BBVOICE

A STREAMING TREE APPROACH TO WEBRTC VIDEO CHATTING

STEFAN PRISCA, 201703265

IOT FINAL PROJECT

December 2017

Advisor: Niels Olof Bouvin

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

BBVOICE

STEFAN PRISCA

A streaming tree approach to WebRTC video chatting

IoT Final Project
Department of Computer Science
Science & Technology
Aarhus University

December 2017

ABSTRACT

It has never been easier to communicate with one another than it is these days. Internet communications have created a whole new way of interacting with people from afar. However, most of the popular communication apps have centralized servers that direct the traffic (Facebook, Google Hangouts, etc).

This project proposes a decentralized solution for real life communications over the internet, using only one's browser. The advantage of this is that people can have a more secured and trustworthy communication channel over the internet by opening a direct (and possibly secured) stream between their computers. Moreover, the report proposes a different architectural approach to chat system, which uses streaming trees instead of mesh based networks. This leads to a more configurable and dynamic network.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 ONLINE VIDEO CHATS

It has never been easier to communicate with one another than it is these days. Internet communications have created a whole new way of interacting with people from afar. Everybody has Facebook, and we all know how easy it is to open a chat window and talk to one of your friends. There are also a variaty of applications that come directly installed on your computer (e.g. Skype for Microsoft Windows).

However, most of the popular communication apps have centralized servers that direct the traffic (Facebook, Google Hangouts, etc). This means that the messages exchanged in the chat first pass through a centralized server component, and only then they get forwarded to the receipient. For most use-cases, this does not create any issues, as chat participants are kept agnostic of what is going on behind the scenes. But the centralized server component presents a couple of disadvantages:

- **Scalability issues**: as mentioned before, for normal use cases this is not a problem. But it is something that the service provider needs to keep in mind. There might be periods with a lot of traffic (lots of people chatting). And in this cases, the servers need to scale up to support it, which means increased costs for the providers.
- **Security issues**: having a centalized component that contains all the data is a big issue when it comes to security. This creates a single attack point for mallicious software to corrupt the system and there have been numerous attacks of this sort.TODO
- **Privacy issues**: having all your messages pass through a server before arriving to their destination raises some concerns about the level of privacy one has in online chats. It is very easy to analyse messages and conversations at the server side, and there have been a lot of privacy breaching cases which resulted in the provider being sued by its clients.

1.2 DISTRIBUTED P2P CHATS

An alternative that might provide solutions to the above problems is peer to peer (P2P) communication. This means getting rid of the

server component from the middle, and having a direct communication link between the chat participants. The computational cost of encoding, sending, receiving and decoding messages is supported by each peer individually.

It means that a system such as this is way more scalable than the server-based systems, since each participant handles its part of the trafic. The more peers connect to the system, the more the system's computational power grows.

Moreover, since all computations and data are handled on different independent machines, there is no one central component that knows about everything. Thus there is no single attack point, making it harder for attackers to damage the system. This is one of the perks of having peer to peer networks.

Finally, the communication link between peers can be encrypted (e.g. a https connection). This greatly improves data privacy, as messages travel through an ecrypted secured connection from the source to the destination.

However, most of the existing p2p communication networks use a mesh based architecture. This means that all peers are interconnected to one another. The complexity of such network grows exponentially. This prevents scaling the system to support chat groups with a large number of people, limiting the chat size to that supported by the peers with lowest bandwidth.

A proposed solution for this problem is to use a streaming tree based architecture that introduces several scalability options. In such a system the complexity is reduced to a linear function, and peers with lower bandwidth can listen only without having to send data to everyone else. And peers with higher bandwidth available can do more work. This has the potential of creating a more balanced system, where data is distributed among the combined bandwidth of all peers.

This idea of using streaming trees will be further discussed throughout the next chapters. The experimental results show that it is a viable alternative to mesh-based systems, and that it can be improved to allow for more scalable p2p chat systems.

mode:flyspell ***

2 RELATED WORK

The first application that comes in mind when thinking about P2P Voice over IP (VoIP) communications is Skype. Salman A. Baset and Henning Schulzrinne [3] present a overview of the prodocol used by the Skype application. The structure of Skype's network is made out of the following components: ordinary nodes, super nodes and a login server. In this system, supernodes play the role of entry points for ordinary nodes. There is also a central server component that is responsible for the login functionality. The authors then analyse how skype clients establish a connection. They run experiments with clients on public web addresses or behind NAT and firewalls. It is interesting to note that peers exchange a series of signaling messages through a third online node that acts as a relay.

The authors of [5] present a thorough comparison between mesh based streaming and streaming trees. They describe how the two layouts are built, and how data packages are send between nodes in both systems. They conclude that mesh-based systems have a better performance compared to tree-based systems. The motives for this are that content is statically mapped to a certain tree, and the placement of nodes as internal in some trees and leafs in others.

2.1 FRAMEWORKS AND TECHNOLOGIES

Moving to more modern technologies, the authors of [4] present a realtively new standard for real time communications in the web: WebRTC. They define an API that allows peer to peer (P2P) communications directly from the browser, while using standart IETF protocols for navigating through NATs and Firewals (STUN and TURN servers). Furthermore, WebRTC includes various features for program developers, giving them easy access to media devices and media objects in the browser. The authors then present the overall architecture, functionality and components of this system. It is noticeble that WebRTC uses a central server component for signaling purposes. In order for two peers to establish a connection, just like in the case of Skype, these need to exchange a series of messages to negociate NAT and Firewall traversals. For this step, WebRTC uses a dedicated signaling server. After this step, a PeerConnection is established between the two browsers. This represents a direct and secured link between the browsers through which peers can exchange data streams (MediaStream objects). Another important component of WebRTC is the

DataChannel. This is a generic service for exchaning data between browsers in a stream-like fashion.

