Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Telekommunikationssysteme

Bachelor thesis

Design and Evaluation of Distributed WebRTC Architectures

Furkan Tas
Informatics
Matriculation number: 393340

September 29, 2021

First Examiner: Prof. Dr. Manfred Hauswirth
Second Examiner: Prof. Dr. rer. nat. Volker Markl
Adviser: Dr.-Ing. Louay Bassbouss

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Bachelorarbeit ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Teile, die wörtlich oder sinngemäß einer Veröffentlichung entstammen, sind als solche kenntlich gemacht. Die Arbeit wurde noch nicht veröffentlicht oder einer anderen Prüfungsbehörde vorgelegt.

Ort, Datum	Unterschrift

Abstract

The aim of this thesis is to design novel distributed Web Real-Time Communication (WebRTC) architectures and compare and evaluate them together with traditional models. The traditional WebRTC models either only allow for a small number of participants in a session or use a central media server in a star topology. The central node leads to various restrictions. To circumvent the shortcomings in scaling and efficiency of a central server, a distributed system is required. We design two distributed architectures by redesigning existing central media server models so that the server can be exchanged with a distributed system and compare the implementations. Our Distributed Selective Forwarding Unit scales well and can be achieved with a custom interposed signaling server. The Distributed Multipoint Control Unit scales near perfect and can square the number of participants in a session by doubling the latency.

Zusammenfassung

Das Ziel dieser Arbeit ist es, neuartige verteilte Web Real-Time Communication (WebRTC) Architekturen zu entwerfen und diese zusammen mit traditionellen Modellen zu vergleichen und zu bewerten. Die traditionellen WebRTC-Modelle erlauben entweder nur eine kleine Anzahl von Teilnehmern in einer Sitzung oder verwenden einen zentralen Medienserver in einer Sterntopologie. Der zentrale Knotenpunkt führt zu verschiedenen Einschränkungen. Um die Nachteile in Bezug auf Skalierung und Effizienz eines zentralen Servers zu umgehen, ist ein verteiltes System erforderlich. Wir entwerfen zwei verteilte Architekturen, indem wir bestehende zentrale Medienservermodelle so umgestalten, dass der Server gegen ein verteiltes System ausgetauscht wird. Unsere Distributed Selective Forwarding Unit ist gut skalierbar und kann mit einem zwischengeschalteten Signaling Server realisiert werden. Die Distributed Multipoint Control Unit ist nahezu perfekt skalierbar und kann die Anzahl der Teilnehmer in einer Sitzung durch Verdoppelung der Latenzzeit quadrieren.

List of Acronyms

API - Application Programming Interface

AWS - Amazon Web Services

CPU - Central Processing Unit or a computer system

DMCU - Distributed Multipoint Conferencing Unit

DSFU - Distributed Selective Forwarding Unit

E2E - End To End

IP - Internet Protocol

KMS - Kurento Media Server

MCU - Multipoint Conferencing Unit

NAT - Network Address Translation

NTP - Network Time Protocol

OCR - Optical Character Recognition

P2P - Peer to Peer

RAM - Random Access Memory

RFC - Request for Comments

RTC - Real Time Communication

SDP - Session Description Protocol

SFU - Selective Forwarding Unit

STUN - Session Traversal Utilities for NAT

TURN - Traversal Using Relays around NAT

VM - Virtual Machine

W3C - World Wide Web Consortium

WebRTC - Web Real-Time Communication

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation . . . 1
- 1.3 Aim and research questions . . . 1
- 1.4 Structure . . . 2

2 State of the Art and Related Work **3**
- 2.1 Audio and Video Processing . . . 3
  - 2.1.1 Codec . . . 4
- 2.2 WebRTC Connection Establishment . . . 4
  - 2.2.1 Signaling and media negotiation phase . . . 4
  - 2.2.2 Interactive Connectivity Establishment (ICE) phase . . . 5
- 2.3 Developers Perspective . . . 6
- 2.4 Related Work . . . 6

3 Concept and Architecture **8**
- 3.1 Traditional Architectures . . . 8
  - 3.1.1 P2P Mesh . . . 8
  - 3.1.2 Selective Forwarding Unit (SFU) . . . 9
  - 3.1.3 Multipoint Conferencing Unit (MCU) . . . 10
- 3.2 Distributed Multipoint Conferencing Unit (DMCU) . . . 11
- 3.3 Distributed Selective Forwarding Unit (DSFU) . . . 12

4 Realization **14**
- 4.1 WebRTC media server . . . 14
  - 4.1.1 Attempt to build a custom SFU media server . . . 14
  - 4.1.2 Choosing a media server . . . 14
  - 4.1.3 Kurento Media Server (KMS) . . . 15
- 4.2 SFU architecture implementation . . . 16
- 4.3 MCU architecture implementation . . . 17
- 4.4 DSFU architecture implementation . . . 18
- 4.5 DMCU architecture implementation . . . 20
  - 4.5.1 DMCU tree . . . 20
  - 4.5.2 DMCU unit-to-unit . . . 21
- 4.6 Test setup . . . 21
  - 4.6.1 System under test . . . 21
  - 4.6.2 Measuring End-to-End (E2E) delay . . . 22
  - 4.6.3 Controlled Media for Consistency . . . 23
- 4.7 Test Methodology . . . 24

5 Evaluation **25**
5.1 SFU . . . 25
5.2 MCU . . . 25
5.3 DMCU . . . 26
5.4 DSFU . . . 27
5.5 Client-side Results . . . 27

6 Conclusion and Outlook **28**
6.1 Conclusion . . . 28
6.2 Outlook . . . 28

7 Appendix **30**

1 Introduction

1.1 Background

During the pandemic working from home became suddenly the new normal with web conferencing being an important means of communication. Research suggests that this practice will stick and that we are currently witnessing a permanent shift in the importance of real-time communication (RTC) due to the increased exposure and change of perception of the general public [1].

The Web Real-Time Communication (WebRTC) standard allows well known software like Microsoft Teams, GoToMeeting, Google Hangouts and many more to share audio, video and data in real time between browsers without additional software [2], [3]. Before WebRTC companies had to develop software to enable real time video conferencing with either a client or often with plugins like Adobe Flash for web browsers [4]. Now WebRTC is supported by all major web browsers and developers can leverage the high level Application Programming Interface (API) to exchange RTC data. WebRTC allows for peer-2-peer communication, for example video conferencing between browsers is possible mostly without an intermediary server.

Nonetheless WebRTC does not specify the topology in which data should be exchanged. Furthermore, video and audio data can be modified at the nodes allowing developers to be flexible with topologies and architectures.

1.2 Motivation

A simple way to provide web conferencing capability is to use an open source web conferencing solution like Jitsi or BigBlueButton deployed on a server. Typically in order to scale new servers each with an instance of the conferencing software is deployed alongside the existing instances.

For example in a project, video conferencing was enabled by deploying BigBlueButton, an open source web conferencing solution. Since a single BigBlueButton server instance was not enough to serve the large user base the users or more specifically meetings were load balanced on multiple servers. If a user planned a large meeting with more than a hundred participants a server would be automatically reserved exclusively for this meeting. Furthermore, the meetings were distributed in a way so that the servers had headroom in order to guarantee a high quality of experience and because the strain on the server could vary dynamically since e.g. users should be able to join or change their video quality at any time.

Although WebRTC does not specify how data should be processed and what topology and overall architecture should be used, typically these open solutions use a central node that forwards data when more than a few users need to communicate.

For many, this approach is viable due to the lack of alternatives and its simplicity. Yet the potential for improvement and alternative solutions captured our attention and curiosity.

1.3 Aim and research questions

To circumvent the shortcomings in scaling and efficiency of a central server, a distributed system is required. According to Tanenbaum [5, p. 2]

"A distributed system is a collection of independent computers that appears to its users as a single coherent system".

The aim of this thesis is to design novel distributed WebRTC architectures and compare and evaluate them together with traditional architectures. In particular scalability, efficiency, advantages, disadvantages as well as use cases shall be investigated.

The metrics to be examined both for the peers and, if present, for the servers include bandwidth, latency, number of streams and CPU load with a reference system specified. The aforementioned traditional architectures refer to the mesh-, Selective Forwarding Unit (SFU) and Multipoint Conferencing Unit (MCU) models.

In order to limit our scope, we will not examine WebRTC data channels which are used to exchange arbitrary data but instead focus on video and audio communication.

The goal of this thesis is to show that architectures besides the traditional ones are viable and preferable in their use cases.

Furthermore, the following questions will be addressed:

- Which paradigms and technologies exist to enable scalable and efficient real-time communication?
- What are the advantages and disadvantages for the user and the service provider when using the respective models?
- What technical hurdles can arise for the implementation of a distributed architecture in already existing real-time communication solutions?

1.4 Structure

This paper is organized in the following structure:

Chapter 2 - State of the Art and Related Work: This chapter gives information about WebRTC that is important for the subsequent chapters. This includes audio, video processing, connection establishment and an overview of WebRTCs high level programming API.

Chapter 3 - Concept and Architecture: In this chapter the WebRTC models are discussed in detail together with their advantages, disadvantages and their use cases.

Chapter 4 - Realization: Explains how we implemented the architectures and models from the previous chapter. The test setup and test methodology are also explained in detail.

Chapter 5 - Evaluation: The values gathered from the tests are being analysed.

Chapter 6 - Conclusion: In the conclusion findings are summarized and options for future research is suggested.

Chapter 7 - Appendix: The appendix contains mainly the raw test results.

2 State of the Art and Related Work

WebRTC (Web Real-Time Communication) is an open framework that enables peer-to-peer audio, video and data sharing in real time between browsers. The collection of standards, protocols and the simple JavaScript APIs allow teleconferencing inside the browser including smart devices without any additional software. Using its APIs, WebRTC abstracts much of the complexity of audio and video capturing and streaming for web application developers. The World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF) are responsible for defining the JavaScript APIs and the underlying communication. Unlike typical browser communication that uses TCP, WebRTC utilizes UDP if possible since lower latency is prioritized over reliability in real time applications. [6]

2.1 Audio and Video Processing

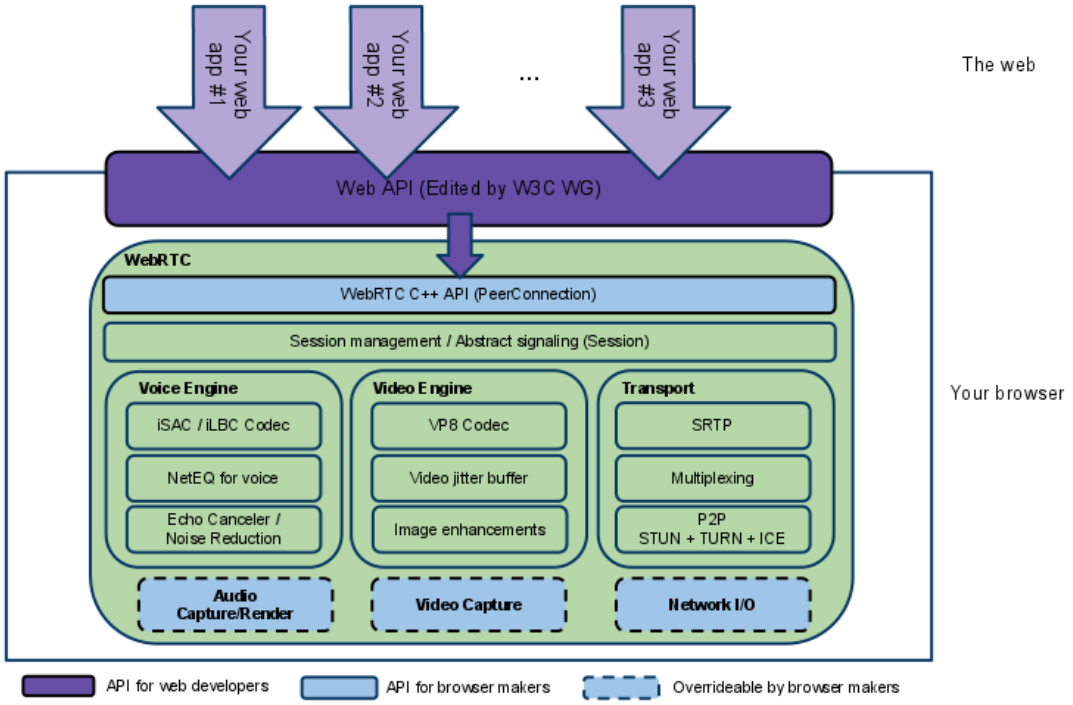


Figure 1: WebRTC architecture [7]

In order to enable audio and video communication in real time, these streams have to be captured and then processed in real time. For this reason WebRTC implementations are equipped with complete audio and video engines as can be seen in figure 1. The audio and video engines take care of error concealment, processing and optimizing quality by choosing optimal codec settings. The processing done by the audio and video engines includes on the sender side: synchronization, adjusting the output bitrate to the changing bandwidth, encoding and quality enhancements like echo and noise cancellation. On the receiving end clients have to decode in real time and adjust to jitter and delays. Furthermore, the browser adapts dynamically to changing audio and video stream parameters, as well as changing network conditions. [6]

To request audio and video stream the getUserMedia() method is used to get a MediaStream object. A stream is made up of multiple tracks (MediaStreamTracks) such as audio or video tracks. The MediaStreamTracks of a MediaStream are synchronized with each other and can also be used together with other browser APIs like WebGL, the Web Audio API and Canvas API. [6]

The getUserMedia() API also allows the developer to set constraints that specify which tracks should be included and which parameters should apply for the tracks of the returned MediaStream. Using these constraints, audio parameters and video parameters like resolution, frame rate and also parameters like gain control, echo cancellation, iso and focusDistance can be set. [8]

2.1.1 Codec

To transmit audio and video data, two devices must agree on a codec for each track. A codec is a program or device that encodes and decodes data streams by being able to handle specific compression technologies [9].

