	48.9	N.A.	N.A.
Packet loss (%)	0.00	0.00	0.00	0.00
Resolution (pixels)	1914 x 1076	973 x 547	N.A.	N.A.

Table 8-9: Average call characteristics for different browsers

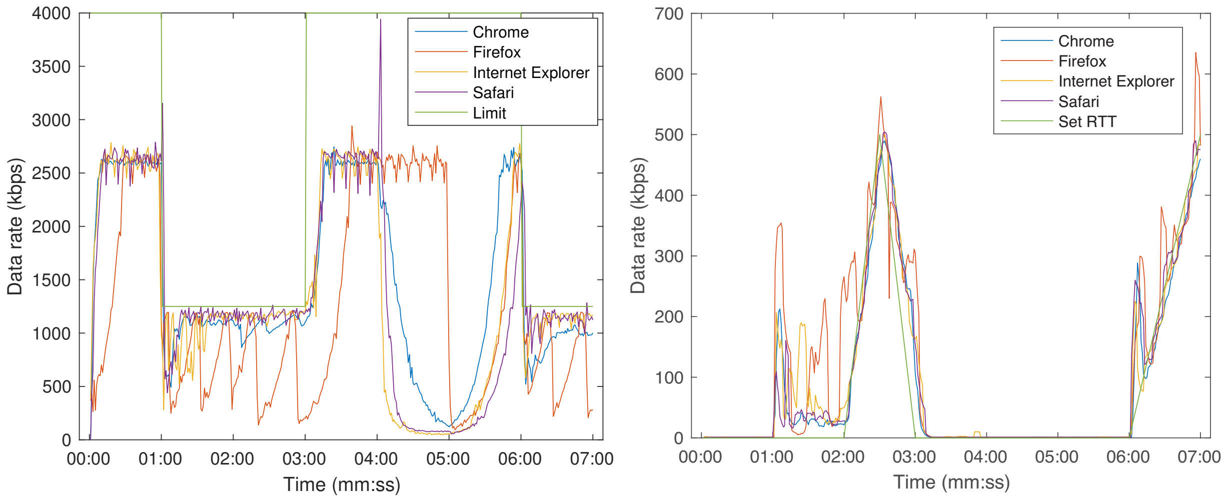
As seen in figure 8-13a, Firefox takes significantly longer to reach the maximum data rate compared to the other browsers (30 seconds vs 10 seconds). Firefox is also not able to receive the original resolution and does not get beyond 560p while the original video is sent at 1080p.



(a) Data rate **(b)** Round-trip-time

Figure 8-13: Call characteristics for different browsers

Internet Explorer and Safari do not have data available for the framerate and resolution. However, when looking subjectively it can be seen that the framerate is unacceptable (< 10FPS) for both browsers using the AdapterJS plugin to support WebRTC which is most likely due to the video decoder. Both unsupported browsers furthermore seem to handle the data transfers like the other browsers. Internet Explorer does experience slightly higher round-trip-time than other browsers as shown in figure 8-13b. To test the performance with extra network effects, the same seven minute test procedure as shown in table 8-7 is used. The results are shown below in figure 8-14 and table 8-10.



(a) Data rate **(b)** Round-trip-time

Figure 8-14: Call characteristics for different browsers subject to network effects

The limited bandwidth from minute 1 to 3 and after minute 6 results in a choppy data rate with Firefox compared to other browsers (figure 8-14a). Firefox also show a slow data rate ramp up after minute 3 as seen before and higher RTTs (figure 8-14b). Interesting to see is that packet loss does not affect Firefox's data rate at all (minute 3) which could be because of

	Chrome	Firefox	Internet Explorer	Safari
Data rate (kbps)	1439.7	1277.3	1321.7	1334.7
RTT (ms)	83.6	109.9	91.1	88.9
Framerate (FPS)	47.4	45.1	N.A.	N.A.
Packet loss (%)	2.51	7.18	2.16	1.96
Resolution (pixels)	858 x 483	539 x 303	N.A.	N.A.

Table 8-10: Average call characteristics for different browsers subject to network effects

a communication error with the RTCP reports which do not trigger a rate change. Chrome adapts the fastest to different network effects as seen at minute 4 - 6. Chrome overall does seem to be superior. Even though Internet Explorer and Safari seem to be able to handle the data, the framerate decoded by the AdapterJS plugin is unacceptable for the full HD stream. Firefox does not respond to packet loss, fluctuating data rates and higher RTTs, but possibly performs better when Firefox is tested against Firefox.