WebRTC has gained a lot of popularity lately, being used in various streaming applications like [6], [1] or [2]. All these applications present different solutions to video streaming over the internet in a distributed p2p fashion.

3 ANALYSIS

Building a chat application presents several differences compared to media streaming applications. The first one is that video/audio data is not available on one of the nodes. This is generated live, and needs to be transmitted and received by all peers as soon as possible. This is a constraint that must be considered at the architectural level. The second difference is that, unlike a video streamming application where data is streamed from one node to another knowing at each time which node is the source, in chat applications all nodes produce data. All nodes have their own data that they need to stream, and all nodes need to receive data from others as soon as possible. Because of this, traffic in chatting applications can become a serious problem when having an increased number of peers.

The first of these challenges is solved by WebRTC. The API provides easy access to user media (camera / microphone), and allows this to be captured as a MediaStream that can be sent over a PeerConnection. For this reason, the solution proposed in the next chapters is built using the WebRTC platform.

Furthermore, to address the issue of managing bandwidth, the comparison charts presented by [5] are analysed. Although the mesh system presented in the paper is not a full mesh, but a random mesh, it is noticeable that the tree based systems bandwidth consumption is lower. What is important to the analysis is that the bandwidth patterns appearing are all logarithmic functions (which is expected from a tree structure). Given this observation, the poposed solution is to minimize bandwidth consumption by employing a multi stream tree based architecture.

It is also worth mentioning that peers with lower bandwidths can act as bottlenecks in full mesh chat systems. Since these can handle restricted amounts of data, it might ruin thechat experience for themselves as well as others (delayed transmissions from the weak peers). The proposed stream tree solution also opens the possibility to reballance trees base on the streaming capabilities of each peer. Thus allowing the system to self-organize for best overall experience, where weaker peers can appear more often as leaves and less frequently as internal nodes.

The proposed system is discussed with more details in Chapter 4 below.

4

DESIGN AND METHODOLOGY

In this chapter the streaming tree approach to p2p video chatting is presented in more details. As mentioned in the previous chapter, the design is based on the WebRTC framework. Although this is intended for p2p communications, WebRTC requires a server component that handles messages exchanged between peers during the signaling phase. Based on this requirement, the application can be somewhat divided into two separate components:

- Server component: lightweight message broker that enables peers to created the necessary connections.
- Client component: client's browser. This is the level at which WebRTC runs, and where peers communicate with one another.

In order to achieve a streaming tree topology for the network, the signaling phase needs to be altered a bit. More precicesly, the server needs to route messages such that the PeerConnections that are created between WebRTC peers form streamming trees between them. Moreover, on the client side (in the browser), we need to define two different types of connections: *incoming* and *outgoing*. In order to better understand the solution, the next section explains the logic behind the client side. After that, the server side workflow is presented.

4.0.1 *Client side: WebRTC in the Browser*

As mentioned in previous chapters, WebRTC is built into the browser. This is a client-side technology, that runs as part of each client's browser. In order for a peer connection to be established, WebRTC peers need to exchange a series of messages (see 1), and one of the peers needs to initiate the connection. There are two types of such messages:

- Session Description Protocol messages (SDP): describe the media that is being shared between peers. This includes information about how the video is going to look like what encryption is being used, how much bandwidth is available, etc. SDP consists of a pair of offer/answer messages. The initiator will send and offer message to the other (receiver). Upon receiving the offer, the receiver will reply back with an answer.
- Internet Connectivity Establishment (ICE): After the SDP exchange is successful (peers know what they will send and receive), a serise of ICE messages is being sent. The purpose of

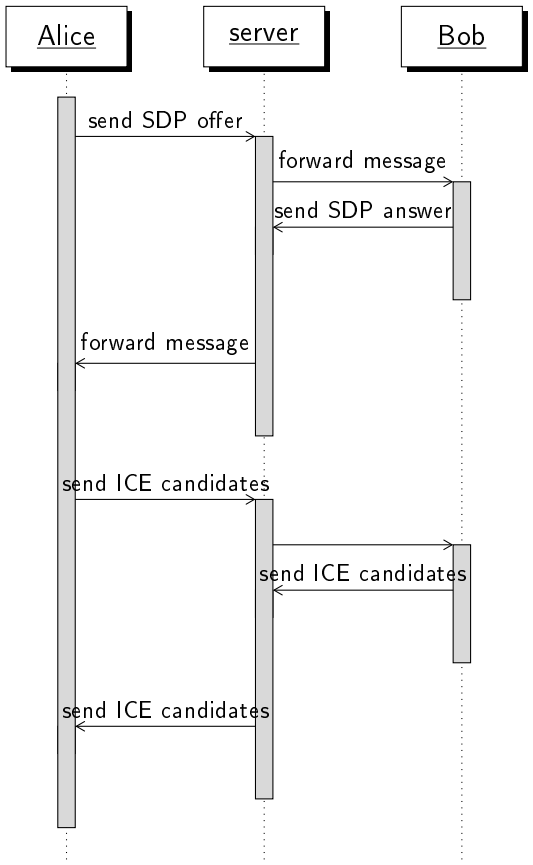

Figure 1: Sequence diagram showing a high overview of SDP and ICE message exchanges between WebRTC peers

this is for the peers to find a common path for traversing NAT addresses and Firewalls.

At the end of this interaction, a PeerConnection is created. This represents a bidirectional streamming channel between the two peers.