The WebRTC specification does not specify which codecs must be used. However, RFC7742 and RFC7874 specify that any WebRTC implementation must support at least the video codecs VP8 and H.264, as well as the audio codecs OPUS and G.711 [10]–[12].

2.2 WebRTC Connection Establishment

2.2.1 Signaling and media negotiation phase

In order to establish a peer-to-peer connection and allowing media exchange, a discovery and negotiation process has to take place, this is known as signaling. During signaling peers locate one another and exchange negotiation messages using an agreed-upon signaling server [13].

The Session Description Protocol (SDP) is used by WebRTC to negotiate session capabilities between peers [14]. Using the SDP protocol and a signaling server two peers can negotiate information about what kind of data they want to communicate like the media type, format, used codecs and their parameters. [13]

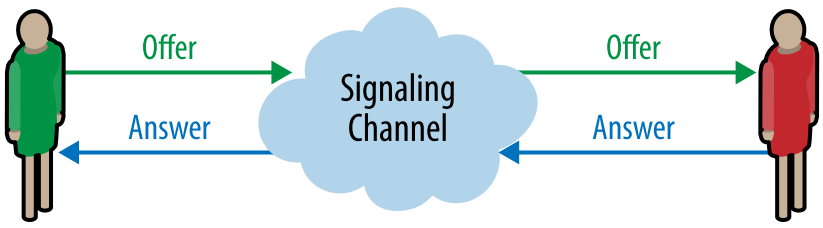


Figure 2: Offer/answer exchange using a shared signaling channel [6]

If a peer wants to call another peer it sends a SDP offer via the signaling server, which represents the peers local description, to the other peer. The other peer records this SDP offer as its remote description and returns an SDP answer. After the initiating peer sets the SDP answer as its remote description, media can be transferred. [6]

2.2.2 Interactive Connectivity Establishment (ICE) phase

After exchange of media information our peers have to establish a connection. Ideally our peers would only have to exchange their public addresses in order to transmit data. However locating one another requires additional steps due to NAT and firewalls being widely adopted [15], [16].

Network Address Translation (NAT) as described by [17] is a router function that allows the reuse of a public unique IP address inside a local network with multiple private local IP addresses, using a IP translation table. The advantage of NAT is that it originally solved the issue of depleting IPv4 addresses. However the disadvantage of NAT is that it lessens the end-to-end significance of an IP address.

A client that lives behind a NAT-server is difficult to address directly using IP in a peer-to-peer manner since the client's private IP address is exchanged with the NAT-servers public IP address. This poses a problem especially since NAT is widely used in residential networks, enterprises and organizations for peer-to-peer real time communication [15], [16].

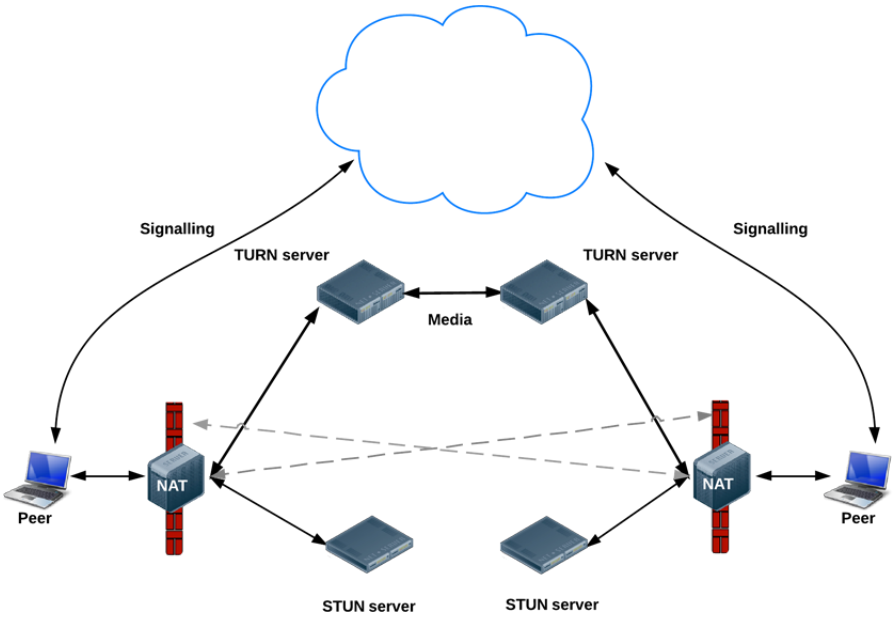


Figure 3: Relaying after STUN did not succeed [18]

In order to traverse NAT devices, WebRTC uses a method called Interactive Connectivity Establishment in short ICE that is built on top of two protocols: STUN and TURN [19].

STUN (Session Traversal Utilities for NAT protocol) enables an endpoint to determine the NAT-assigned IP address and port that corresponds to its private IP address and port. Using STUN a direct peer-to-peer connection between the peers can be made using UDP. STUN is a client-server protocol and requires a STUN server that resides on the public network. [20]

TURN (Traversal Using Relays around NAT) is also a client-server protocol and allows to relay data between two peers by using a relay server that is commonly referred to as a TURN server. A TURN server is best to use when STUN fails and therefore a direct communication is not possible. [21]

ICE starts with each peer collecting its possible STUN and TURN candidate transport addresses using STUN and TURN servers. The collected ICE candidates are then exchanged by the peers using the signaling server. The peers then try to establish a connection with the best candidates first. [19]

2.3 Developers Perspective

WebRTCs JavaScript APIs abstract significant amounts of complexity. For example the RTCPeerConnection API encases connection setup, management and connection state [6].

The following describes simplified steps that have to be undergone in order to successfully establish a connection and communicate from the perspective of the initiating peer using the JavaScript APIs [22].

First a STUN, TURN and signaling server should be set up. Coturn [23] is an open source implementation for TURN and STUN. For demonstration purposes the signaling server can be a simple broadcasting application.

- Creating a new `RTCPeerConnection(configuration)` with `configuration.iceServers` being the addresses to the STUN and TURN servers.
- Using `getUserMedia()` to receive a stream and passing the streams tracks to the RTCPeerConnection object `peerConnection.addTrack()`
- Calling `peerConnection.createOffer()`, the returned SDP offer is passed to `peerConnection.setLocalDescription()` and sent to the other peer using the signaling server
- After executing `setLocalDescription()` the WebRTC implementation automatically starts asynchronously collecting ICE candidates, this is referred to as trickle ICE [6]. The developer implements the callback function `peerConnection.onicecandidate` and sends the ICE candidate to the other peer.
- On receipt of the SDP answer our peer can set `peerConnection.setRemoteDescription()` and will receive the ICE candidates asynchronously
- Finally when our peer has set its `remoteDescription` to the SDP answer media exchange is possible and `peerConnection.onTrack` will trigger the developers callback function and remote media can be displayed in the browser

2.4 Related Work

Muaz Khan's WebRTC Scalable Broadcast is a peer-to-peer broadcasting demo. Instead of a regular mesh it uses a tree topology to propagate media [24]. This circumvents the main issue of a regular mesh topology being the upload limitations of end users [25].

It works as follows: The root node is the one that shares its media. Every node except the root node receives media and has the ability to broadcast it. The nodes achieve that by having two `RTCPeerConnection`s and using the incoming stream of the receiving peer connection as the outgoing stream of the peer connection that broadcasts the media to other nodes. [24] Muaz Khan distributes the peers with a function called

`getFirstAvailableBroadcater()`. It returns the first peer that does not already broadcast to three other peers. This results in the tree depicted in figure 4 with the numbers indicating the sequence in which the peers are added. [26]

The disadvantage of this approach is that the multiple relays result in a loss of quality as well as an increase in latency [24]. The loss of quality and decrease of latency with each added relay can be seen when looking at an official WebRTC sample [27]. In this sample the local media stream is shown on the left and on the right we can see the media stream after it has been relayed, similar to the peer-to-peer broadcasting demo, a specific number of times.

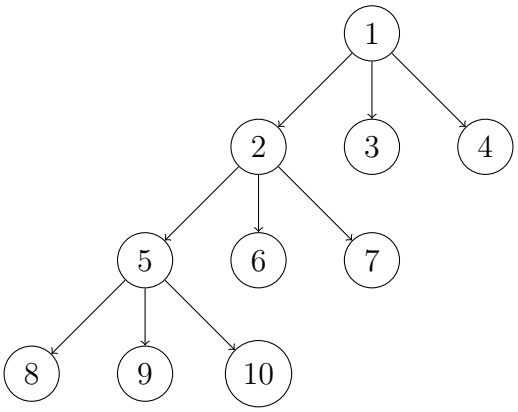


Figure 4: WebRTC Scalable Broadcast [24]

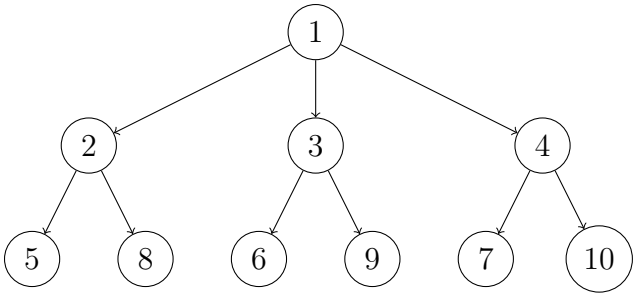


Figure 5: Alternative peer arrangement suggestion

We however do not see the advantage of choosing this order instead of a breadth-first approach depicted in figure 5. The benefit of the breadth-first order would be that the depth of the tree would be reduced and would therefore yield better latency and quality. Nonetheless WebRTC Scalable Broadcast is favorable to a typical mesh topology for a peer-to-peer broadcast or in a one-to-n situation if the peers are limited in their upload or CPU capacity.

3 Concept and Architecture

3.1 Traditional Architectures

There are three common WebRTC architectures. This section describes each together with their advantages, disadvantage and use cases.

3.1.1 P2P Mesh

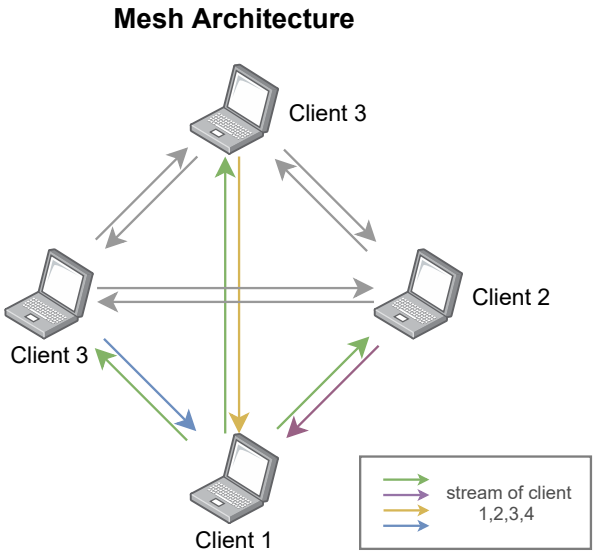


Figure 6: Peer-to-peer mesh with four peers

WebRTC is designed for peer-2-peer use and for that reason a P2P mesh architecture is the first architecture that comes to mind. A mesh network is a topology in which every pair of nodes is connected by a path or a direct connection [28]. In the context of WebRTC when we talk about a mesh topology we refer to a fully connected peer-to-peer mesh network, since typically every participant wants to be able to broadcast its media directly to every participant. Therefore every client has $n-1$ outgoing streams and $n-1$ incoming streams, in total there are $n(n-1)/2$ connections with n being the number of participants.

A major advantage of a P2P architecture is that no media server is required. Since streams are sent directly to the respective recipient, processing time and network latency of a media server is omitted. Therefore the latency is lower compared to architectures that require a media server, assuming that packets are routed by equal means.

The biggest disadvantage of a mesh architecture is the fact that clients have to upload their streams $n-1$ times. Typically end users have less bandwidth available for uploading than for downloading data [25]. The number of possible participants is limited by the available upload rate of peers.

For example in Germany the median upload speed is 9.78 MBit/s [25] in 2020. Although one of our laptop's webcam with VGA resolution (640×480) produces up to 2.11 MBit/s in network load, research shows that increases in video bitrate has diminishing returns in perceived video quality and 0.5 MBit/s and 1 MBit/s are sufficient for 360p, 720p video [6], [29], [30]. This means for example that six participants can video chat if we require

video participants to have a minimum upload rate of 3-5 MBit/s and each video stream uses 0.5 - 1.0 MBit/s.

Furthermore, all post processing like recording, echo cancellation and synchronization has to be done on the client side. For example a recording functionality for consumers might not be reasonable using this architecture because less capable devices like smartphones and older hardware might not be able to do this kind of video processing. Clients are also likely to perform the same type of processing like recording, therefore a central node would be beneficial.

Whilst a peer-to-peer mesh architecture does not require a server, servers are still desirable for discovery, signaling, relay and firewall traversal purposes. However these tasks are low in resource consumption and are generally carried out when a WebRTC service is offered. [6]

For these reasons a P2P mesh architecture is optimal for small video conferences or up to medium sized audio conferences where no resource intensive processing is necessary. The following two architectures on the other hand are better suited if large sessions or additional processing is necessary.