8-7 Video codecs

By default, Google Chrome uses the VP8 video codec for its WebRTC video calls. Since Google Chrome v48, support has been added for its successor: VP9. VP9 is supposed to provide the same objective quality as VP8 at a lower bitrate due to its more efficient compression efficiency [62] as also explained in section 4-2. This does however come at the expense of extra CPU power. VP9 can therefore be useful when bandwidth is limited (on e.g. cellular and congested networks) or when you want the same quality at a lower bitrate. Support for the H.264 video codec has also been added in Chrome v50, which allows hardware-accelerated decoding for many devices. Even though these newly supported codecs are not used by default, WebRTC can be forced to use them by altering the generated Session Description Protocol (SDP) as explained in 7-2-2. In this section the VP8, VP9 and H.264 video codecs are compared. Because these newly added codecs are still under heavy development, a 720p variant of the mediastream is used to prevent CPU limitations while encoding and decoding. First a five-minute test is conducted without network constraints as shown in figure 8-15.

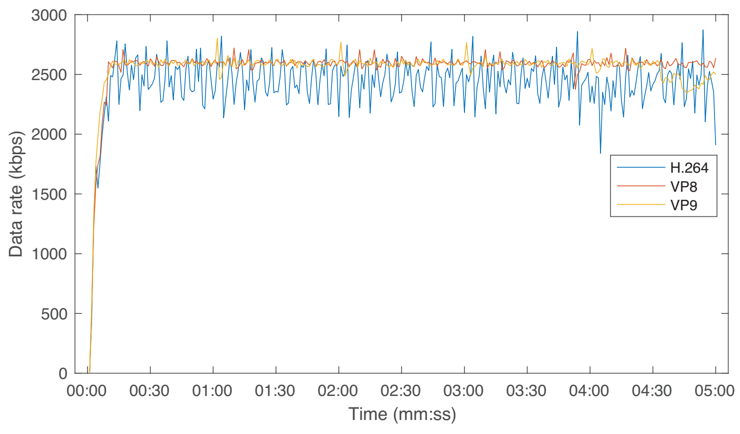
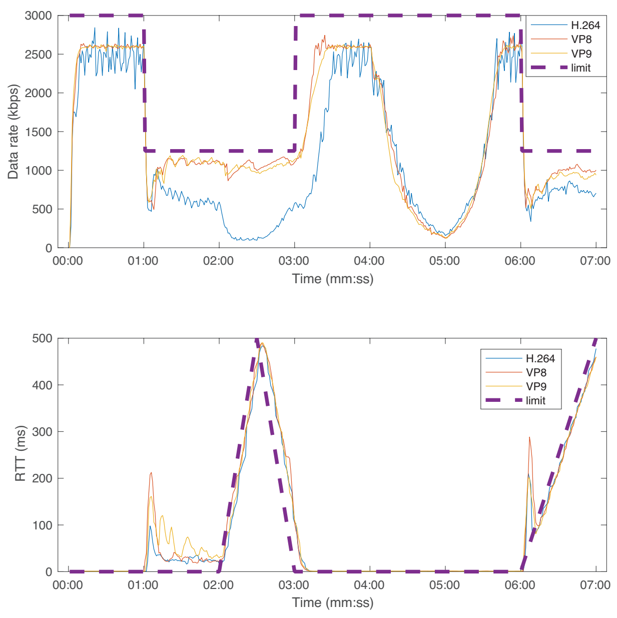
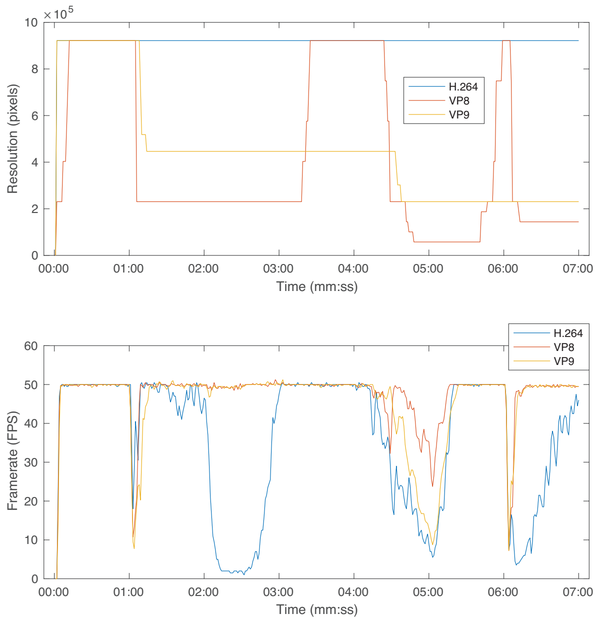


Figure 8-15: Average data rates for different video codecs

This shows VP9 behaves similar to VP8, while H.264 is unable to keep a constant data rate. All other call characteristics are similar; the original resolution and framerate are quickly reached, RTT is low and no packet loss is observed. More interesting is to see how the different video codecs behave under the seven minute test described in table 8-7. The results are shown in figure 8-16 and table 8-11.