Streaming trees require a bit more special handling of PeerConnections. The proposed solution involves having one output streaming tree for each peer in the network, and each peer has a streamming peers in all the other trees. The conceptual diagram of a single peer can be seen in 2.

In order to achieve this architecture using WebRTC, two types of PeerConnections must be considered:

- Incoming Connection: A peer receives streams for other peers through incoming connections. In a system with n peers, each peer will have exactly $(n - 1)$ such connections (one from each other peer in the network).
- Outgoing connections: A peer sends data out through outgoing connections. The number of these is variable. Each peer needs to have at least b such connections (where b is the branching factor of the streaming trees), and at most $b*(n-1)$.

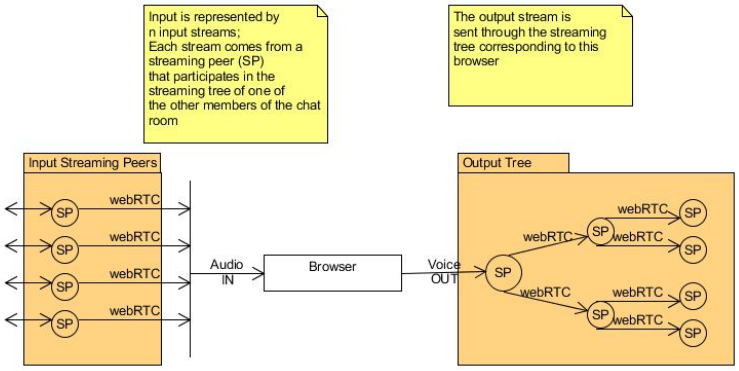

Figure 2: Streaming trees in the browser

In order to make WebRTC initialization step work with two types of connections, some rules have to be made as to who is responsible for initializing connection. Thus PeerConnections will always be initialized only by peers who want to sending data, and never by peers who receive data. Considering two peers Pi and Pr, the SDP initialization step for the proposed system can be described as follows: 1) Pi creates an outgoing connection for Pr, and sends an offer to Pr. 2) Upon receiving the offer, Pr creates an incoming connection for Pi and sends back the answer. This approach mimics the data flow of a streaming tree, where new nodes join by sending a message that they want to receive that data, and then are contacted by a node in the tree that is available to stream to them.

4.0.2 *Server side*

Knowing what kind of interaction needs to happen on the client side to establish p2p connections, makes it easier to understand what logic needs to be implemented on the server to support it. In the proposed solution, the server acts as a routing mechanism that creates the streaming tree layouts by controling the exchange of WebRTC messages. For this, it needs to maintain a model of the connections that are curently active.

The server then cares only about three evens in the network:

- A peer joins: at this point, the server is responsible for 1)finding a spot for this peer in all the other streaming trees, and 2) creating a new streaming tree for the peer itself. For step 1), *newcomer* messages will be sent to all the chosen parents, notifing them that they should initialize an outgoing connection on that tree with the new peer. For step 2), the server assigns the new

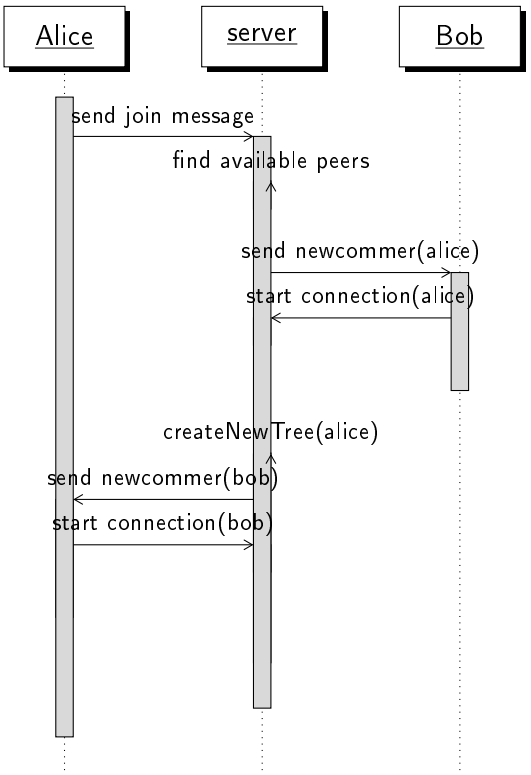

Figure 3: Sequence diagram showing an overview server behavior upon a node joining the network

node as the root, and repets step 1) on the new tree for each of the existing peers. This behavior can be seen in 3.

- A peer leaves: The server removes the node from all trees, by sending a *bye* message to all nodes that had a connection to it. Then, for each node that is left orphaned it applies step 1) from above.

- WebRTC Signaling Messages: The sever can remain realtively agnostic to this phase. Once the *newcomer* message is sent to one of the peers, this starts to initialize the connection. All messages that are exchanged during the initialization contain information about both the sender and the destination of the message. Thus, the server only needs to forward messages to the appropriate peers.

At the end of this interaction, Alice will be added to Bob's streaming tree, and a new tree containig Bob will be built for Alice. If Bob and Alice were the only participats, there would be only two trees of the form *Alice » Bob* and *Bob » Alice*.

In the continuing chapters the exact implementations details are presented, together with the chosen environments and programming languages. Furthermore, the **??** chapter presents some metrics on how this system behaves in a real application.