3.1.2 Selective Forwarding Unit (SFU)

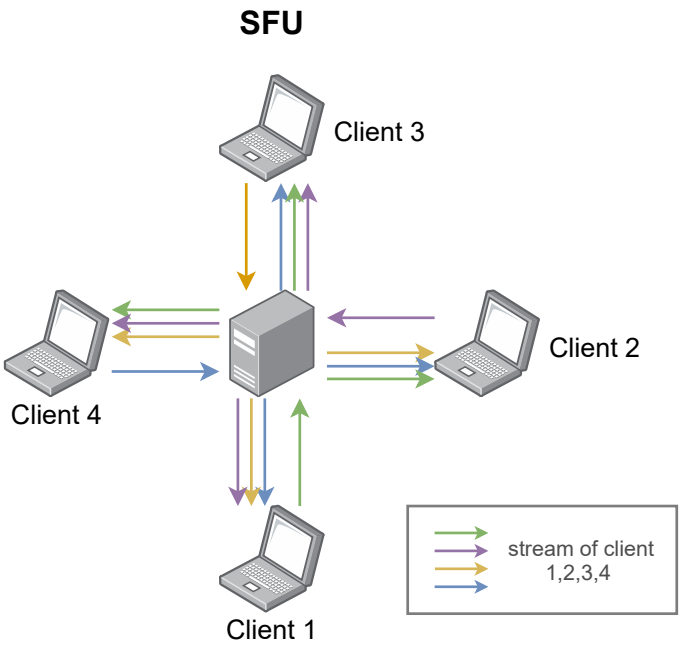


Figure 7: SFU with four clients

A SFU architecture is typically a star topology with the media server in the center and the leaves being clients that want to communicate with each other. Each client sends its streams to the SFU which then relays them to the other participants, as a result each client receives multiple streams. The streams are processed as little as possible. [31]

The main advantage of an SFU is low cost on the service provider side whilst still being able to allow large sessions and to serve many users. An SFU avoids the issue of the clients upload rate becoming a bottleneck, which is a major disadvantage of P2P mesh architectures. Using an SFU each client only has to upload its streams to the server.

A drawback of using an SFU architecture is the fact that clients have to decode streams of n-1 participants. The available processing power of clients limits the amount of videos that can be displayed. This disadvantage is also shared by a P2P architecture.

Although an SFU does as little processing as possible, processing is still required if transcoding due to incompatible browser codecs or other video processing is required [32].

An SFU architecture is the default choice for web conferencing solutions since unlike a P2P mesh solution it enables large sessions and medium sized sessions if every participant has video enabled. If however meetings with a very high user count or high amount of video participants is necessary, the following architecture is recommended.

3.1.3 Multipoint Conferencing Unit (MCU)

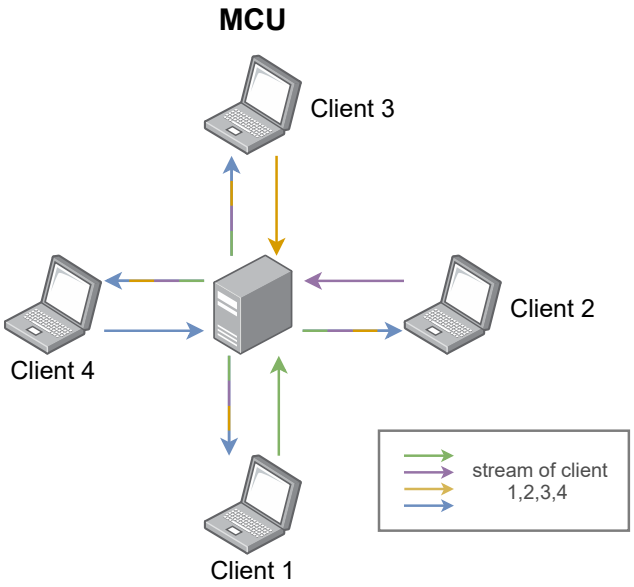


Figure 8: MCU with four clients

The MCU receives multiple streams from the participants but only returns a single video and or audio stream to the participants which represents the composite media that the MCU receives. The result on the client side is a single video in which the video streams of participants are scaled and arranged in a certain layout e.g. a grid. [4]

The main advantage of using an MCU is that it allows clients to display many participants, with the main limiting factor being how many streams can be arranged in a single video which can also be transmitted and displayed properly. An MCU architecture is especially less taxing on clients, since only one video has to be received and processed instead of n. Zoom for example can display up to 300 participants on the users screen utilizing an MCU [33], this would not be possible using an SFU due to the clients processing restrictions. Additionally the server needs less outgoing bandwidth.

Using an MCU instead of other architectures has however multiple important drawbacks. First and likely the biggest reason why MCUs are less commonplace than SFUs, as can be seen in section 4.1.2, is the fact that it is computationally expensive on the server side. In addition since the video is rendered on the server side the layout is not as flexible, the client can not change the layout to its liking. Supporting multiple form factors would

require the server to render multiple videos and is therefore not advisable. Participants can not individually resize and hide videos of select participants. If audio is also composited into a single track participants can not mute or change the volume of individuals. [34]

In conclusion an MCU is a good choice when lots of participants are supposed to be seen at once, however the inflexibility and cost to operate should be kept in mind.

3.2 Distributed Multipoint Conferencing Unit (DMCU)

A regular MCU server receives media from its clients, composites it and sends the result back to the clients. In order to achieve a distributed MCU architecture there are multiple possibilities.

The first variant is that participants are distributed equally among the MCUs and each MCU sends its result to all clients. The client's responsibility is to display the different streams from multiple MCUs in a coherent layout. For example if we assume a grid layout and have two MCUs, the client would receive half of a complete grid from each MCU and would be responsible to display them in a coherent grid. This architecture could be achieved without changing the MCU itself since the MCUs do not communicate with each other. A load balancer and modified client code could be sufficient.

Using this approach the video processing load would be split among the MCUs, however it could be difficult to achieve complicated video layouts and unlike a regular MCU the client now needs to process multiple video streams.

Another possibility would be an architecture where again participants are distributed among the MCUs, but each MCU only sends streams back to its own participants and the MCUs communicate with each other. Every MCU composites the media of its own participants and sends it to all MCUs. Each MCU composites all results of the other MCUs together with its own and sends the stream to its participants. Each participant receives a single stream that is a compound of all participant streams.

In this variant every MCU has to broadcast its results to every MCU, since this scenario is similar to that of a web conference a SFU could be used. However in this situation a mesh where every MCU broadcasts its data directly is the most suitable in order to avoid additional overhead. This architecture can be achieved by changing the MCU logic and adding a load balancer. Modifications to the client are not necessary.

The advantage over the former variant is that the client only has to handle a single video stream.

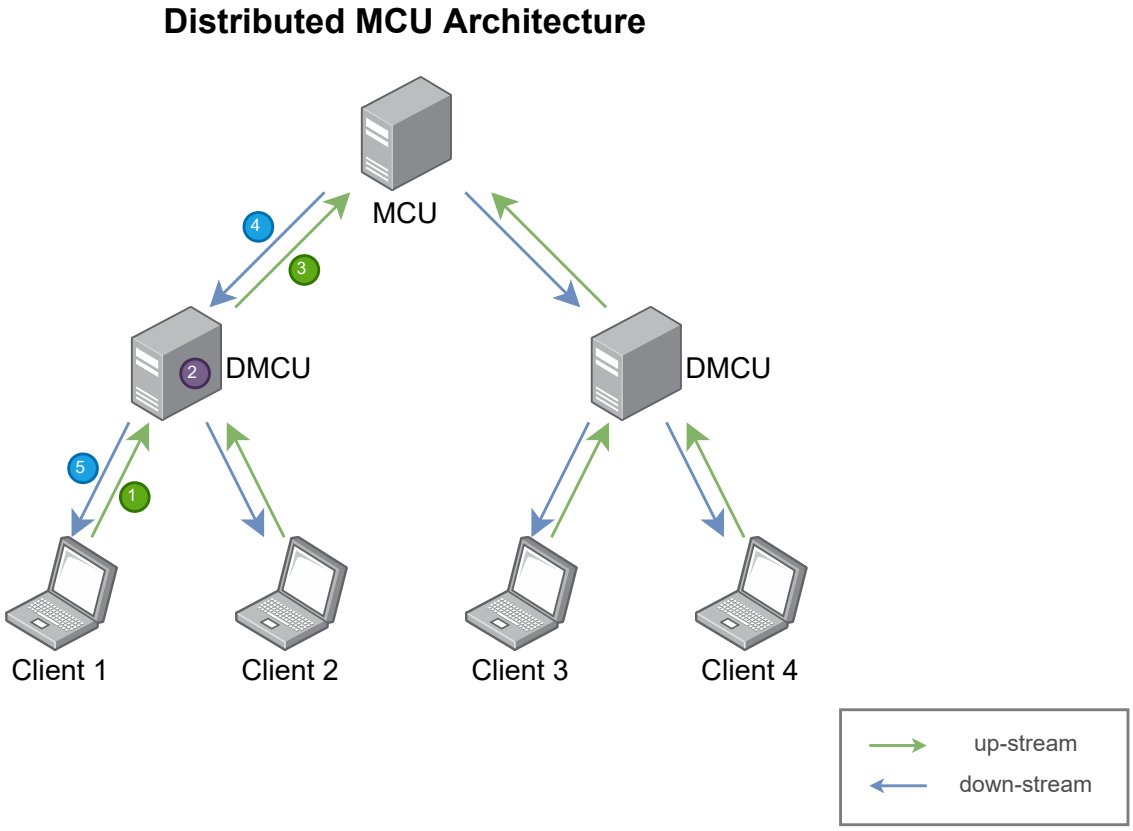


Figure 9: Distributed MCU tree architecture with four clients

At first we implemented the second variant with two MCUs which communicate directly with each other. However we encountered a problem with our media server, the result of the composite element can not be composited again without causing render anomalies in the same media server. This could have been fixed with a workaround, but we decided against it since it would have added processing cost and delay which would have falsified the tests.

In our case the MCUs are ordered in a tree structure. The children send their composite streams recursively to their respective parent until the root MCU is reached. The resulting stream contains media from all leafs and is trickled downwards from the root MCU to the leafs that are represented by clients. In order to achieve this structure each distributed MCU (DMCU) node has additionally an RTC endpoint to its parent, except the root node since it has no parent.

The view from a single DMCU is as follows: The DMCU sends its composite stream of its clients to its parent MCU. The parent MCU returns a composite of composites including the stream of the DMCU. Finally the DMCU forwards the received stream from its parent to its clients.

3.3 Distributed Selective Forwarding Unit (DSFU)

In general terms an SFU is a relay unit that forwards streams to relieve its clients from the associated network load. In order to turn a single SFU into a distributed system two possibilities come to mind.

One option is that every participant sends its streams to all SFUs so that every SFU still receives all streams. Unlike a regular SFU each distributed SFU forwards the streams it receives only to its equal share of participants. The client only receives streams from a single SFU. This distributes the outgoing streams equally among the SFUs and the associated processing cost. However in this variant clients have to send their streams to multiple SFUs which lessens the advantage over a P2P mesh.

The superior option would be to split the outgoing streams of the clients equally among the SFUs where each client sends its streams only to its respective SFU. Each SFU forwards the streams that it receives to all clients. A client receives streams from multiple SFUs. This variant keeps all advantages of a regular SFU. The number of incoming and outgoing streams are equal to a regular SFU and are distributed evenly among the SFUs. Therefore the load should also be distributed evenly among the SFUs.

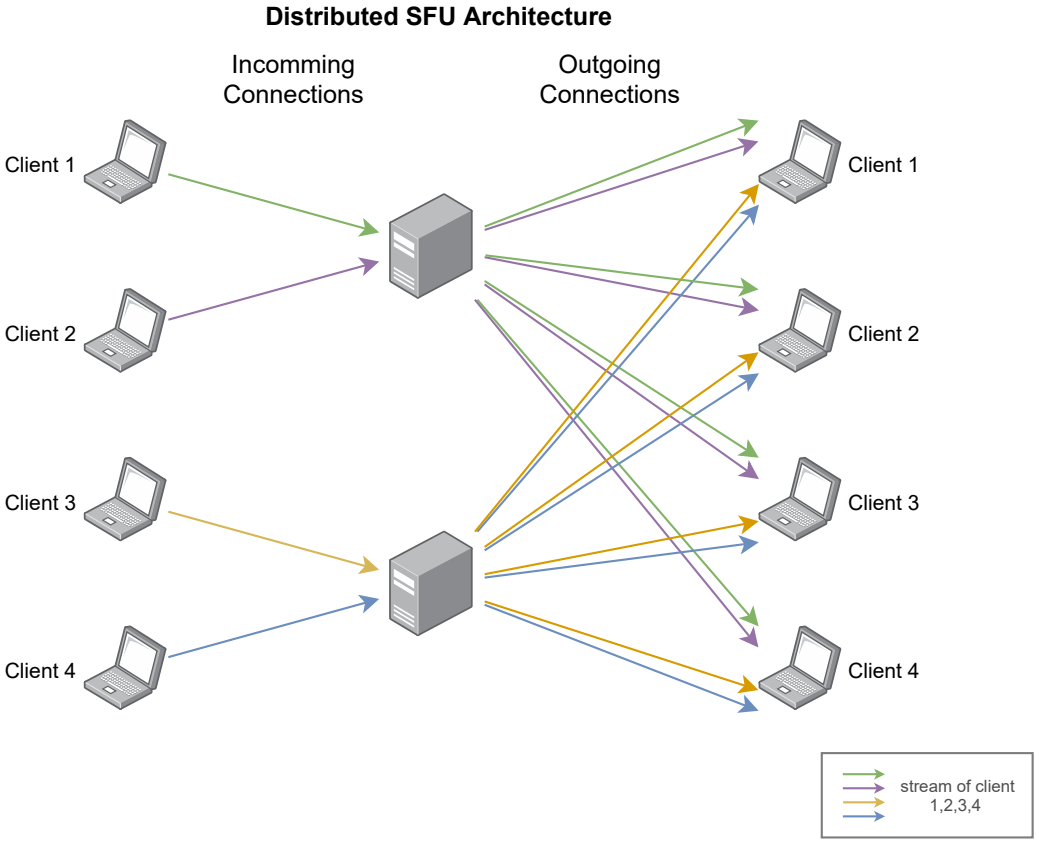


Figure 10: Distributed DSFU architecture with four clients

4 Realization

4.1 WebRTC media server

In chapter 3 we have discussed that in order to achieve group communication and media processing a WebRTC media server is typically required. A media server enables for example a SFU or MCU architecture.