(a) Data rate (top) and RTT (bottom)



(b) Resolution (top) and framerate (bottom)

Figure 8-16: Average call characteristics for the VP8, VP9 and H264 video codecs

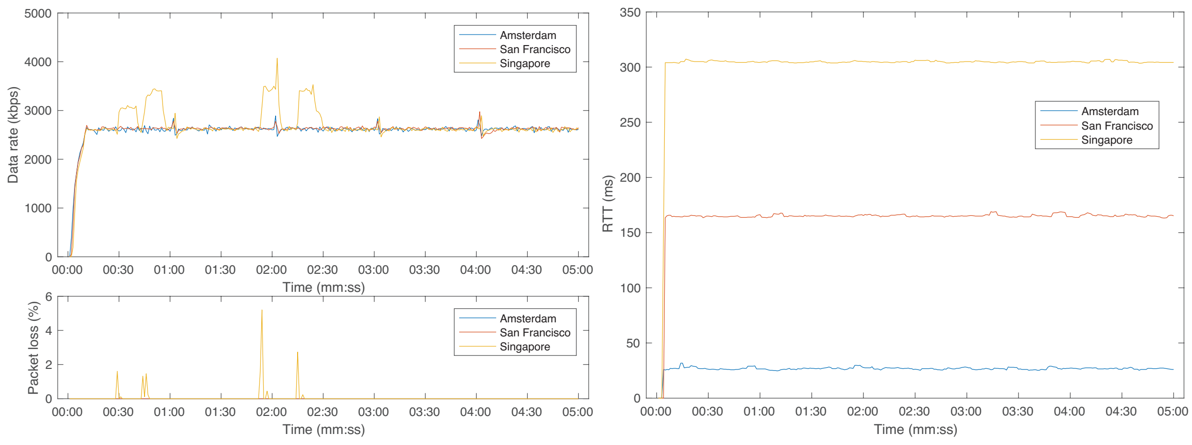
	VP8	**VP9**	**H.264**
Data rate (kbps)	1439.7	1422.5	1154.2
RTT (ms)	83.6	84.2	77.6
Framerate (FPS)	47.4	45.5	36.1
Packet loss (%)	2.51	2.44	2.43
Resolution (pixels)	858 x 483	892 x 502	1279 x 719

Table 8-11: Average call characteristics for different video codecs

Based on the data rate, it is immediately clear that H.264 is more heavily affected by the limited bandwidth and added latency when compared to VP8 and VP9. H.264 also differs in the way it uses its data rate. Where VP8 and VP9 balance out between framerate and resolution, H.264 only lowers its framerate while maintaining a constant maximum resolution. This causes the framerate to drop to an unacceptable 1 FPS around minute 2:30. Like expected, VP9 outperforms VP8 when congestion occurs due to its more efficient compression capabilities. This can be seen from the higher resolutions from minute 1 to 3. When congestion is removed at minute 3 or minute 5, VP9 however does not scale back up to the original resolution (1280x720) while VP8 does.

8-8 Multi-domain testing

All of the conducted tests in the previous sections are run on a local network with simulated network conditions, without these RTP streams being subject to network effects introduced by the internet. In this section the accuracy of this method is verified. To allow multi-domain testing, a couple of virtual machines are launched in different locations around the world as explained in section 7-7. The host is situated in Utrecht, The Netherlands and the different virtual machines are located in Amsterdam (The Netherlands), San Francisco (USA) and Singapore. WebRTC video calls are made between the host and these respective virtual machines. To avoid CPU limitations, the 720p variant of the injected video stream is used. First, the characteristics of a five minute call without network limitations are analyzed as shown in figure 8-17 and table 8-12.