5 IMPLEMENTATION

Similar to the design seen previously, the implementation is also split in two major components: *client side* and *server side*. Note that these are not the traditional client-server components of a distributed system. The server's only purpose here is to route messages between clients such that these create the appropriate WebRTC PeerConnections. After this signaling pahse, the server is no longer needed, as clients now have direct p2p communication links between them. The contents of this chapter present the implementations of both these parts. First the client side with WebRTC is detailed, showing the way WebRTC's signaling part works. After that, the server side and the technologies used there are explained.

5.1 CLIENT SIDE: WEBRTC

As seen in previous chapters, WebRTC is a browser technology. For this reason, its API is written in Java Script and requires basic kwnoledge of the language. The official documentation of this API can be found on the MDN docs pages. Parts of the API that have been used in this project are explained below (together with parts of the Web Media API). There are a lot more methods in the API that are not covered here.

- MediaStream: Web media object representing a stream of various media contents. The stream consists of multiple MediaTracks, which can be added or removed from it.
- getUserMedia(): gets hold of the curent user's media devices (camera / microphone), and creates a MediaStream object from them. The MediaStream object is then pased to a callback method for the developer to use it.
- RTCPeerConnection: probably the most important part of the API. This represents a bidirectional streaming connection between two browsers. Once this is established, the two browsers can exchange their MediaStreams or other data contents (via a DataChannel).
- RTCPeerConnection.createOffer()/.createAnswer(): These are the methods that create offers and answers for the SDP package exchange that takes place during the signaling phase.

- RTCPeerConnection.setLocalDescription/.setRemoteDescription: Locally created SDP offers/answers are set as local descriptions. Received offers/answers are set as remote descriptions.
- RTCPeerConnection.addStream(): adds the given MediaStream to the current connection.

As mentioned in the design chapter, clients need two different tyeps of PeerConnections: incoming and outgoing. Given this requirement, the client component of this project is structured in the following way (**??**):

- IncommingConnectios: the input connections through which a specific client receives data from others. These connections provide the streams that this client has to display on it's page, and might have to propagate further on the output connections
- VideoElements: HTML video elements where the received stream are displayed.
- Streams: a collection of streams identified by the tree from which they originated. These streams are captured through IncommingConnections, displayed into the VideoElements and output through the outgoing connections.
- Outgoing connections: Connections through which the streams are passed to the other peers of the network.

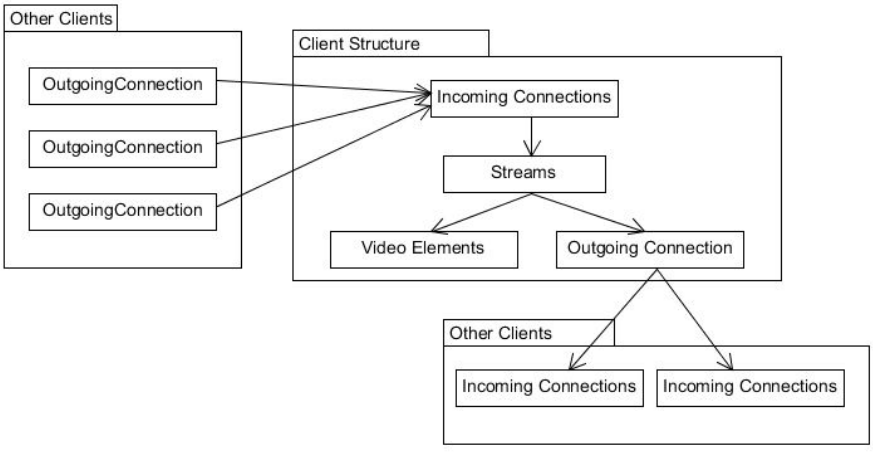

Figure 4: Streaming trees in the browser

Since incoming and outgoing connections are defined as unidirectional in this case, and the RTCPeerConnection is a bidirectional connection, the receiver of an offer needs to send back a dummy empty MediaStream. Otherwise WebRTC won't create a connection between the two peers.

Moreover, in order to create the WebRTC connections, a series of messages needs to be exchanged. This can be thought of as a state machine, where the two peers move from one state to another until they are connected. To achieve this, messages are exchanged between the peers and the server. The message structure is relatively simple: {senderId, treeId, destination, message}. The *senderId* identifies who sent the message. The *treeId* identifies the tree under which the message is sent. This is useful for knowing what streams to add/send, etc. *destination* represents whom the message targets and *message* represents the message itself. The messages exchanged are the following (chronological order in Figure 5):

- Create or Join: The first message sent by a peer to the server. If it's the first chat participant, it is considered that the client creates the chat.
- Join/Created: Server tell the client that it either joined or created the chat.
- Got user media: Client sends this message when it captured the user's media element. Clients that receive this message from the server have to prepare to receive an offer from the peer who got the user media (the one who sent this to the server).
- Newcommer: Server sends this message to announce clients that they need to start sending their stream from the corresponding tree to a new peer.
- WebRTC Messages: various messages that contain SDP offers/answers or ICE candidates.

Note that in figure 5, the startRTCSignaling represents the WebRTC signaling process throught which SDP and ICE messages are exchanged to created the communication between them. The direction of the call in the diagram represents who is the initiator of this connection (the one who sends data). The *tree* makred on the diagram is the identifier of the stream that the peer needs to send. In this case, when Alice receives *newcommer(bob, alice)* it interprets the message as "Bob is a newcommer in tree Alice". Therefor, it will initialize an outgoing connection to bob and send it the stream from tree *alice*.

The above shows how clients are structured in this application. Next, the server and the technologies used to make this work is explained in more details.