A WebRTC media server is a type of RTC media server with WebRTC capabilities. Unlike a streaming media server a RTC media server is designed for bidirectional communication and low latency applications therefore packet retransmission is not vital. [35]

A WebRTC media server typically enables the following functionality as media passes through it [36]:

- media bridging capabilities to attain interoperability despite incompatible media formats and protocols e.g. transcoding
- mixing and or forwarding for group communication capabilities
- media archiving capabilities

4.1.1 Attempt to build a custom SFU media server

At an early stage we tried to implement software that acts as a SFU and MCU in order to have as much control as possible. We used a project called node-webrtc [37] that brings spec-compliant WebRTC functionality to Node.js. We successfully built a SFU-like prototype however had to find out that our CPU utilization was unrepresentatively high. We quickly realized that although our server is incapable of displaying media, media still got processed by the audio and video engine of the WebRTC implementation. One option to solve this issue would have been to alter the underlying C++ WebRTC implementation and expose an API to toggle media decoding and encoding. However since success was not guaranteed we decided against this approach and chose to use a WebRTC media server.

4.1.2 Choosing a media server

Our main requirement for a media server is that it should be flexible and allow us to achieve different architectures, ideally without modification to its source code. It should have support for mixing and forwarding so that we can build a SFU, MCU and hybrid topologies.

Janus, Licode and Kurento are the media servers that can both be used as an SFU and MCU. Janus WebRTC gateway is a lightweight solution and depends on plugins to enable functionality. However a plugin for Janus that enables MCU functionality is not readily available. Licode claims to enable MCU functionality using a specific module but this module is not capable of mixing [38], [39].

	Isolation	Abstraction	Composability	Reusability	Extensibility
Jitsi	×	✓	×	×	×
Janus	✓	✓	×	✓ (partial)	✓
Medooze	×	✓	×	×	×
Licode	✓ (partial)	✓	×	×	×
Kurento	✓	✓	✓	✓	✓

Figure 11: Modularity comparison between popular media servers [35]

4.1.3 Kurento Media Server (KMS)

We chose Kurento media server (KMS). Kurento is a very good match for our purposes; it is designed with modularity in mind, is well documented and rich in capabilities. It enables transcoding, recording, mixing, broadcasting and routing as well as advanced media processing like computer vision. Kurento's unique feature is its modular architecture which complies with the main characteristics of modularity: isolation, abstraction, composability, reusability and extensibility as demonstrated in figure 11). [40]

The main disadvantage of KMS is that it shows unreliable RTT and video quality which is pointed out in a comparative study of open source SFUs [41] and can be seen later on in some of our tests.

Kurento API

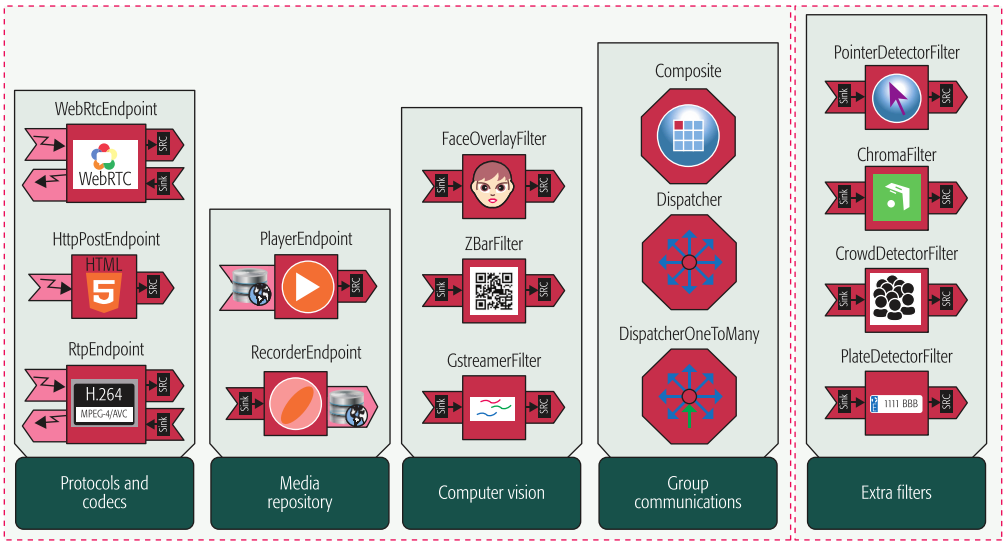


Figure 12: Kurento media element toolbox [35]

In practical terms the KMS exposes an API that can be used to orchestrate media elements with each media element enabling certain functionality [35]. The available media elements can be seen in figure 12. A graph of interconnected media elements is referred to as a media pipeline [35]. Examples of media pipelines can be seen in figures 14,15,17,18 which use illustrations for media elements from figure 12. Every media element has to be part of

a media pipeline and can not access elements outside its pipeline. Pipelines are meant to be modified during their lifetime and are typically used to isolate and divide applications and sessions.

Kurento offers API clients for JavaScript, Node.js and Java. We chose the Java Client resulting in a three-tiered application. The application server implements the pipeline-, application logic and mediates between browser and the media server. Once a connection is established, media traffic flows directly between the browser and the media server. The Java clients methods abstract the fact that the application server and the Kurento media server are separate entities.

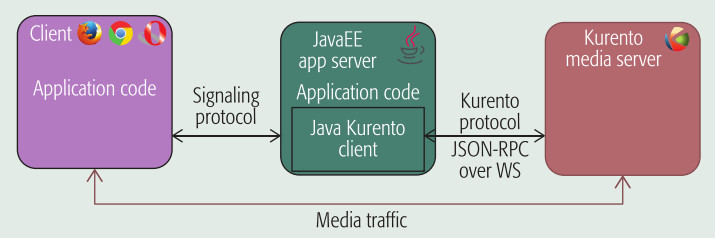


Figure 13: Three-tiered web development model using Kurento [35]

4.2 SFU architecture implementation

In order to have a point of reference for the tests of the distributed architectures we first need to have a working SFU and MCU architecture as a baseline.

For our SFU architecture we use the example application from the Kurento project that is available at [42]. Modifications were made to remove unnecessary HTML/CSS and to display streams in a grid for testing. Further URL parameters were added to make automation easier.

The application server is Spring Boot based with the Kurento client as a dependency. Spring Boot has the advantage over traditional Java web applications in that it directly embeds a webserver into the application. Typically Java web applications require a webserver to be installed and after compilation the resulting file needs to be deployed on the webserver. Spring Boot automates that process and allows the developer to "just run" the web application [43] using in our case the embedded Tomcat webserver.

When a client connects to the URL of the webserver it gets served a website including the necessary JavaScript code. To join a room the client sends a `joinRoom` message to the server. The server generates a `UserSession` object for every client.

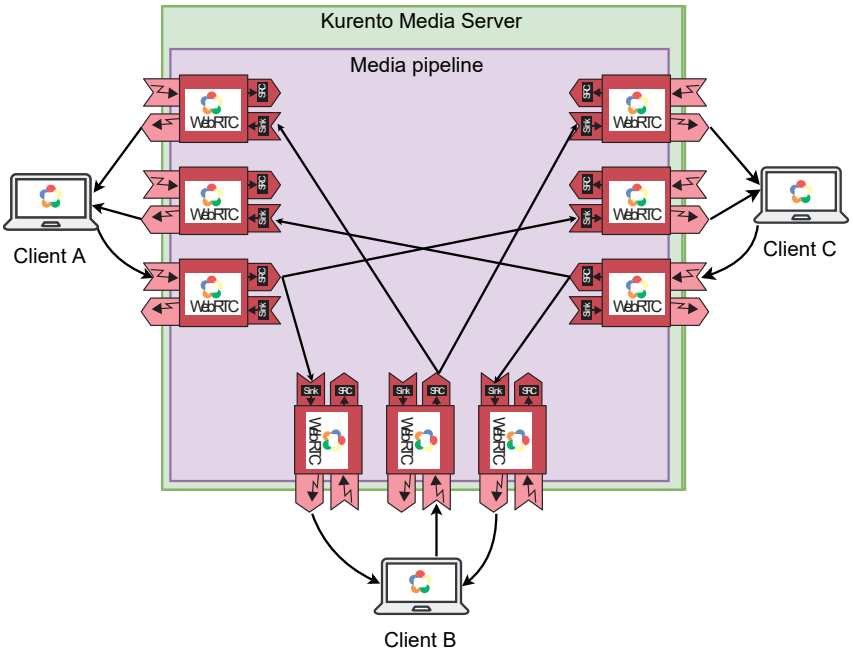


Figure 14: Media pipeline in an SFU scenario with three clients

A `UserSession` object has one `WebRTCEndpoint outgoingMedia` and has a list with $n-1$ `WebRTCEndpoints` called `incomingMedia`. After a client sends a `joinRoom` message, the server responds with the names of the participants. The client then sends for each participant name it received, including its own, a `receiveVideoFrom` message which includes an SDP offer to the server. The server generates the necessary endpoints in the `incomingMedia` list for that client to receive media from. The server also generates the `outgoingMedia` endpoint to distribute that client's media and then sends an SDP answer to the client. When a new client joins the room, every participant receives a `newParticipantArrived` message so that they can request media using a `receiveVideoFrom` message.

If for example three clients called A, B and C want to communicate with each other, the media pipeline inside the media server would look like in figure 14. To receive media from client B and C, client A uses two RTCPeerConnections that are connected to the respective `incomingMedia` endpoints. Client A has a separate peer connection to an `outgoingMedia` endpoint that acts as a source for the endpoints of client B and C.

Using this implementation, every client has n peer connections. Alternatively it could be implemented in a way that every client has a single RTCPeerConnection object with multiple incoming and a single outgoing stream. This is referred to as multistream and has the advantage that it saves overhead and uses fewer resources on the client side. However multistream is less flexible, difficult to implement and requires renegotiation with everyone when a participant joins or leaves [44], [45].

4.3 MCU architecture implementation

Similar to the realization of the SFU we use Spring Boot. We reuse code that is architecture-independent like ICE handlers, Spring Boot specific code, session handling, and event listeners and focus on the media pipeline and application logic.

In the MCU pipeline every client has a single WebRTC endpoint. Every endpoint is connected to a `Composite` element. The `Composite` element mixes audio streams and

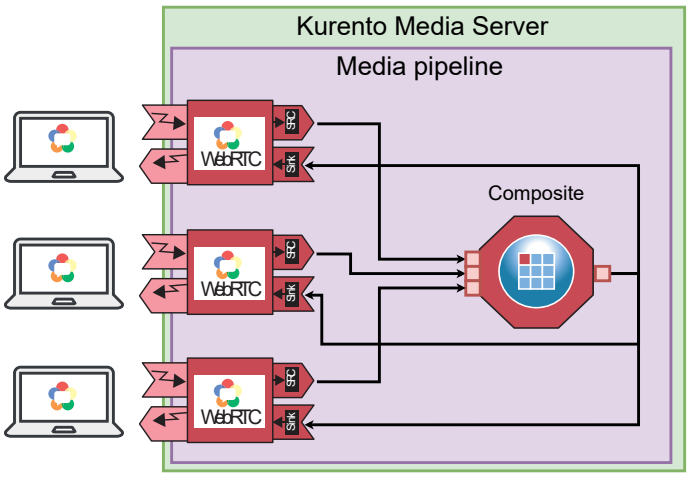


Figure 15: MCU media pipeline using composite element

constructs a grid with the video streams that it receives [46]. Because `Composite` is of type `Hub` it requires a `HubPort` to communicate with endpoints. A `HubPort` element is displayed in figure 15 as a square stub and has one sink and multiple sources. Every WebRTC endpoint connects first to a separate `HubPort` and a single `HubPort` distributes the mixed streams back to the endpoints.

Every client has a single peer connection that connects to its respective endpoint from which it receives mixed media streams and sends its media to.