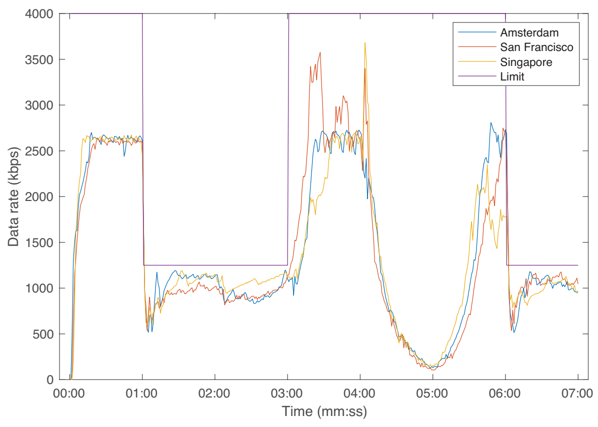
(a) Data rate (top) and packet loss (bottom) **(b)** Round-trip-time

Figure 8-17: Call characteristics for a 5 minute call between Utrecht and other locations

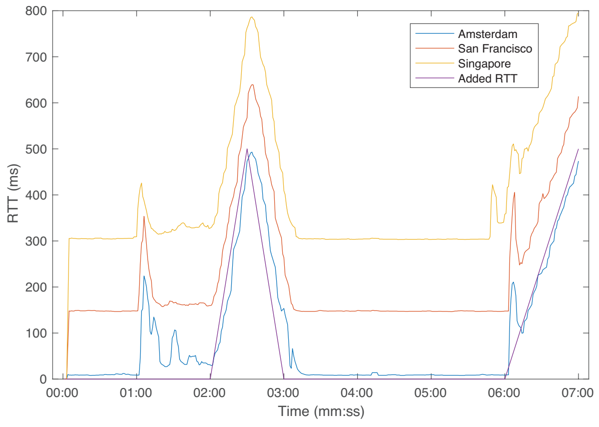
	Amsterdam	**San Francisco**	**Singapore**
Data rate (kbps)	2583.5	2590.5	2675.5
RTT (ms)	26.6 ± 2.9	163.0 ± 19.0	301.2 ± 31.6
Framerate (FPS)	50.3	50.2	50.1
Packet loss (%)	0.00	0.00	0.05
Resolution (pixels)	1278 x 719	1277 x 718	1275 x 717

Table 8-12: Average call characteristics for different locations

The round-trip-time remains constant during the call as can be seen from the graphs and the table even when connecting to the other side of the world. The long distance video call with Singapore however did experience occurrences of packet loss which directly result into an increase in data rate (figure 8-17a) which is due to retransmissions. The additional latency does not trigger the congestion control algorithm (and thus affect the data rate) as expected, since the latency variation is minimal. Next, several network effects are added to the long distance call. For this test the same seven minute sequence as shown in table 8-7 is used. Graphs and details for this test are shown in figure 8-18 and table 8-13.



(a) Data rate



(b) Round-trip-time

Figure 8-18: Call characteristics for a multi-domain call subject to different network effects

	Amsterdam	**San Francisco**	**Singapore**
Data rate (kbps)	1403.4	1388.3	1393.0
RTT (ms)	91.6	226.3	390.8
Framerate (FPS)	48.1	48.1	47.9
Packet loss (%)	2.49	2.53	2.52
Resolution (pixels)	746 x 420	721 x 405	735 x 414

Table 8-13: Average call characteristics for different locations while subject to network effects

From the table and figure shown above it can be stated that additional latency for distant locations do not amplify the effect of network effects. The call characteristics of the short distanced call with Amsterdam shows marginal improvements with a slightly better data rate, packet loss and resolution (table 8-13). When looking at the data rate, the short distant call does show less volatile behavior while the other calls show lower bandwidth and/or retransmissions (figure 8-18a). These tests do however verify that the approach used by simulating different network effects is representative of what happens while conducting calls over the internet.

Chapter 9

Conclusion

In this thesis several different scenarios for WebRTC-based video conferencing are analyzed. WebRTC has evolved greatly since its introduction in 2012. While the support was really limited in 2012 with only 2%, WebRTC is now available on 70% of the used browsers in 2016. The ongoing collaboration between W3C and IETF to define a unified standard for all browsers also paid off by forcing all browsers to support the same set of media codecs. WebRTC is currently available on all platforms and browsers, even though some browsers require additional plugins for WebRTC to work and Apple iPhones require WebRTC to be embedded in a mobile app.

WebRTC uses the Google Congestion Control algorithm on top of the otherwise unreliable UDP protocol to adjust the data rate when congestion occurs. GCC comprises of two systems: a receiver side controller which adjusts its data rate based on the inter-arrival time of the packets received, and a sender side controller which changes its rate based on the amount of packets lost. These subsystems communicate via RTP's media packets and RTCP's feedback packets and adjust the data rate accordingly.