5.2 SERVER-SIDE

Recall that the server needs to maintain a mapping of all the trees in the application, and route message to create the appropriate connections. To implement this server, the Google Go programming lan-

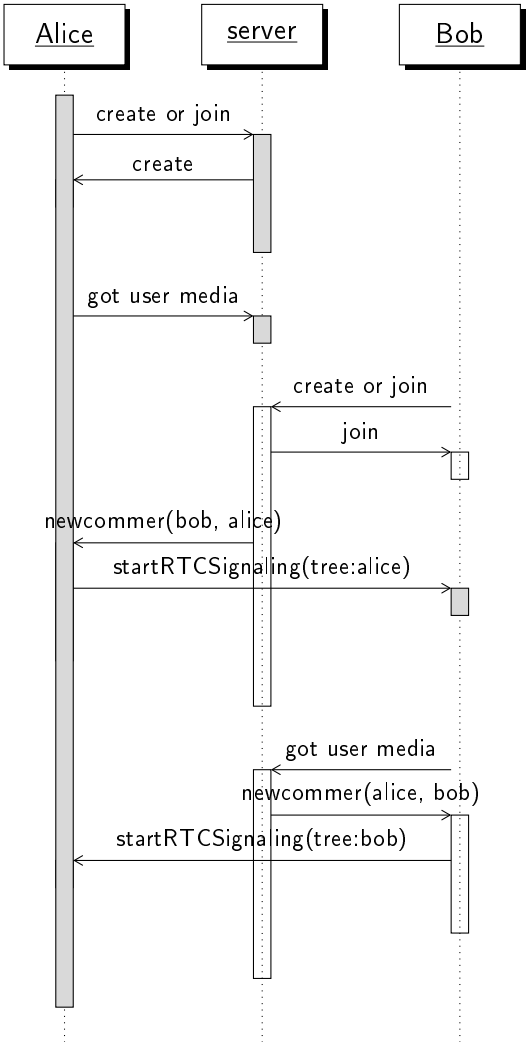

Figure 5: Sequence diagram showing the messages exchanged between peers and the server

guage has been used (see the Golang homepage). This is a very ligh-weight and powerful programming language, that has parallelilsm built into itself. This power comes from the so called *goroutines*, which are threads created and managed by the runtime environment, and from go's *data channels* which are communication channels between the routines.

An overview of the server side components can be seen in fig. 6. The main component here is the *Router*. This is responsible for maintaining the structure of the streaming trees, and to route messages between peers based on these trees. It holds a local copy of the trees, represented as a *StreamingMap*. For ensuring data consitency during cuncurrent access from different routines, the *StreamingMap* contains a Lock element. The router will lock on this each time it processes a message, and releases the lock in between messages. *StreamingMap* also contains the list of *Streamer*s. Each *Streamer* is an active peer in the system. Peers are identified by a string, which is also the root for their *StreamingTree*. Therefor, each peer has a streaming tree associated with it, which then contains the ids of all other peers in it (the children). Besides the tree, streamers also hold a mapping of which streams they have received and are ready to send. This is a synchronization measure to ensure that peers are not asked to send a stream before having it themselves. Moreover, to allow for effective internal communications, each streamer has a dedicated golang data channel (*Channel*)

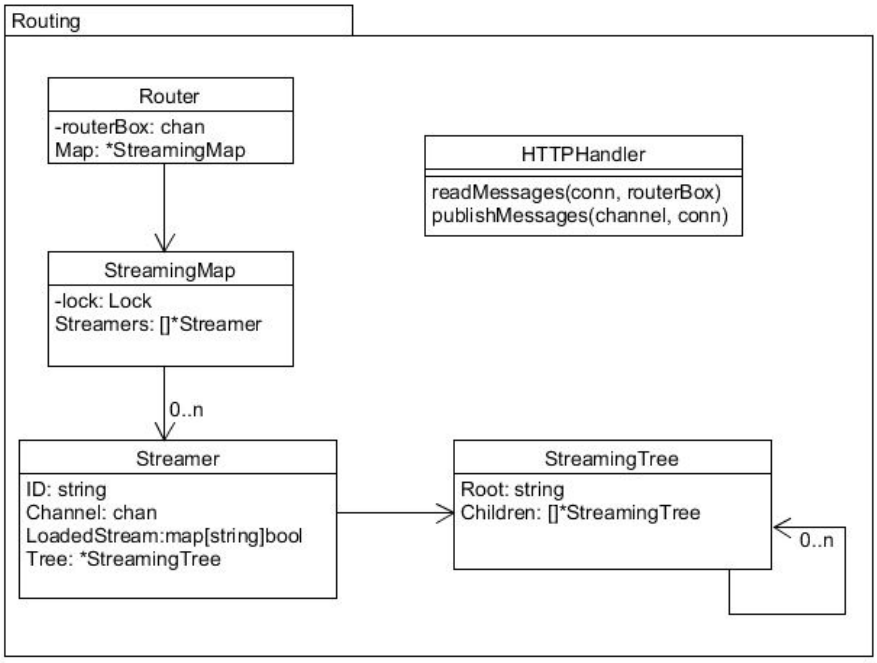

Figure 6: Streaming trees in the browser

Communication with the Java Script clients is achieved through the *HTTPHandler* component. This has two basic methods: read and write

(publish). When a client joins the network, the *HTTPHandler* opens a secured socket connection (through ssl) with it, and start a goroutine (background thread) to read messages received from the client (the *readMessage* routine). This routine will read incomming messages from the websocket, and place them on the *routerBox*. After this, the *HTTPHandler* starts another routing that publishes messages to the client connection. These messages are received through the internal server communication channels (data channels).