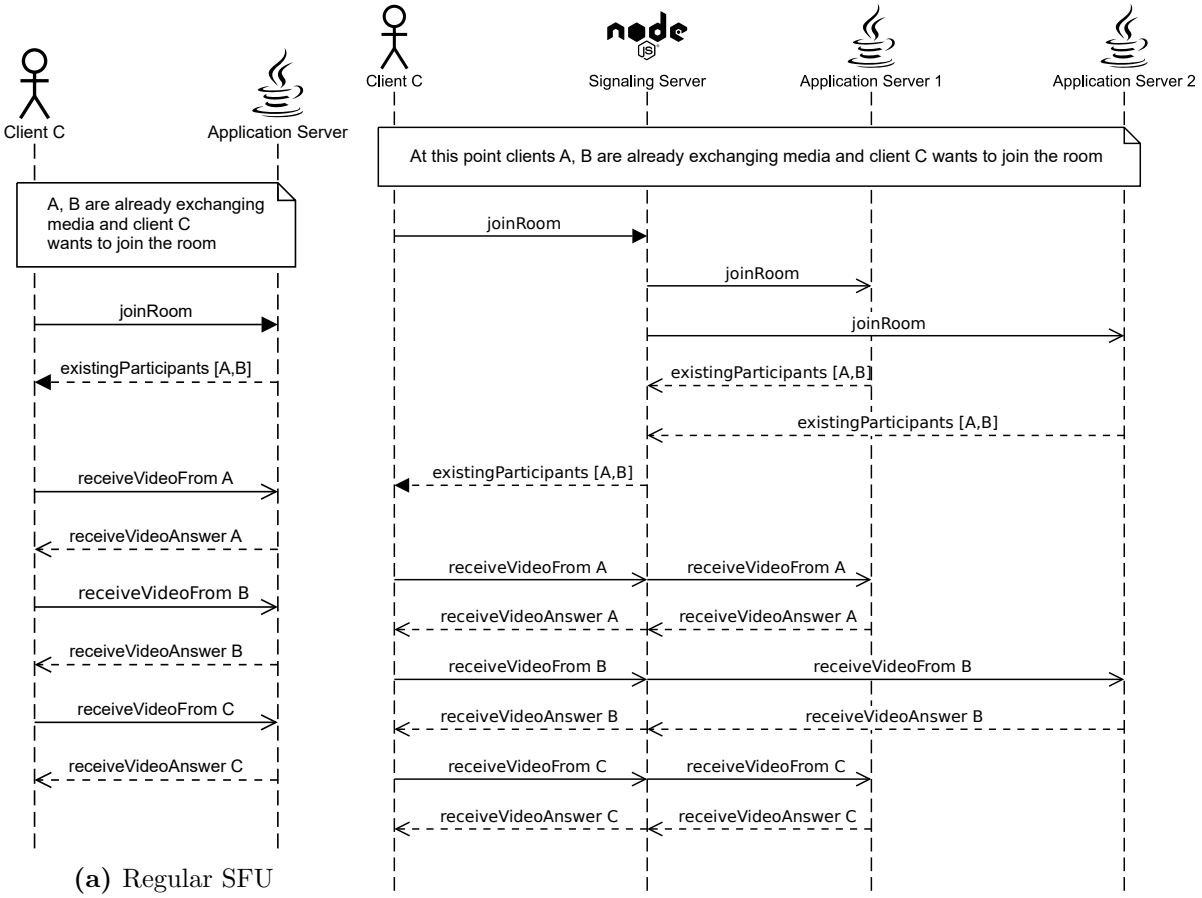
4.4 DSFU architecture implementation

For our DSFU we reused the SFU from subsection 4.2. When a new client joins the server, the server sends all the participant names to the client. The client can then send SDP offers, receive SDP answers using the application server and connect to the media servers endpoints.

The goal is to distribute the load equally on multiple media servers even if all participants are in a single room. One option to achieve a distributed architecture would be to modify the application in a way that it controls multiple media servers. The application would distribute the clients streams and endpoints equally on the media servers.

However we opted for a variant where we deploy the SFU topology example unmodified from subsection 4.2 multiple times in parallel and build a signaling server in front of the application servers to achieve load balancing. We chose this variant to demonstrate that a distributed SFU architecture can be achieved in some cases with little modifications to existing software.

The signaling application acts as a load balancer by routing messages between client and the application servers in a way that the number of streams and WebRTC endpoints are distributed equally among the media servers. The signaling application is written in JavaScript and runs using the Node.js platform.



(a) Regular SFU

(b) DSFU with custom signaling server

Figure 16: Simplified sequence diagram: Difference between SFU and DSFU

When a client wants to join a room, the signaling server first assigns that participant to one of the SFUs in a round robin manner. If a client wants to receive media from a specific participant, it makes an SDP offer to the signaling server which forwards the request to the corresponding application server. The application server responds to the signaling server which then forwards the SDP answer to the requesting client.

For example if we have two SFUs from subsection 4.2 and two clients A and B. Client A would be assigned to application 1 and client B to application 2. Now client C wants to join the session. Figure 16b depicts how messages would be routed by the signaling server. As a comparison figure 16a shows how a single SFU from subsection 4.2 would behave.

First client C is registered to all applications and receives in return a list of existing participants. Because both applications respond with an identical message the signaling server only forwards one of them. Then client A is able to request media from A and B. The media request for client A is forwarded to SFU 1 and the request for B is forwarded to SFU 2. The answers are forwarded by the signaling server back to client C. Because client C also wants to share its audio and video, it establishes a connection to its assigned SFU using the signaling server.

Both figures are simplified and do not show messages between the application and the media server and do not display ICE candidate handling or messages towards clients A and B. For our tests we will use two SFU nodes.

4.5 DMCU architecture implementation

In order to achieve a DMCU architecture we explored two variants in section 3.2. Both require mixing the results of multiple MCUs and returning the result to the user.

4.5.1 DMCU tree

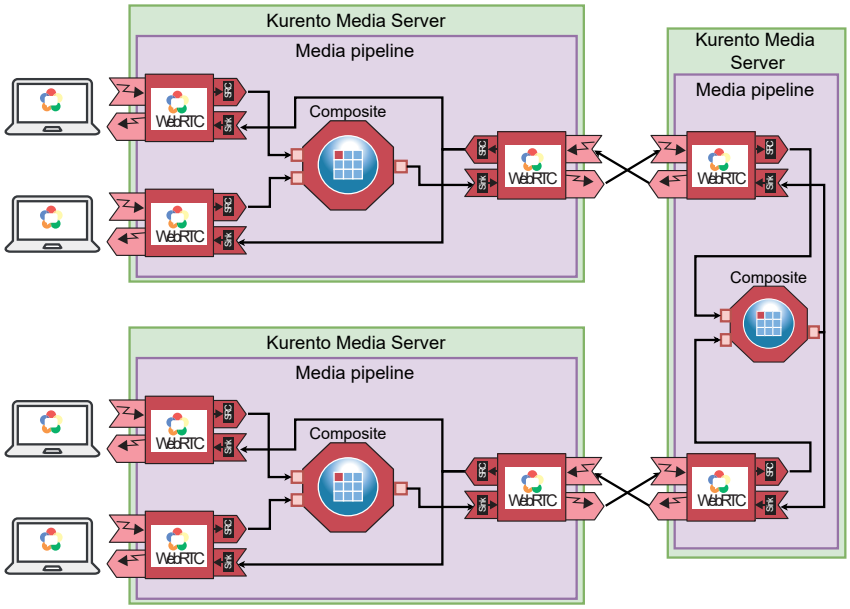


Figure 17: Tree DMCU architecture with one root node and two leafs

This variant orders MCUs in a tree structure. The root MCU is the one implemented in section 4.3. The non-root MCU nodes are however customized. First they additionally posses a WebSocket server that can actively establish a connection unlike the WebSocket client. Secondly they have an additional WebRTC endpoint that connects to the parent node. This endpoint sends the result of the composite element to its parent. Further, the endpoint receives media from the parent and acts as a source for the other endpoints. For our measurements we will use three MCUs with one of them being the root node.

4.5.2 DMCU unit-to-unit

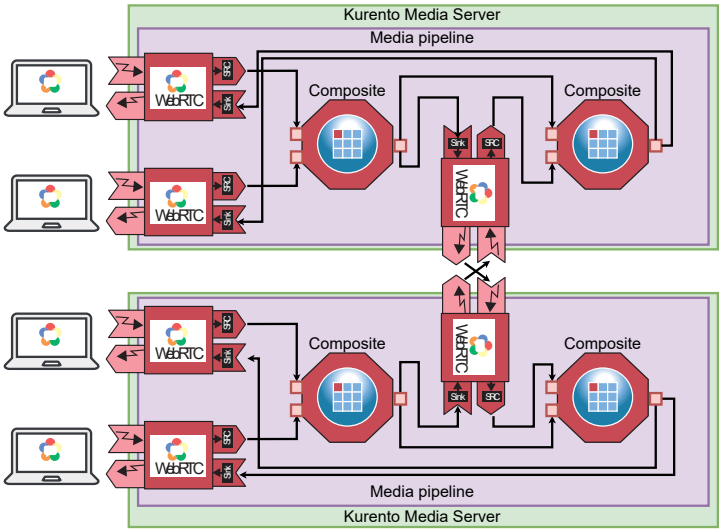


Figure 18: DMCU unit-2-unit architecture

In this variant, the MCUs exchange their interim results using WebRTC endpoints similar to how peers exchange media in a peer-2-peer fashion. Each MCU channels its sub result together with the sub results of the other MCUs into a second `Composite` element. From there the MCU then sends the final result back to the clients. To establish a connection with other nodes the MCUs each have a WebSocket server and a client similar to the previous subsection.

4.6 Test setup

4.6.1 System under test

For our tests we used Google Chrome 85 and Kurento Media Server 6.15. To eliminate network inconsistencies and transmission delay we ran clients and servers using Amazon Web Services (AWS) on Elastic Compute Clouds (EC2) virtual machines (VM). All VMs were located in the same availability zone. The used instance types are depicted in Table 1

	Media Server VMs	Client VMs	Client Measurement VM
AWS instance type	m5.large	m5.8xlarge	m5.xlarge
vCPUs	2	32	4
Amount	1-3	1-2	1
Rationale	few but performant cores for measuring perfomance and limits	powerfull VM to generate as much clients as possible	realistic specification of an end consumer
RAM	8 GB	128 GB	16 GB

Table 1: Instance types of the VMs for the media server and clients

We want to measure how the different architectures scale and find out what their limitations regarding number of users in a single session are.

Therefore for our media servers we chose an instance type with few cores that are still reasonably performant. For the clients two different VM instance types were used for two different purposes. A single m5.xlarge instance was used to determine the CPU load that occurs on the client as well as latency between clients. Two m5.8xlarge VMs with 32 vCPUs each were necessary to generate video participants since it becomes increasingly difficult to add new video participants in certain scenarios.

One possible solution would be to avoid video decoding and processing, however this would only be possible with a different or custom WebRTC engine.

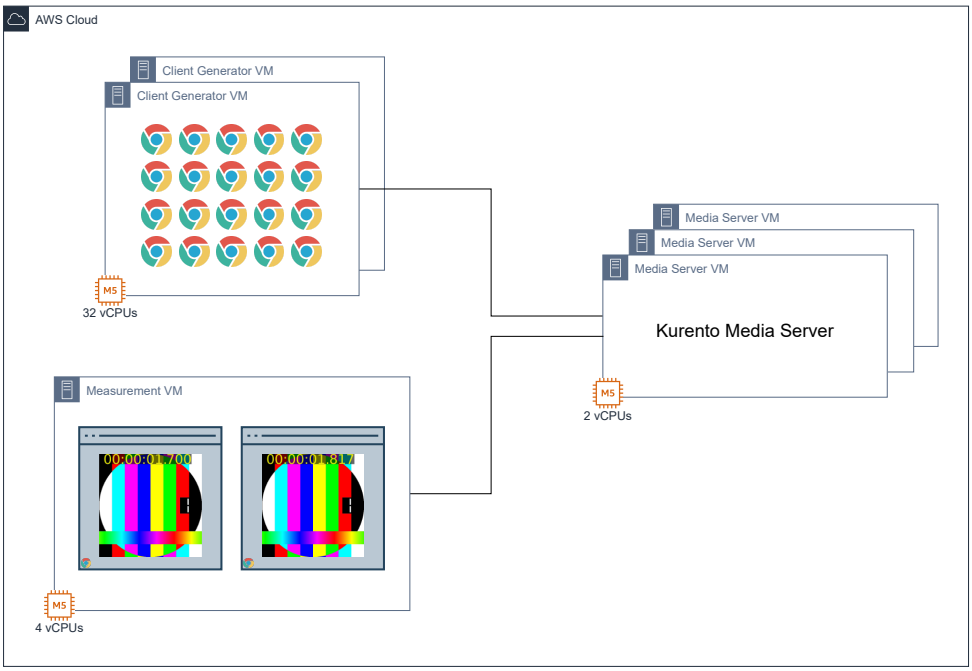


Figure 19: Test configuration [35]

4.6.2 Measuring End-to-End (E2E) delay

We limit ourselves to measuring video latency but will also discuss how audio latency can be measured in a similar fashion.
Video latency can be measured visually by taking a screenshot of a video stream from the source participant and the destination participant. Since every video stream has a visible timestamp embedded we can read off the E2E latency between two participants by determining the timestamp deltas and by subtracting the NTP delta between the clients clocks.

$$e2eDelay = timestampAtReceiver - timestampAtSender - \Delta NTP$$

In our case we do not have to account for the NTP delta because both participant are on the same machine. A screenshot is taken and the timestamp of the outgoing video is substracted by the one of the received video. We repeated this three times for each measurement.

For measuring audio latency a similar option would be to record and compare the resulting audio files instead of using timestamps.