An experimental test bed is set up to analyze WebRTC's performance. High end computer have been used to prevent CPU limitations and Google Chrome is used to inject a representative 1080p video stream for fair comparison. To test WebRTC's performance with different network characteristics, Dummynet is used to simulate packet loss, latency and limit the available uplink and downlink bandwidth. WebRTC's RTCPeerConnection.getStats() function is exploited to generate all statistics needed for the performance analysis.

A series of experiments show that WebRTC adjusts the data rate based on different network effects as expected; capping the available bandwidth results in about 75% - 80% of bandwidth usage, when experiencing packet loss the sender side controller adjusts the bandwidth depending on how many packets are lost as explained in equation 5-11 and an increase in latency between both parties results in a decreasing data rate. At the same reduced bandwidth, the framerate is more heavily affected than the resolution of the video stream when packet loss occurs compared to when the bandwidth is capped.

For WebRTC video conferences with more than 3 people, an SFU is suggested to reduce the CPU usage and required uplink bandwidth when compared to a full meshed topology. An

SFU does increase the latency and it is therefore recommended to provide multiple SFUs in different locations around the world when video chats are performed globally to minimize the latency as much as possible. When WebRTC competes with cross traffic, the competing streams are more equally distributed due to the newly introduced adaptive threshold for GCC. Fairness is achieved for intra-protocol competition (i.e. multiple WebRTC streams). The fairness for inter-protocol competition (WebRTC with competing TCP flows) has also improved, but still has room for further improvement since WebRTC takes almost 75% pf the available bandwidth.

For mobile devices, WebRTC limits the data rates and resolutions of the sent video streams resulting in lower quality streams. Both Android and iOS perform similarly with slightly increased round-trip-times compared to Chrome. Even though every browser supports WebRTC, Google Chrome performs far more superior than other browsers. Firefox adapts slower to network effects and the plugin-based Internet Explorer and Safari are unable to produce an acceptable framerate when decoding the 1080p video. When different video codecs are compared, H.264 needs to improve and provide a better tradeoff between resolution and framerate when congestion occurs.

9-1 Recommendations

For future research, several recommendations can be made. To narrow down the scope of this work, all performance tests ensured that the data rate was not degraded due to CPU limitations by performing all tests on high end hardware. When a CPU limitation is reached, the RTCStatsReport reports a true value for googCpuLimited which was prevented during the performance analysis. For some tests (e.g. the video codec comparison), the 1080p video stream was too heavy and a lighter 720p stream was chosen instead. For future research, it is however recommended to also investigate the CPU limitations and/or energy consumption of WebRTC. WebRTC currently detects if CPU capacity is exhausted and scales down the video quality accordingly. New research could shed light on what side effects occur during this process. Energy consumption is also a crucial factor on mobile devices and an energy consumption comparison with other video conferencing platforms will be valuable.

As mentioned earlier, the adaptive threshold to provide fairness with concurrent TCP flows is not ideal. For future research this finding should be investigated further. With the provided GCC equations in chapter 5, congestion control simulations can be made while tweaking the adaptive threshold (γ) to explore possible better values which provide better fairness when competing with concurrent TCP flows. These values can then be changed in the source code of Google Chrome, recompiled, and new cross traffic tetsts can be run to confirm if better fairness is reached.

For future research, it can also be interesting to conduct tests using realistic mobile network conditions. Even though several multi-domain tests have been performed with nodes across the world, the effects of a lossy WiFi link or a cellular have not been tested. WebRTC is known to underutilize the bandwidth on a lossy wireless link as shown in [50] where packet grouping is suggested to minimize this effect. WebRTC has also not been tested on cellular connections which have limited (uplink) bandwidth and greater latencies. Also mobile handoff (e.g. switching from 4G to 3G) occurs when roaming around with a cellular connection which could impact the performance of an ongoing WebRTC conference.

A last recommendation for future research is to more thoroughly analyze WebRTC's performance for different internet browsers. In section 8-6 one client uses Google Chrome, which is then subsequently conducts WebRTC calls with other internet browsers. From these results, the Google Chrome with Google Chrome video call performed far superior with respect to the other browsers. What is however not tested, is how e.g. a Firefox with Firefox WebRTC call performs, because one client uses Google Chrome for all tests.

List of Figures

List of Tables

Bibliography

[1] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba. (2016, Jan.) Webrtc 1.0: Real-time communication between browsers. [Online]. Available: http://www.w3.org/TR/webrtc/

[2] T. Hardie, C. Jennings, and S. Turner. (2012, Oct.) Real-time communication in web-browsers. [Online]. Available: https://tools.ietf.org/wg/rtcweb/

[3] P. Rodriguez, J. C. Arriba, I. Trajkovska, and J. Salvachua, "Advanced videoconferencing services based on webrtc," in *Proceeding of IADIS multi conference on computer science and information systems*, 2013, pp. 1–8.