Internally, the server runs several go routines that communicate with each other. As seen in the previous paragraph, the *HTTPHandler* starts two go routines for each peer. There is another bigger routine, which is the *Serve* routine. This waits for messages to be published on the router's *routerBox* channel. Upon receiving a message, the router will make the necessary actions to handle it. In order to then send messages to the peers, the router will publish this message to the corresponding peer's data channel. This is then captured by the *HTTPHandler*'s go routine listening on that channel, and published on the web socket corresponding to the peer. This interaction is illustrated in figure 7 below.

Holding a mapping of the actual connections on the server side allows it to control the connections, and ajust them in order to achieve best bandwidth utilization as mentioned in the previous chapters. The current implementation does not do this, but it leaves space for improvements. The *StreamingTree* logic is separated from the actual routing through a simplistic API consisting of three methods: newTree, addChild, deleteChild. It is possible that when bandwidth information is avalible, the logic inside addChild is changed to take this into account. The idea would be that the addChild method should always prefer parents with the maximum available bandwidth. Then when a new node with more bandwidth joins the tree, the connections can be re-made such that the node with higher bandwidth is responsible for more trafic. The next chapter presents the experimental results of the curent implementation, leading to the conclusion of this report.

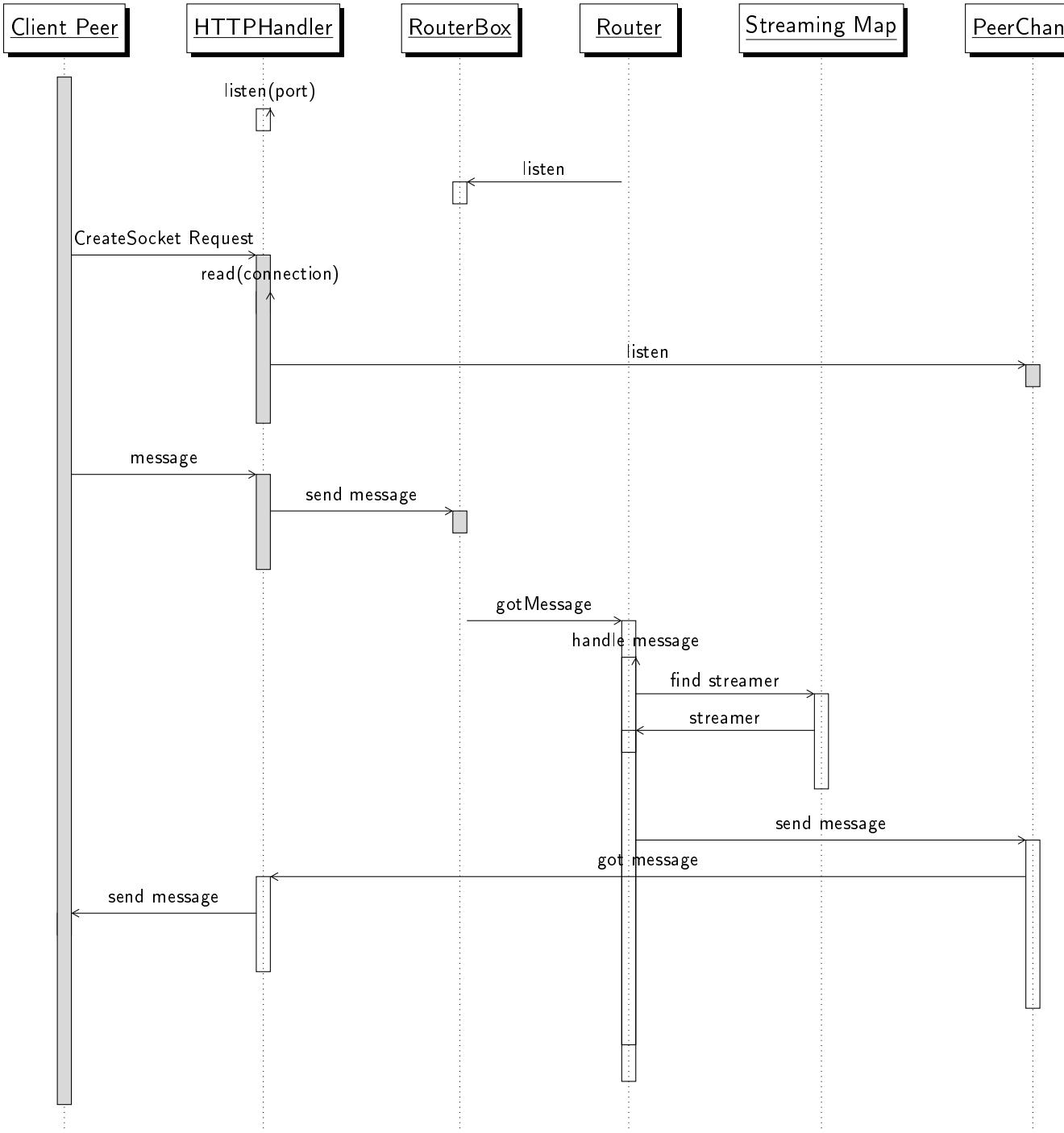

Figure 7: Sequence diagram showing the interaction that happens on the server side

6 EVALUATION

For evaluating the system, two different machines have been used. The server was running on a dedicated Amazon machine with 512 MB RAM, 1 vCPU and 20 GB SSD, under the Ubuntu linux distribution. A different machine has been used to run the client browsers, with 16GB RAM, intel I7-7700 HQ processor with 8 virtual cores and a 256 GB SSD. The web browser used during experiments as Google Chrome. Due to lack of resources, multiple clients have been opened from the same machine in different Google Chrome tabs. Also, for the same reason, measurements have been made with audio data only. Streaming video data for more then 4 peers is too expensive for the client machine.