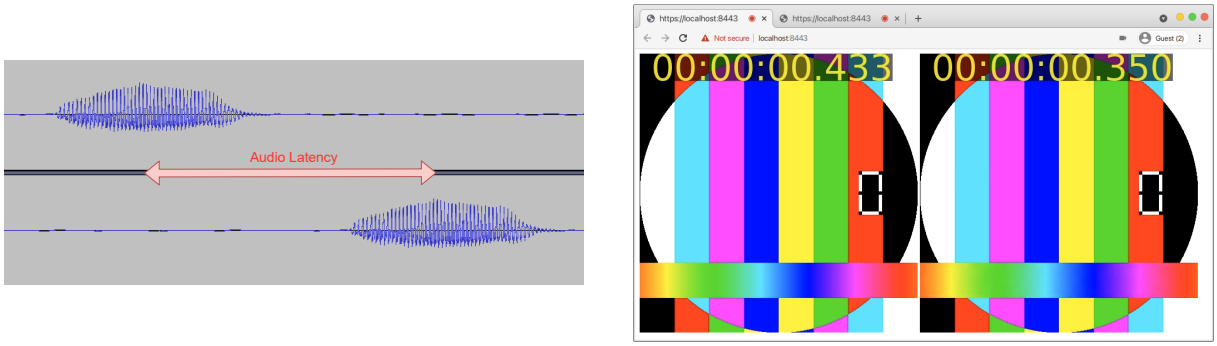


Figure 20: Example of how latency can be determined visually

Calculating E2E delay using WebRTC Statistics API

Another option to calculate E2E delay can be achieved by using the WebRTC Statistics API as discussed by Henrik Boström, an editor of said API draft, in a related GitHub ticket [47].

$$e2eDelay = currentReceiverNtpTimestamp - estimatedPlayoutTimestamp - \Delta NTP$$

The value `estimatedPlayoutTimestamp` is part of the specification and can be acquired from incoming remote streams. It returns the timestamp in sender NTP clock time for the last playable sample of the receiver.

However this approach might not be feasible if a server is relaying the media as described in the aforementioned ticket. Therefore we decided against this approach.

4.6.3 Controlled Media for Consistency

We leveraged Chrome's ability to regulate the media each client sends to create a controlled environment. Whilst Google Chrome has a built in test video with timestamps and frame numbers it is limited to 20FPS. For that reason we generated our own test video using FFmpeg using the following command:

```
# generates video sample, sets properties, adds timestamp, outputs .y4m
ffmpeg \
    -f lavfi\
    -i testsrc=duration=10:size=500x500:rate=60\
    -vf "drawtext=text='timestamp: %{pts \: hms}': x=0: y=0: fontsize=65:
        ↪ fontcolor=yellow@0.9: box=1: boxcolor=black@0.6" \
    -pix_fmt yuv420p test.y4m
```

The generated sample video has a resolution of `500x500` with 60FPS. This way the time between frames is 17ms instead of 50ms.

To use the generated video in Chrome the following start arguments were used:

- use-fake-device-for-media-stream
- use-file-for-fake-video-capture=test.y4m
- use-fake-ui-for-media-stream

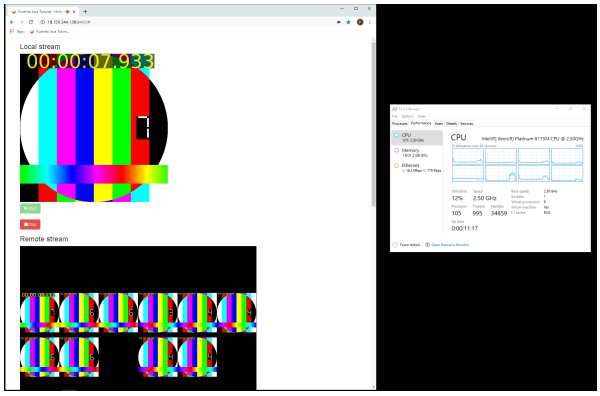
4.7 Test Methodology

The Chrome instances on the client generator VMs were started using Puppeteer in headless mode. One advantage of using browser based testing frameworks like Puppeteer and Selenium WebDriver is that concurrency is given out-of-the-box. Otherwise issues and limitations due to user profiles can occur in our experience.

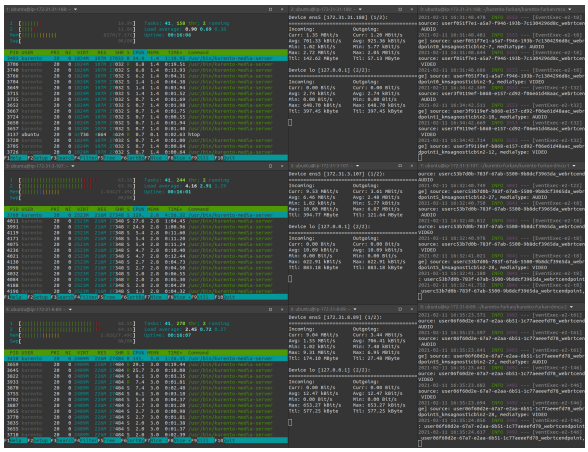
For each architecture we first start the media servers and the necessary applications. Then we connect the two participants that are used for measurement in the corresponding VM. Video participants are then generated by the core-rich m5.8xlarge instances. Between each new browser instance is a delay to accommodate for the ramp-up phase until a stable WebRTC connection is established.

For each new participant, screenshots of the resource usage of the measurement and media server VMs are taken as well as screenshots to determine the latency. Each measurement is the average of three screenshots.

In total more that 200 screenshots were taken. To aid with reading the values from the screenshots OCR (Optical Character Recognition) software was used. In retrospect a more automated approach despite the upfront cost could have been more time efficient.



(a) Client

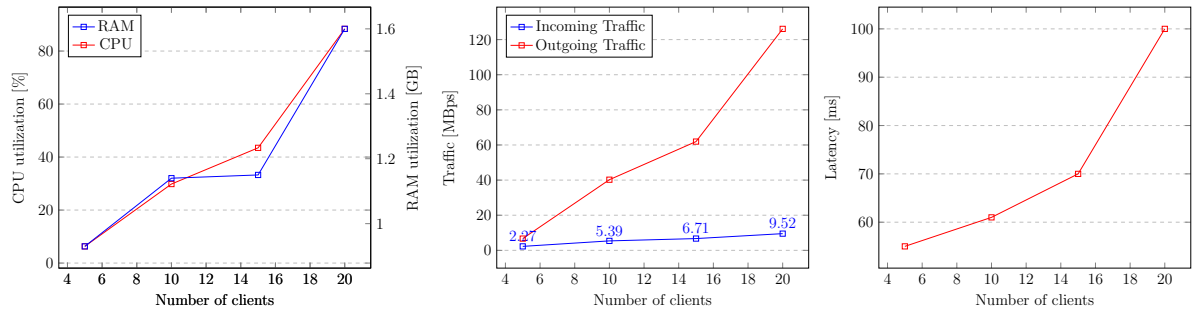


(b) Media servers

Figure 21: Example of screenshots used for measurement (DMCU)

5 Evaluation

5.1 SFU

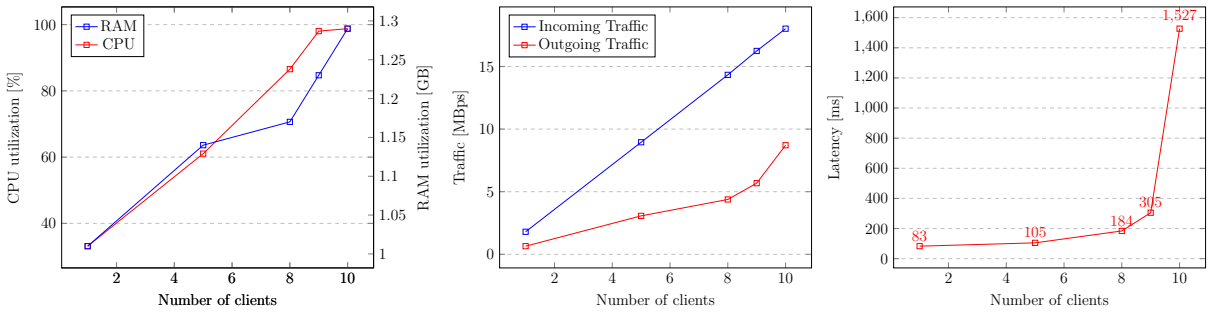


We argued that the amount of connections and traffic SFUs produce increases quadratic in an n-to-n scenario. Examining the graphs for the SFU test the outgoing traffic does not behave like a typical quadratic function. It is remarkable that the increase between 15 and 20 clients is higher than traffic that is required to reach 15 clients.

The relation between incoming and outgoing traffic is also visible, the outgoing traffic should theoretically be $n-1$ (see section 4.2) times larger than the incoming data. In this test the outgoing traffic is about $(n-1)*0.7$ times larger than the incoming traffic. We could not clearly identify the source of this discrepancy. Since after retrying the test with five clients the outgoing traffic was as expected $n-1$ times larger than the incoming traffic with a margin of error. We suspect that the bandwidth estimation component of the WebRTC video engine causes this discrepancy.

Furthermore, a correlation between outgoing traffic and CPU load is visible. The latency also increases with the number of clients. It is noteworthy that the latency increases rapidly once the media server reaches its CPU resource limits, this is especially visible on the following MCU test.

5.2 MCU



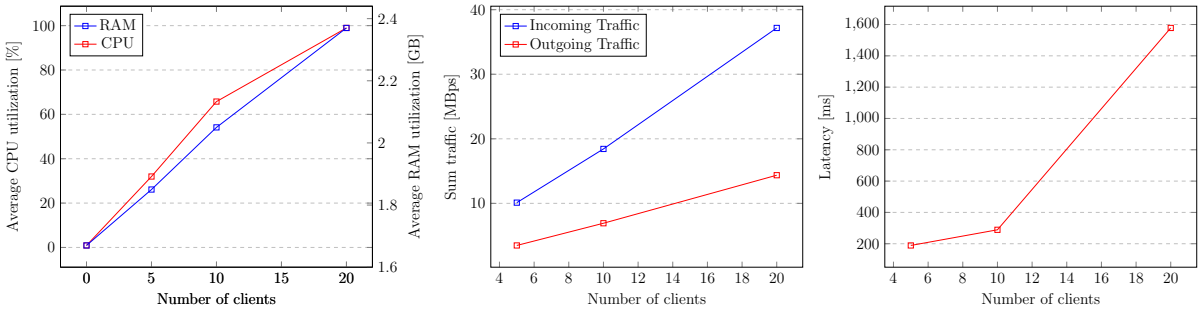
Looking at the graphs from the MCU test it is especially remarkable how linear the CPU load and incoming traffic scales with increasing client numbers. The outgoing traffic, even though it has some irregularities, seems to also scale linearly. The latency grows rapidly after the CPU load reaches 86% to a point where it exceeds the recommended 150 ms end-to-end delay for high-quality real time traffic [48].

Unlike the SFU test the incoming traffic is higher than the outgoing traffic. The reason why the outgoing traffic is smaller than the incoming traffic, instead of identical, is because

the outgoing media does not preserve the incoming media quality. Kurento Media Server's composite element has a fixed resolution of 800x600 and an FPS of 60 [41].

Comparing the values from the SFU and MCU the strengths and weaknesses can be recognized. Our media server with only 2 vCPUs can not serve as many clients in a single session when it acts as an MCU compared to when it is used as an SFU. On the contrary the network resources required for the MCU are significantly lower since MCUs scale linearly in this regard. Whilst the SFU scales better when the users are distributed among multiple sessions, MCUs scale best when its clients are distributed in as few sessions as possible. In our test the latency using an SFU was significantly better than the results achieved by our MCU.

5.3 DMCU



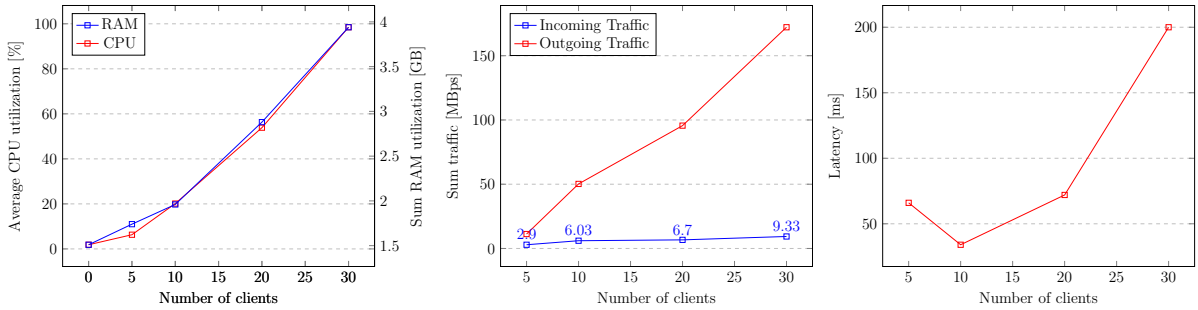
The DMCU values above only regard the two DMCU nodes that are not the root node. The values for the individual nodes including the root node can be found in the appendix. The reason why the root node is not included in the graphs is because it only has an auxiliary function and requires negligible resources. This is because the root MCU in this case only has two clients.

When we look at the statistics it is an achievement to see that the DMCU can handle twice as many clients as a single MCU in a session. Furthermore, the CPU, RAM as well as the incoming and outgoing traffic behave linear as hypothesized and the DMCU scales nearly perfect.

The values of the DMCU behave similar to the MCU values except for the latency. Although it is to be expected that the latency is higher when using two MCUs in series compared to one, the latency on the DMCU is on the verge of being usable at half of its maximum capacity. This is mostly due to the fact that the latency of all the MCUs that are in series are added. Because of that, this implementation might not be viable for use-cases like video conferencing.

This could change if another more efficient MCU would be used. At the time of writing we could not find any latency metrics for another MCU. The in section 4.5.2 proposed DMCU unit-2-unit topology could have had better latency performance due to the two composite elements being in the same pipeline in the same media server; we could however not test that due to a bug in Kurento Media Server. Nonetheless this topology is not well suited for latency critical applications. A MCU-SFU or MCU-P2P hybrid would likely perform significantly better than our proposed DMCU topologies since the delay of the second layer of MCU would be avoided.