[4] R. Dillet. (2016, Jun.) Safari puts yet another nail in flash's coffin. [Online]. Available: https://techcrunch.com/2016/06/15/safari-puts-yet-another-nail-in-flashs-coffin-again/

[5] H. Alvestrand. (2011, May) Google release of webrtc source code. [Online]. Available: https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html

[6] S. Holmer, H. Lundin, G. Carlucci, L. D. Cicco, and S. Mascolo. (2015, Oct.) A google congestion control algorithm for real-time communication. [Online]. Available: https://tools.ietf.org/html/draft-ietf-rmcat-gcc-01

[7] D. C. Perkins, M. J. Handley, and V. Jacobson, "SDP: Session Description Protocol," IETF RFC 4566, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc4566.txt

[8] J. Uberti, Google, C. Jennings, Cisco, and E. E. Rescorla. (2016, Mar.) Javascript session establishment protocol. [Online]. Available: https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-14

[9] M. Westerlund and C. Perkins. (2011, Jul.) Iana registry for interactive connectivity establishment (ice) options. [Online]. Available: https://tools.ietf.org/html/rfc6336

[10] S. Loreta and S. P. Romano, *Real-Time Communication with WebRTC*. O'Reilly Media, 2014.

[11] M. Petit-Huguenin and G. Salgueiro, “Datagram Transport Layer Security (DTLS) as Transport for Session Traversal Utilities for NAT (STUN),” RFC 7350, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7350.txt

[12] P. Matthews, J. Rosenberg, and R. Mahy, “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),” RFC 5766, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc5766.txt

[13] WebRTC Statistics. (2014, Jun.) Webrtc revolution in progress. [Online]. Available: http://webrtcstats.com/webrtc-revolution-in-progress/

[14] E. Rescorla and N. Modadugu, “Datagram Transport Layer Security Version 1.2,” RFC 6347, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc6347.txt

[15] M. Baugher, “The Secure Real-time Transport Protocol (SRTP),” RFC 3711, Mar. 2013. [Online]. Available: https://rfc-editor.org/rfc/rfc3711.txt

[16] R. R. Stewart, “Stream Control Transmission Protocol,” RFC 4960, Oct. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc4960.txt

[17] I. Grigorik, *High performance browser networking.* O’Reilly Media, 2013.

[18] Mozilla Developer Network. (2016, Jul.) Mediadevices.getusermedia(). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

[19] ——. (2016, Jul.) Rtcpeerconnection. [Online]. Available: https://developer.mozilla.org/en/docs/Web/API/RTCPeerConnection

[20] T. Levent-Levi, “Webrtc multiparty video alternatives, and why sfu is the winning model,” July 2016. [Online]. Available: https://bloggeek.me/webrtc-multiparty-video-alternatives/

[21] M. Khan, “Rtcmultipeerconnection,” July 2016. [Online]. Available: https://github.com/muaz-khan/RTCMultiConnection

[22] P. Hancke. (2016, May) Is webrtc ready yet? [Online]. Available: http://iswebrtcreadyyet.com

[23] NetMarketShare. (2016) Market share statistics for internet technologies. [Online]. Available: http://netmarketshare.com

[24] A. Roach. (2014, Nov.) [rtcweb] mti video codec: a novel proposal. [Online]. Available: http://www.ietf.org/mail-archive/web/rtcweb/current/msg13432.html

[25] Microsoft Edge Team. (2016, Apr.) Roadmap update for real time communications in microsoft edge. [Online]. Available: https://blogs.windows.com/msedgedev/2016/04/13/roadmap-update-for-real-time-communications-in-microsoft-edge/

[26] ——. (2016, Apr.) Webm, vp9 and opus support in microsoft edge. [Online]. Available: https://blogs.windows.com/msedgedev/2016/04/18/webm-vp9-and-opus-support-in-microsoft-edge/

[27] Temasys Communications Pte Ltd. Temasys webrtc plugin. [Online]. Available: http://skylink.io/plugin/

[28] Microsoft Edge Team. (2015, Sep.) Ortc api is now available in microsoft edge. [Online]. Available: https://blogs.windows.com/msedgedev/2015/09/18/ortc-api-is-now-available-in-microsoft-edge/

[29] A. Zmora. (2016, Jan.) Ortc vs. webrtc: That's not the right question. [Online]. Available: https://thenewdialtone.com/ortc-webrtc/

[30] The WebRTC Project, "Webrtc is a free, open project that provides browsers and mobile applications with real-time communications (rtc) capabilities via simple apis." July 2016. [Online]. Available: https://webrtc.org/

[31] R. Bos, X. Bombois, and P. M.J. Van den Hof, "Designing a kalman filter when no noise covariance information is available." in *Proceedings of the 16th IFAC World Congress*, ser. IFAC '05, 2005, pp. 212–212.