Metrics that were considered for measuring were time taken for a peer to receive data from others, time taken from others to receive data from a new peer, time taken to re-balance the tree after a peer left the network, and bandwidth used under different scenarios. Time result can be seen in table 1 below, and bandwidth results in the following images.

Times have been measured under three setups: 5 existing peers, 7 existing peers and 10 existing peers. Measurements for times have been done from the client perspective. This is the effective time in JavaScript which is required to receive all input streams from the moment the peer joins the network.

The time to re-balance the tree is estimated to be around 40ms for a connection. Given the tree-like structure of the connections, the worst case is when a node that appears as a streamer in most trees leaves the network. Considering n peers in the network, and a branching factor of 2, then only n/2 nodes will be streaming in each tree. So the probability of a streaming node to leave the network is 1/2. Also, considering that only the children of the leaving node are affected by the rebalence, it means that in all trees at most n/2 children from

Table 1: Table illustrating times in ms required for peers to start receiving data in the network

Receiver	5 peers	7 peers	10 peers
Existing Peers	80ms	163ms	200ms
New Peer	68ms	100ms	191ms

each tree are affected. Based on this, one can argue that, although reablencing can damage the quality of the chat for a few moments, the cases in which it happens are rare.

Following are some graphs captured using Google Chrome's net internals API. These show the bandwidth usage in different scenarios. It is interesting to note the spikes that are created when a peer joins a network, and the noise that appears when a peer leaves. All the graphs bellow are generated on networks with 7-10 peers.

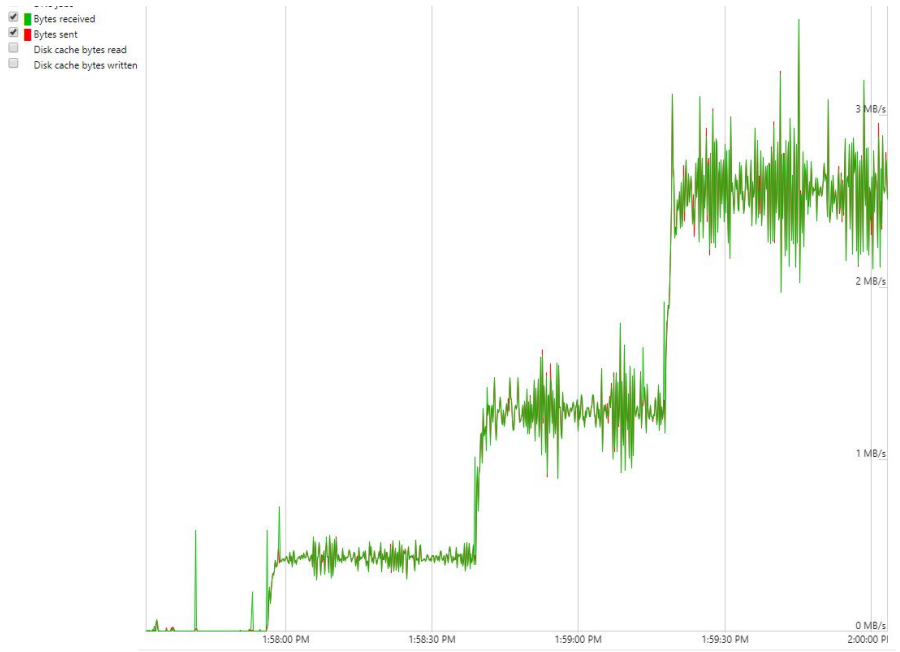

Figure 8: Illustration of network traffic during video communications

It is easy to see that when a peer joins the network, the trafic increases. It seems that in case of video streaming (8) the traffic increasses exponentially. This is expected, as a single machine handles the traffic for all clients in the network. This bandwidth would be linear in the case where each client joins with a different machine, as it would work in the real world.

The joining spikes are mostly visible in fig 9. This is expected, as there are a lot of messages being exchanged between the clients and the server. It is important to note that this does not last for too long, even with more peers in the network (the peak between 2:05:30 and 2:06:00). The graphs proove that the joining times are relatively small, and it does not affect the experience a client goes through when joining the network. This can also be seen when a peer leaves the network. However, in this case there is a bit more traffice, and more messages are sent (red spikes in fig 10).

Rejoining the network creates the most noise. Looking at fig 11, one can note that there are larger areas of red spikes (more data sent) when a peer rejoins the network (around time 2:19:00), and even more