5.4 DSFU



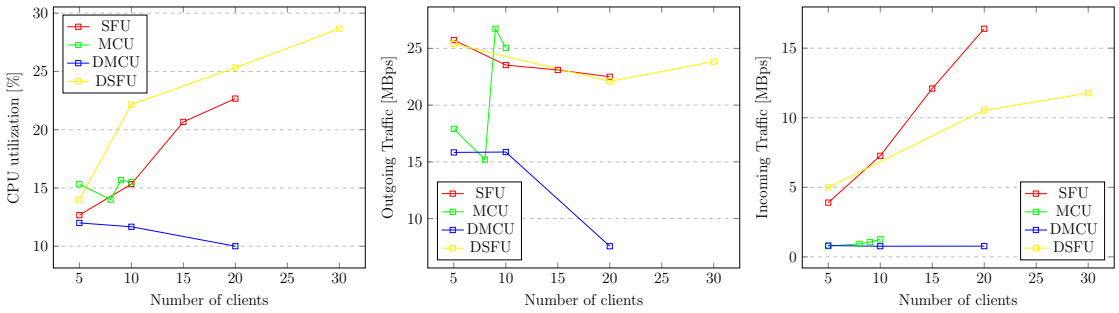
It is notable that the DSFU graphs are smoother than the ones of the SFU due to the fact that the graphs combine values from two SFUs. This shows that repeating tests could aid with reducing random deviations and getting more representative results.

The overall behaviour of our DSFU is similar to the one of the SFU. For example the ratio of incoming and outgoing traffic behaves similarly making the outgoing traffic significantly higher than the incoming traffic.

After the comparison between DMCU and MCU one might mistakenly expect that our DSFU should be able to handle twice as many clients as the tested SFU. The reason why this is not the case is because all clients are in a single session and therefore the load increases quadratically. The CPU load scales as expected when compared to the CPU load of a single SFU. The latency values are in the same range as that of a regular SFU. The DSFU architecture does not add additional latency like the MCU does.

Overall this topology scaled as expected and the test results do not show any disadvantage over a regular SFU.

5.5 Client-side Results



The client statistics behave for the most part as expected. The depicted values are taken from the same client and do not represent all clients. Typically the pairs SFU/DSFU and MCU/DMCU behave similar. The CPU load and incoming traffic when using an SFU/DSFU increases with the amount of clients. The MCU/DMCU does not show this behaviour, the CPU load and incoming traffic remains rather the same.

Unfortunately the readings we got for the outgoing traffic were unusable. The values are unreasonably high. For example the outgoing traffic from our client that was connected to an SFU with five clients was 25.73 MBps, for comparison the server had a total incoming traffic of 2.27 MBps. The presumption is that an undesired local loopback was recorded.

6 Conclusion and Outlook

6.1 Conclusion

We have successfully implemented a distributed SFU and distributed MCU. These approaches offer notable advantages, mainly the ability for larger sessions and better load distribution.

In order to achieve this, we first took a closer look at WebRTC with a focus on the important aspects of our research, namely signaling, connection establishment and audio and video processing and inspected related work. We then analyzed existing designs, used the gained knowledge to design our own models and discussed advantages and disadvantages of each model as well as their performance characteristics both for servers and clients/peers. A media server was carefully selected to implement said models and a test environment was formed. Finally the tests were being carried out and the results were evaluated. We analyzed the results, pointing out notable occurrences and checked for irregularities.

Transitioning from SFU to a DSFU could be achieved with already existing real-time communication solutions without modification of the SFUs themselves by adding an additional signaling server. We did not have to modify the client side for the DSFU.

The DMCU on the other hand requires modification since the MCUs need to communicate with each other. Therefore the MCUs require client functionality and in the case of our proposed unit-to-unit design further modification.

Besides the fact that our distributed units can achieve larger sessions than their regular counterparts it is also important to note that the distributed units also promote efficiency. This is because resource headroom of servers that would be used to compensate for dynamic workload increases is less relevant since resources of another server can be used. Therefore the distributed resources offer the ability to make use of otherwise unused resources.

6.2 Outlook

Several recommendations can be made for further research. Since we determined that models should be chosen depending on the use case, a solution that can switch between various models would be desirable. Existing solutions like Jitsi already switch from a mesh topology to an SFU topology depending on the amount of users [49], this could for example be expanded by the MCU model.

Also a design of a system that could utilize peers with a surplus of resources to aid weaker peers by mixing and forwarding or in other words acting as SFUs and MCUs themselves would be interesting. The broader goal is a system which adapts and morphs dynamically depending on the use case and available resources.

WebRTC load testing and video processing

We also suggest further research on load testing scalable WebRTC applications. The main issue of load testing a WebRTC application is the fact that video decoding and encoding are computationally heavy tasks. Accompanied by the fact that media servers are efficient and can handle many clients, it becomes difficult to do load testing that can bring existing implementations to its limits. We had to resort to very powerful 32 vCPU VM instances in order to load test SFUs/MCUs on low powered 2 vCPU instances.

One approach would be to modify the existing WebRTC implementation in a way that video processing can be toggled via an API. This way for example SFUs could be load tested with significantly less effort. In order to test a SFU with 20 clients each client needs to decode 19 incoming streams, although we do not gain any additional insight by decoding the same stream with multiple clients after receiving it.

Furthermore, if such an option would exist we would likely not have failed at building our own SFU since the main issue was bad performance due to the fact that we could not forward streams without unnecessarily decoding and encoding the same stream before forwarding.

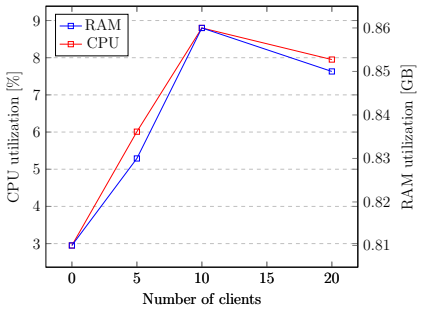
7 Appendix

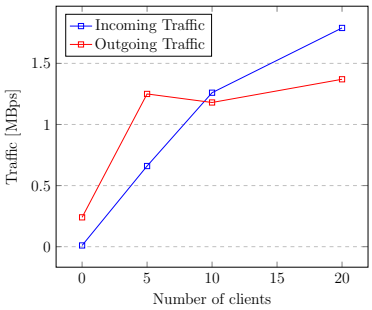
Source Code

The source code generated can be found at:
https://git.tu-berlin.de/furkantas/bachelor-source-code/

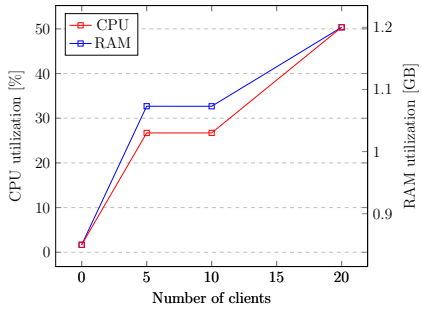
DMCU Individual Nodes

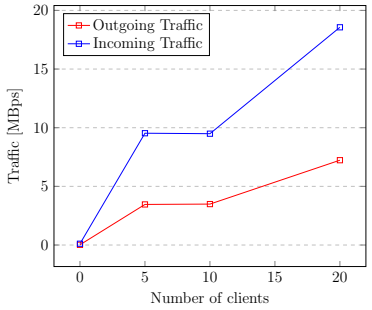
Root Node MCU



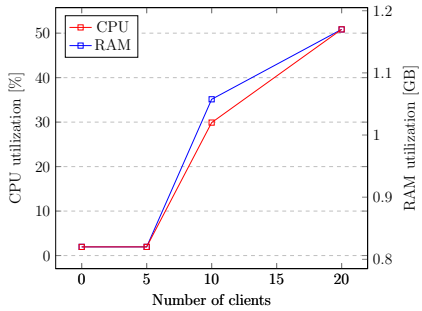


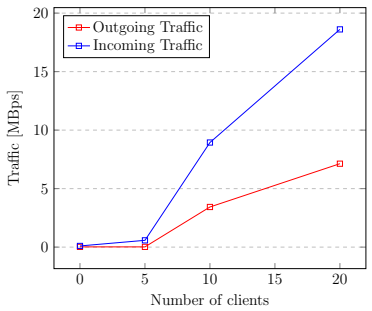
DMCU Node 2





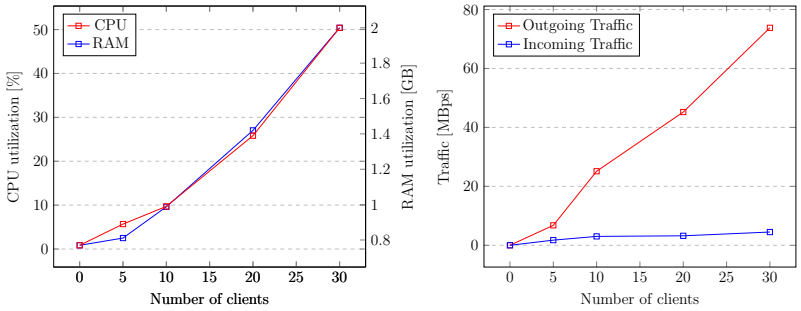
DMCU Node 3



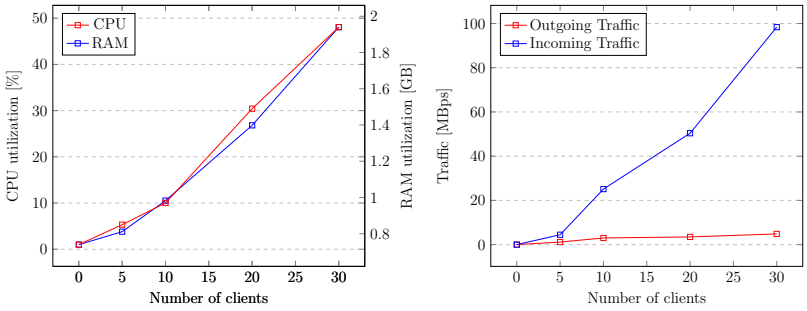


DSFU Individual Nodes

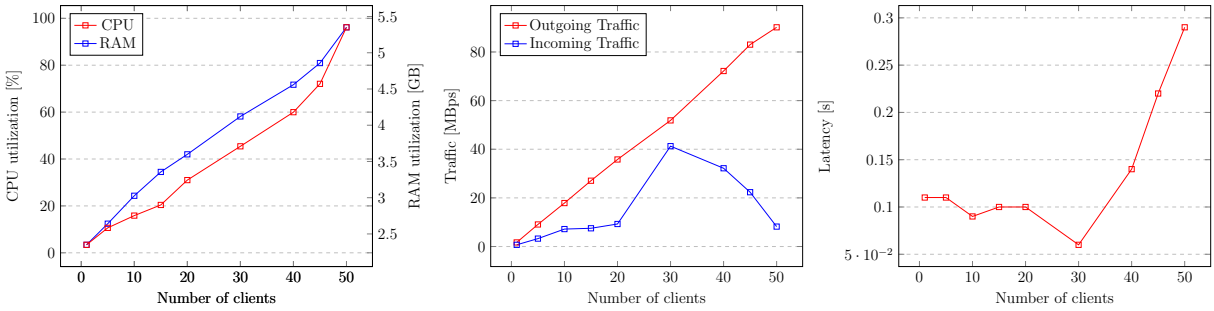
DSFU Server 1



DSFU Server 2



Large MCU Failed Test (m5.xlarge)



Server							Client					
SFU												
#Clients	CPU 1 [%]	CPU 2 [%]	RAM [GB]	IN [MBit/s]	OUT [MBit/s]		#Clients	CPU [%]	RAM [GB]	S [MBit/s]	R [MBit/s]	E2E [s]
5	6.20	6.40	0.93	2.27	6.70		5	12.67	2.50	25.73	3.90	0.06
10	29.93	29.70	1.14	5.39	40.23		10	15.33	2.53	23.53	7.27	0.06
15	44.23	42.73	1.15	6.71	61.93		15	20.67	2.70	23.10	12.10	0.07
20	88.77	87.97	1.60	9.52	126.10		20	22.67	2.80	22.50	16.40	0.10
							21	21.33	2.80	21.60	12.43	0.11
							22	19.33	2.80	19.20	10.43	0.18
MCU												
#Clients	CPU 1 [%]	CPU 2 [%]	RAM [GB]	IN [MBit/s]	OUT [MBit/s]		#Clients	CPU [%]	RAM [GB]	S [MBit/s]	R [MBit/s]	E2E [s]
1	33.03	33.20	1.01	1.80	0.65		1	12.00	1.80	4.40	0.69	0.08
5	60.83	61.27	1.14	8.95	03.07		5	15.33	1.90	17.90	0.50	0.11
8	86.50	86.60	1.17	14.33	4.38		8	14.00	1.80	15.20	0.93	0.18
9	98.03	98.03	1.23	16.24	5.68		9	15.67	1.80	26.73	01.06	0.31
10	98.87	98.70	1.29	18.02	8.71		10	15.50	1.80	25.05	1.27	1.53
large MCU (failed, m5.xlarge)												
#Clients	CPU [%]	RAM [GB]	IN [MBit/s]	OUT [MBit/s]			#Clients	CPU [%]	RAM [GB]	S [MBit/s]	R [MBit/s]	E2E [s]
1	3.41	2.35	1.68	0.70			1	13.33	1.80	14.00	0.69	0.11
5	12.42	2.58	09.07	3.25			5	15.33	1.80	23.50	0.82	0.11
10	24.33	2.75	17.87	7.17			10	13.67	1.80	18.43	0.83	0.09
15	34.49	2.90	27.06	7.50			15	18.00	1.80	21.97	0.97	0.10
20	41.99	3.24	35.79	9.26			20	14.67	1.87	26.73	1.12	0.10
30	58.15	3.70	51.86	41.27			30	13.50	1.90	24.58	1.63	0.06
40	71.69	4.18	72.17	32.18			40	14.67	1.90	26.80	1.23	0.14
45	80.92	4.57	83.02	22.33			45	14.67	2.50	24.37	1.37	0.22
50	96.16	5.35	90.14	8.19			50	13.00	2.10	22.10	1.23	0.29
DSFU												
#Clients	Server	CPU 1 [%]	CPU 2 [%]	RAM [GB]	IN [MBit/s]	OUT [MBit/s]	#Clients	CPU [%]	RAM [GB]	S [MBit/s]	R [MBit/s]	E2E [s]
0	1	0.70	0.70	0.77	0.00	0.01	0	1.00	1.80	0.03	0.03	-
0	2	0.00	0.70	0.74	0.00	0.01	5	28.00	1.90	25.40	5.00	0.07
5	1	4.00	3.97	0.89	1.75	6.77	10	44.33	2.20	24.70	10.87	0.03
5	2	5.60	6.00	0.85	1.16	4.47	20	50.67	2.30	22.10	10.53	0.07
10	1	18.23	18.30	0.99	03.02	25.12	30	57.33	2.30	23.83	11.77	0.26
10	2	19.03	18.63	0.97	03.01	25.11						
20	1	53.07	53.73	1.39	3.22	45.21						
20	2	51.63	52.30	1.49	3.48	50.38						
30	1	99.80	99.60	2.00	4.50	73.79						
30	2	94.13	96.37	1.94	4.83	98.36						