[32] L. D. Cicco, G. Carlucci, S. Holmer, and S. Mascolo, "Analysis and design of the google congestion control for web real-time communication (webrtc)," in *Proc. ACM Mmsys 2016, Klagenfurt, Austria, May 2016*, 2016.

[33] L. D. Cicco, G. Carlucci, and S. Mascolo, "Understanding the dynamic behaviour of the google congestion control for rtcweb," in *20th International Packet Video Workshop, PV 2013, San Jose, CA, USA, December 12-13, 2013*, 2013, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/PV.2013.6691458

[34] G. Carlucci, L. De Cicco, and S. Mascolo, "Modelling and control for web real-time communication," in *Proc. of IEEE 52nd Conference on Decision and Control (CDC)*. IEEE, 2014.

[35] H. Alvestrand. (2014, Apr.) Rtcp message for receiver estimated maximum bitrate. [Online]. Available: https://tools.ietf.org/html/draft-alvestrand-rmcat-remb-03

[36] V. Singh, A. A. Lozano, and J. Ott, "Performance analysis of receive-side real-time congestion control for webrtc." in *PV*. IEEE, 2013, pp. 1–8. [Online]. Available: http://dblp.uni-trier.de/db/conf/pv/pv2013.html#SinghLO13

[37] A. Hartaman, B. Rahmat, and Istikmal, "Performance and fairness analysis (using jain's index) of aodv and dsdv based on aco in manets," in *2015 4th International Conference on Interactive Digital Media (ICIDM) ,December 1-5, 2015 Bandung - Indonesia*, 2015, pp. 1–7.

[38] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "Performance analysis of the janus webrtc gateway," in *Proceedings of the 1st Workshop on All-Web Real-Time Systems*, ser. AWeS '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:7. [Online]. Available: http://doi.acm.org/10.1145/2749215.2749223

[39] D. Vucic and L. Skorin-Kapov, "The impact of mobile device factors on qoe for multi-party video conferencing via webrtc," in *Telecommunications (ConTEL), 2015 13th International Conference on*, July 2015, pp. 1–8.

[40] F. Fund, C. Wang, Y. Liu, T. Korakis, M. Zink, and S. S. Panwar, "Performance of dash and webrtc video services for mobile users," in *2013 20th International Packet Video Workshop*, Dec 2013, pp. 1–8.

[41] G. Carullo, M. Tambasco, M. Di, Mauro, and M. Longo, "A performance evaluation of webrtc over lte," in *2016 12th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, 2016, pp. 170–175.

[42] S. Taheri, L. A. Beni, A. V. Veidenbaum, A. Nicolau, R. Cammarota, J. Qiu, Q. Lu, and M. R. Haghighat, "Webrtcbench: a benchmark for performance assessment of webrtc implementations." in *ESTImedia 2015*, 2015, pp. 1–7.

[43] A. Heikkinen, T. Koskela, and M. Ylianttila, "Performance evaluation of distributed data delivery on mobile devices using webrtc." in *IWCMC*. IEEE, 2015, pp. 1036–1042. [Online]. Available: http://dblp.uni-trier.de/db/conf/iwcmc/iwcmc2015.html#HeikkinenKY15

[44] ITU-T Recommendation G.100/P.10, "New Appendix I - Definition of Quality of Experience (QoE)," International Telecommunication Union, Geneva, Switzerland, Jan. 2007.

[45] ITU-T Recommendation P.910, "Subjective video quality assessment methods for multimedia applications," International Telecommunication Union, Geneva, Switzerland, Apr. 2008.