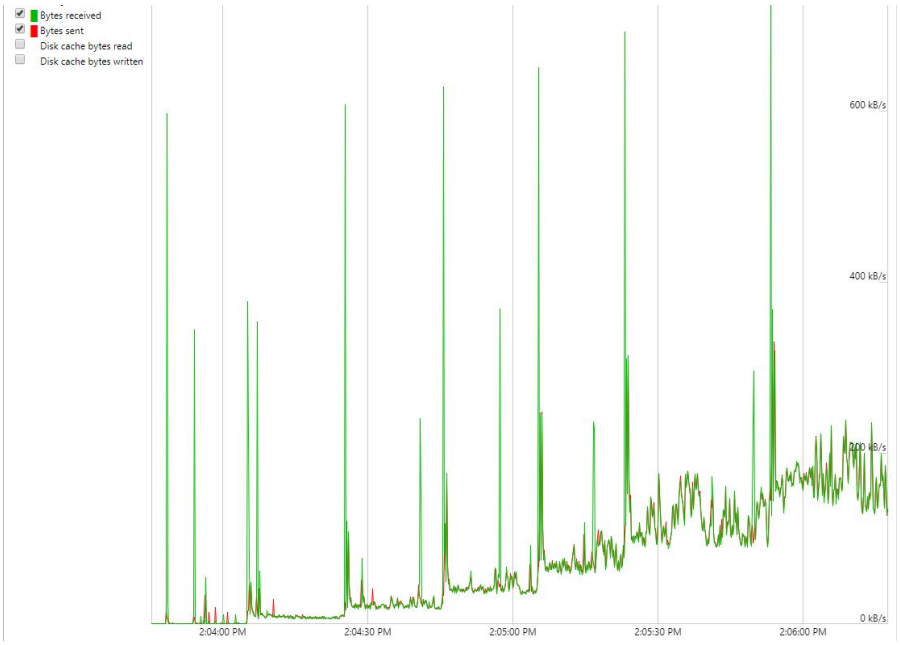

Figure 9: Illustration of network traffic during audio communications

if a peer that streams in most of the tree leaves (around time 2:19:30). However, this is barely noticeble in the chat as the dealy at a chat of this size is usually under 1 second.

Figure 10: Illustration of network traffic while peers leave an audio call

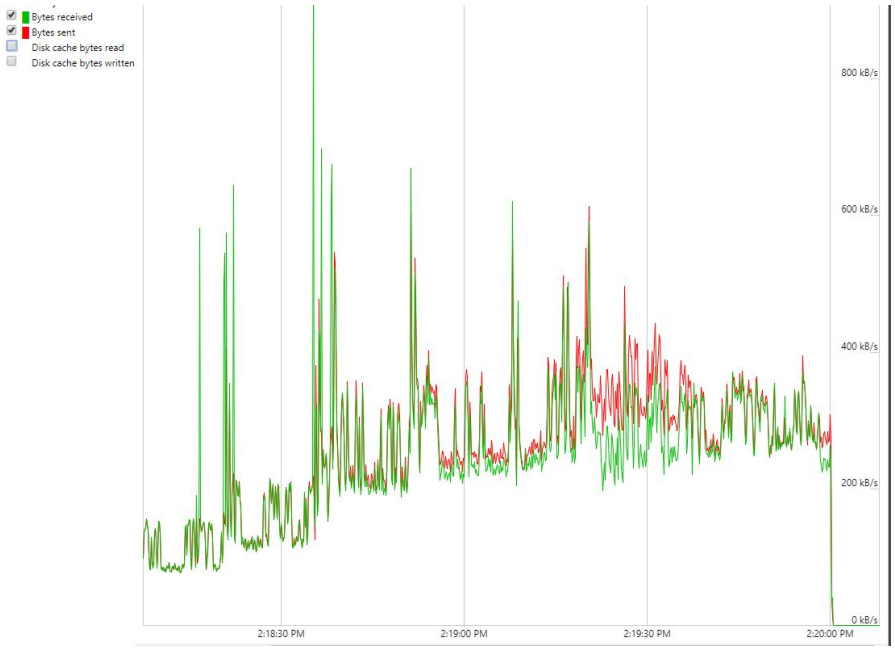

Figure 11: Illustration of network traffic while peersr rejoin an audio call

7 CONCLUSION

Throughout this report, a different approach to online video chats has been presented. Replacing the traditional mesh-based systems with multiple streaming trees can enable chatting systems to configure themselves based on peer bandwiths, thus allowing for more scalability. The experiments proove that this is fesible, as the bandwidth used grows lineary with the number of peers that join the network. However, there are still some challenges to overcome. Rebalancing the trees after a peer leaves the network can be very expensive. As seen, when this happens it creates a lot of noise in the system, and for a while peers will loose some of the connections.

In order to benefit from the full potential of streaming trees, some improvements can be made. The system should take into acount the available bandwidth of each peer, and form connections based on these. The resulting trees should distribute the data among all bandwidth available, and not allow weaker peers to become bottlenecks in the system. Moreover, to reduce rebalencing time, a better approach can be taken. Subtrees that are left orphaned can be reparented as a whole to a new parent instead of doing a complete rebalance. At first, this might create a longer branch inside the tree, but it reduces the number of connections that need to be created at once and orphaned peers get hold of the streams faster. Then, after the network is stable, a second phase can start slowly rebalancing the tree.

Given the streaming tree architecture defined throughout the report, and the possibility of future improvements, it can be concluded that this approach is a viable alternative to mesh systems, and could be investigated further to improve the quality of peer to peer communication systems.

BIBLIOGRAPHY

[1] Kasper Lodahl Flye Emil Stephansen. *Guardian tree: A single Tree-Based P2P live streaming system*. URL: http://www-cs-faculty.stanford.edu/\~{}uno/abcde.html.

[2] Mathias Glavind Schidt Jens Frederik Krogh Holdam. *Latency Aware multi-tree based P2P live video streaming*. URL: http://www-cs-faculty.stanford.edu/\~{}uno/abcde.html.

[3] Presenter-Bob Kinicki. "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol." In: ().

[4] Salvatore Loreto and Simon Pietro Romano. "Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts." In: 16 (Sept. 2012), pp. 68–73.

[5] Nazanin Magharei, Reza Rejaie, and Yang Guo. "Mesh or multiple-tree: A comparative study of live p2p streaming approaches." In: *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE. 2007, pp. 1424–1432.

[6] Jukka K Nurminen, Antony JR Meyn, Eetu Jalonen, Yrjo Raivio, and Raúl Garcıa Marrero. "P2P media streaming with HTML5 and WebRTC." In: *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*. IEEE. 2013, pp. 63–64.