DMCU													
#Clients	Server	CPU 1 [%]	CPU 2 [%]	RAM [GB]	IN [MBit/s]	OUT [MBit/s]		#Clients	CPU [%]	RAM [GB]	S [MBit/s]	R [MBit/s]	E2E [s]
0	1	4.90	5.80	0.81	0.01	0.24		0	0.33	1.80	0.02	0.00	-
0	2	1.33	0.90	0.85	0.11	0.02		5	12.00	1.90	15.83	0.81	0.19
0	3	0.90	0.47	0.82	0.10	0.02		10	11.67	1.90	15.87	0.77	0.29
5	1	11.03	7.23	0.83	0.66	1.25		20	10.00	1.90	7.57	0.77	1.58
5	2	63.37	63.17	01.03	9.53	3.46							
5	3	0.90	0.47	0.82	0.58	0.02							
10	1	16.60	14.03	0.86	1.26	1.18							
10	2	63.33	63.50	01.03	9.49	3.49							
10	3	67.27	68.97	01.02	8.94	3.43							
20	1	14.90	15.43	0.85	1.79	1.37							
20	2	98.70	99.33	1.20	18.56	7.23							
20	3	98.70	99.10	1.17	18.61	7.13							

References

[1] J. M. Barrero, N. Bloom, and S. J. Davis, "Why working from home will stick," National Bureau of Economic Research, Working Paper 28731, Apr. 2021. DOI: 10.3386/w28731. [Online]. Available: http://www.nber.org/papers/w28731.

[2] (Dec. 18, 2017). "10 massive applications using WebRTC," BlogGeek.me, [Online]. Available: https://bloggeek.me/massive-applications-using-webrtc/ (visited on 09/29/2021).

[3] SerdarSoysal. (). "Beziehen von Clients für Microsoft Teams - Microsoft Teams," [Online]. Available: https://docs.microsoft.com/de-de/microsoftteams/get-clients (visited on 09/29/2021).

[4] P. Rodriguez, J. Cerviño Arriba, I. Trajkovska, and J. Salvachua, "Advanced video-conferencing services based on webrtc," Sep. 2021.

[5] A. S. Tanenbaum and M. van Steen, *Distributed systems*, 3rd ed. distributed-systems.net, 2017.

[6] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and web performance*, 1st edition. Beijing ; Sebastopol, CA: O'Reilly Media, Oct. 15, 2013, 400 pp., ISBN: 978-1-4493-4476-4.

[7] (). "Architecture — WebRTC," [Online]. Available: https://webrtc.github.io/webrtc-org/architecture/ (visited on 07/24/2021).

[8] (). "MediaTrackConstraints - web APIs — MDN," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MediaTrackConstraints (visited on 07/26/2021).

[9] (). "Codec - MDN web docs glossary: Definitions of web-related terms — MDN," [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Codec (visited on 07/26/2021).

[10] (). "Codecs used by WebRTC - web media technologies — MDN," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/WebRTC_codecs (visited on 07/26/2021).

[11] A. B. Roach. (). "WebRTC video processing and codec requirements," [Online]. Available: https://tools.ietf.org/html/rfc7742 (visited on 07/28/2021).

[12] C. Bran {\textless}cary.bran@plantronics.com{\textgreater}. (). "WebRTC audio codec and processing requirements," [Online]. Available: https://tools.ietf.org/html/rfc7874 (visited on 07/28/2021).

[13] (). "Signaling and video calling - web APIs — MDN," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling (visited on 07/28/2021).

[14] S.
bibinitperiod C. J. Nandakumar. (Feb. 23, 2013). "SDP for the WebRTC," [Online]. Available: https://tools.ietf.org/id/draft-nandakumar-rtcweb-sdp-01.html#rfc.section.3 (visited on 07/28/2021).

[15] Y. Wang, Z. Lu, and J. Gu, "Research on symmetric NAT traversal in p2p applications," in *2006 International Multi-Conference on Computing in the Global Information Technology - (ICCGI'06)*, Aug. 2006, pp. 59–59. DOI: 10.1109/ICCGI.2006.60.

[16] G. Maier, F. Schneider, and A. Feldmann, "NAT usage in residential broadband networks," in *Passive and Active Measurement*, N. Spring and G. F. Riley, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 32–41, ISBN: 9783642192609. DOI: 10.1007/978-3-642-19260-9_4.

[17] P. Francis and K. Egevang. (). "The IP network address translator (NAT)," [Online]. Available: https://tools.ietf.org/html/rfc1631 (visited on 07/28/2021).

[18] S. D. P. November 4th, 2. U. November 24th, and 2. C. 1. Y. b. m. n. s. t. f. i. t. article. (). "Build the backend services needed for a WebRTC app: STUN, TURN, and signaling - HTML5 rocks," HTML5 Rocks - A resource for open web HTML5 developers, [Online]. Available: https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/ (visited on 07/23/2021).

[19] J. Rosenberg and C. Holmberg. (). "Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal," [Online]. Available: https://tools.ietf.org/html/rfc8445 (visited on 07/28/2021).

[20] J. Rosenberg, R. Mahy, and D. Wing. (). "Session traversal utilities for NAT (STUN)," [Online]. Available: https://tools.ietf.org/html/rfc5389 (visited on 07/28/2021).

[21] P. Matthews, R. Mahy, and J. Rosenberg. (). "Traversal using relays around NAT (TURN): Relay extensions to session traversal utilities for NAT (STUN)," [Online]. Available: https://tools.ietf.org/html/rfc5766 (visited on 07/28/2021).

[22] (). "WebRTC 1.0: Real-time communication between browsers," [Online]. Available: https://www.w3.org/TR/webrtc/ (visited on 07/28/2021).

[23] *Coturn/coturn*, original-date: 2015-07-17T08:15:16Z, Jul. 28, 2021. [Online]. Available: https://github.com/coturn/coturn (visited on 07/28/2021).

[24] M. Khan, *WebRTC scalable broadcast*, original-date: 2014-11-12T04:18:43Z, Sep. 19, 2021. [Online]. Available: https://github.com/muaz-khan/WebRTC-Scalable-Broadcast (visited on 09/28/2021).

[25] (). "Internet speeds by country - fastest internet in the world map," [Online]. Available: https://www.fastmetrics.com/internet-connection-speed-by-country.php#median-internet-speeds-2020 (visited on 09/28/2021).

[26] ——, *WebRTC scalable broadcast*, original-date: 2014-11-12T04:18:43Z, Sep. 19, 2021. [Online]. Available: https://github.com/muaz-khan/WebRTC-Scalable-Broadcast/blob/dafab6c522c95eab51c8db18f529883a361d0af8/server.js (visited on 09/28/2021).

[27] (). "Peer connection relay," [Online]. Available: https://webrtc.github.io/samples/src/content/peerconnection/multiple-relay/ (visited on 09/28/2021).

[28] D. Conrads, *Datenkommunikation : Verfahren - Netze - Dienste; 3., überarbeitete und erweiterte Auflage*, ser. Moderne Kommunikationstechnik. Wiesbaden: Vieweg+Teubner Verlag, 1996, ISBN: 3-528-24589-1. DOI: 10.1007/978-3-322-91973-1.

[29] (). "Measuring WebRTC video quality for different bitrates - playing with VMAF," [Online]. Available: http://www.rtcbits.com/2018/10/measuring-webrtc-video-quality-for.html (visited on 09/28/2021).

[30] AGOUAILLARD. (Oct. 11, 2018). "#WebRTC video quality assessment," WebRTC by Dr Alex, [Online]. Available: https://webrtcbydralex.com/index.php/2018/10/11/webrtc-video-quality-assessment/ (visited on 09/28/2021).

[31] S. Petrangeli, D. Pauwels, J. van der Hooft, M. Žiak, J. Slowack, T. Wauters, and F. De Turck, "A scalable WebRTC-based framework for remote video collaboration applications," *Multimedia Tools and Applications*, vol. 78, no. 6, pp. 7419–7452, Mar. 1, 2019, ISSN: 1573-7721. DOI: 10.1007/s11042-018-6460-0. [Online]. Available: https://doi.org/10.1007/s11042-018-6460-0 (visited on 09/28/2021).

[32] Y. A. Ushakov, M. V. Ushakova, A. E. Shukhman, P. N. Polezhaev, and L. V. Legashev, "Webrtc based platform for video conferencing in an educational environment," in *2019 IEEE 13th International Conference on Application of Information and Communication Technologies (AICT)*, 2019, pp. 1–5. DOI: 10.1109/AICT47866.2019.8981724.

[33] (Jun. 26, 2019). "How zoom provides industry-leading video capacity," Zoom Blog, [Online]. Available: https://blog.zoom.us/zoom-can-provide-increase-industry-leading-video-capacity/ (visited on 09/28/2021).

[34] M. Iyengar. (Mar. 10, 2021). "WebRTC architecture basics: P2p, SFU, MCU, and hybrid approaches," SecureMeeting, [Online]. Available: https://medium.com/securemeeting/webrtc-architecture-basics-p2p-sfu-mcu-and-hybrid-approaches-6e7d77a46a66 (visited on 09/28/2021).

[35] B. Garcia, L. Lopez-Fernandez, M. Gallego, and F. Gortazar, "Kurento: The swiss army knife of WebRTC media servers," *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 44–51, 2017, ISSN: 2471-2833. DOI: 10.1109/MCOMSTD.2017.1700006.

[36] B. Garcia, L. López, F. Gortázar, M. Gallego, and G. A. Carella, "Nubomedia: The first open source webrtc paas," in *Proceedings of the 25th ACM international conference on Multimedia*, 2017, pp. 1205–1208.

[37] *Node-webrtc/node-webrtc*, original-date: 2013-07-02T18:21:45Z, Jul. 30, 2021. [Online]. Available: https://github.com/node-webrtc/node-webrtc (visited on 07/23/2021).

[38] (). "Licode," [Online]. Available: https://lynckia.com/licode/ (visited on 07/22/2021).

[39] (). "Is erizo really a MCU, not a router/relay? · issue #170 · lynckia/licode," GitHub, [Online]. Available: https://github.com/lynckia/licode/issues/170 (visited on 07/22/2021).

[40] L. Lopez, R. Vlad, I. Gracia, F. López, M. París, S. Carot, B. García, M. Gallego, F. Gortázar, R. Mejías, J. Santos, and D. Fernández, "Kurento: The WebRTC modular media server," Oct. 1, 2016, pp. 1187–1191. DOI: 10.1145/2964284.2973798.

[41] E. André, N. Le Breton, A. Lemesle, L. Roux, and A. Gouaillard, "Comparative study of WebRTC open source SFUs for video conferencing," in *2018 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, Oct. 2018, pp. 1–8. DOI: 10.1109/IPTCOMM.2018.8567642.

[42] (). "Kurento/kurento-tutorial-java," GitHub, [Online]. Available: https://github.com/Kurento/kurento-tutorial-java (visited on 07/23/2021).

[43] (). "Spring boot," [Online]. Available: https://spring.io/projects/spring-boot (visited on 07/22/2021).

[44] (Jan. 9, 2017). "WebRTC RTCPeerConnection. one to rule them all, or one per stream? • BlogGeek.me," BlogGeek.me, [Online]. Available: https://bloggeek.me/webrtc-rtcpeerconnection-one-per-stream/ (visited on 07/22/2021).

[45] (). "WebRTC in firefox 38: Multistream and renegotiation – mozilla hacks - the web developer blog," Mozilla Hacks – the Web developer blog, [Online]. Available: https://hacks.mozilla.org/2015/03/webrtc-in-firefox-38-multistream-and-renegotiation (visited on 07/22/2021).

[46] (). "Composite (kurento client API)," [Online]. Available: https://doc-kurento.readthedocs.io/en/latest/_static/client-javadoc/org/kurento/client/Composite.html (visited on 07/22/2021).

[47] (). "End-to-end delay metrics · issue #537 · w3c/webrtc-stats," GitHub, [Online]. Available: https://github.com/w3c/webrtc-stats/issues/537 (visited on 09/29/2021).

[48] "Transmission systems and media digital systems and networks," ITU-T, Recommendation G.114, 2003.

[49] (). "Implementing p2p-SFU transitions in WebRTC," [Online]. Available: http://www.rtcbits.com/2019/04/implementing-p2p-sfu-switching-in-webrtc.html (visited on 09/29/2021).