[46] Y. Lu, Y. Zhao, F. Kuipers, and P. Van Mieghem, *Measurement Study of Multi-party Video Conferencing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 96–108. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12963-6_8

[47] F. Kuipers, R. Kooij, D. De Vleeschauwer, and K. Brunnström, "Techniques for measuring quality of experience," in *Proceedings of the 8th International Conference on Wired/Wireless Internet Communications*, ser. WWIC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 216–227. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13315-2_18

[48] L. Ma, W. Chen, D. Veer, G. Sternberg, W. Liu, and Y. A. Reznik, "Early packet loss feedback for webrtc-based mobile video telephony over wi-fi." in *GLOBECOM*. IEEE, 2015, pp. 1–6. [Online]. Available: http://dblp.uni-trier.de/db/conf/globecom/globecom2015.html#MaCVSLR15

[49] R. Atwah, R. Iqbal, S. Shirmohammadi, and A. Javadtalab, "A dynamic alpha congestion controller for webrtc," in *2015 IEEE International Symposium on Multimedia, ISM 2015, Miami, FL, USA, December 14-16, 2015*, 2015, pp. 132–135. [Online]. Available: http://dx.doi.org/10.1109/ISM.2015.63

[50] G. Carlucci, L. De Cicco, and S. Mascolo, "Making google congestion control robust over wi-fi networks using packet grouping," in *Applied Networking Research Workshop 2016*, 2016.

[51] M. Carbone and L. Rizzo, "Dummynet revisited," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, pp. 12–20, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1764873.1764876

[52] W3C, "Identifiers for webrtc's statistics api," May 2016. [Online]. Available: http://www.w3.org/TR/webrtc-stats/

[53] M. Khan, "getstats.js," February 2016. [Online]. Available: https://github.com/muaz-khan/getStats

[54] ESNet. (2016, Jul.) iperf3: A tcp, udp, and sctp network bandwidth measurement tool. [Online]. Available: https://github.com/esnet/iperf

[55] IDC. (2015, Aug.) Smartphone os market share, 2015 q2. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[56] I. B. Castillo, "Cordova ios plugin exposing the full webrtc w3c javascript apis," July 2016. [Online]. Available: https://github.com/eface2face/cordova-plugin-iosrtc

[57] M. Parker. (2015, Oct.) Samsung galaxy s6 vs iphone 6: Which is best? [Online]. Available: http://www.trustedreviews.com/opinions/samsung-galaxy-s6-vs-iphone-6

[58] Microsoft Azure, "Virtual machines - create linux and windows virtual machines in minutes," July 2016. [Online]. Available: https://azure.microsoft.com/en-us/services/virtual-machines/

[59] S. Abbas, M. Mosbah, A. Zemmari, and U. Bordeaux, "Itu-t recommendation g.114, one way transmission time," in *In International Conference on Dynamics in Logistics 2007 (LDIC 2007), Lect. Notes in Comp. Sciences.* Springer-Verlag, 1996.

[60] L. De Cicco, G. Carlucci, and S. Mascolo, "Experimental investigation of the google congestion control for real-time flows," in *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, ser. FhMN '13. New York, NY, USA: ACM, 2013, pp. 21–26. [Online]. Available: http://doi.acm.org/10.1145/2491172.2491182

[61] L. Miniero, "Congestion control (gcc) in janus," July 2016. [Online]. Available: https://groups.google.com/forum/#!topic/meetecho-janus/G8L8N2qWQDo

[62] D. Mukherjee, J. Bankoski, A. Grange, J. Han, J. Koleszar, P. Wilkins, Y. Xu, and R. Bultje, "The latest open-source video codec vp9 - an overview and preliminary results." in *PCS*. IEEE, 2013, pp. 390–393. [Online]. Available: http://dblp.uni-trier.de/db/conf/pcs/pcs2013.html#MukherjeeBGHKWXB13

Glossary

List of Acronyms

API	Application Programing Interface
DTLS	Datagram Transport Layer Security
EPLF	Early Packet Loss Feedback
GCC	Google Congestion Control
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
JSEP	JavaScript Session Establishment Protocol
MCU	Multipoint Control Unit
MOS	Mean Opinion Score
MTU	Maximum Transmission Unit
NAT	Network Address Translation
ORTC	Object Real-Time Communication
P2P	Peer-to-Peer
QoE	Quality of Experience
REMB	Receiver Estimated Maximum Bandwidth

RTC	Real-Time Communication
RTCP	Real-Time Control Protocol
RTP	Real-Time Transport Protocol
S/C	Server-to-Client
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
SFU	Selective Forwarding Unit
SRTCP	Secure Real-time Control Transport Protocol
SRTP	Secure Real-Time Transport Protocol
STUN	Session Traversal Utilities for NAT
SVC	Scalable Video Coding
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
VC	Video Conferencing
VOIP	Voice Over Internet Protocol
W3C	World Wide Web Consortium
WebRTC	Web Real-Time Communication