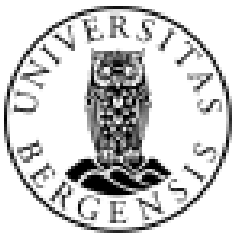
Interworking WebRTC with Enterprise Multimedia Communications

Bergen University College and University of Bergen

Joint Master's Programme in Software Engineering



Håvard A. Heggheim

Supervisors: Lars Petter Helland &
Anders Øvreseth

March 20, 2014

Abstract

The use of communication technologies today consume a large time of our daily lives. A lot of these technologies are now moving to Internet based solutions. Some of these services are collaboration tools that do video/audio and application sharing. This thesis will look at streaming media on the web in a collaborative enterprise environment. There are currently many Internet based communication services existing today that does this, but most of these are not open, cost money and require the installation of a plugin. This thesis will look at how Visual Solutions can integrate the new WebRTC APIs for doing real-time streaming of media in the browser with their current visual collaboration solution Virtual Arena.

The aim of this thesis is to find the most suitable way for Visual Solutions to implement live media sharing in the web browser with their current application Virtual Arena. The goal is to detect challenges and problems that might occur with the use of the new Web APIs.

The HTML5 standard introduces many new APIs that give web browsers the ability to communicate directly with each other in real-time. But there are many solutions and the different providers doesn't seem to agree on a single solution. This thesis will examine using these new APIs to see how we can integrate the browser in current media collaboration systems. The goal being to determine the feasability of using these new APIs and evaluate their usage. This thesis hopes to aid in developing solutions for media sharing in the browser that can be used in an enterprise setting.

Chapter 1

Acknowledgments

Contents

List of Figures

Chapter 2

List of Tables

Acronyms

Chapter 3

Introduction

Chapter 4

Background

This chapter will explain the background for the thesis, starting with the introduction of WebRTC[1]. Then we will look at the use of WebRTC on mobile devices and also take a look at similar technologies. We will also take at current use cases and implementations of WebRTC. We will last look at Visual Solution's application Virtual Arena for doing collaboration such as audio, video, and application sharing.

4.1 Introduction to WebRTC

This section gives an introduction to the new WebRTC APIs.

WebRTC is a collection of standards, protocols, and Javascript APIs. The combination of these enables web browsers to do peer-to-peer audio, video and data sharing between browsers. There is no plugin or third-party software required. Real-time communication is now becoming a standard feature in browsers that any web site can use via simple APIs.

Delivering such functionality such as live audio and video sharing and data exchange requires a lot of new processing capabilities in the browser. This is abstracted behind three primary APIs:

- MediaStream: capturing audio and video streams
- RTCPeerConnection: communication of audio and video data
- RTCDataChannel: communication of arbitrary application data

With the above APIs you can: capture media from a camera on your device, do signaling, peer discovery, connection negotiations and security to name

[1]Web Real-Time Communication: http://www.w3.org/TR/webrtc/

a few.

NAT traversal, media encryption, port multiplexing

4.1.1 Standards and Development

The WebRTC architecture consists of different standards, covering both native and browser APIs:

- All the different protocols and data formats required to make WebRTC work is defined by the IETF Working Group RTCWEB[2]. They are responsible for defining protocols, data formats, security, and all other aspects to enable peer-to-peer communication in the browser.
- The WebRTC W3C[3] Working Group is responsible for defining the browser APIs.

WebRTC is the first open standard to tranport data over UDP[4]. However doing this in the browser requires a lot more than raw UDP to do real-time communication:

4.1.2 Audio and Video

Doing live audio and video sharing requires processing to enhance image quality, doing synchronization, echo cancellation, noise reduction and packet loss concealment**?**. On the transmitting end the bitrate must be adjusted to fluctuating bandwidth and latency between clients. On the receiving end the client must decode the streams in real-time and be able to adjust network jitter and latency delays. These are complex problems, but WebRTC includes fully featured engines in the browser, which takes care of all the signal processing for us. All of the processing is done directly by the browser.

4.1.3 getUserMedia()

Acquiring audio and video is done by usig JavaScript APIs that enables the browser to acquire audio and video from a physical device such as a webcam or microphone. Incoming streams from remote network peers are also captured and everything is packaged in a MediaStream object. Inside the MediaStream object we have one or more individual tracks that are

[2]Web Real-Time Communications: https://tools.ietf.org/wg/rtcweb/
[3]World Wide Web Consortium: http://www.w3.org/http rfc
[4]User Datagram Protocol: https://www.ietf.org/rfc/rfc768.txt

synchronized with one another. The output can be sent to a local audio or video element, post-processing scripts or remote peers.

The MediaStream object represents a real-time media stream and allows the application to manipulate individual tracks and specify outputs.

The getUserMedia() API allows us to specify a list of mandatory constraints to match the needs of the applicaton:

```
var constraints = {
  audio: true,
  video: {
    mandatory: {
      width: { min: 1280 },
      height: { min: 720 },
      frameRate: 30
    },
    optional: []
  }
}

navigator.getUserMedia(constraints, stream, error);
```

Once a stream is acquired we can feed them into other APIs such as Web Audio for enabling advanced audio processing. Canvas API for post-processing video frames and WebGL can apply 3D effects on the output stream.

Simplified the getUserMedia() is an API to acquire audio and video streams. The media is automatically optimized, encoded and decoded by the audio and video engines. Then we can display the media locally in an audio or video element in the browser.

4.1.4 Real-Time Transports

When it comes to real-time communication, synchronization and low latency is more important than reliability. This is the reason why the UDP protocol is preferred for doing real-time communication. While TCP[5] delivers a reliable communication, there can be delays. If a packet is lost, it is re-requested. The human brain does't handle latency in communication very well, but we are good at filling in the gaps. Therefore we use UDP, which is a connectionless solution. It doesn't check the state of the message. So part of the message could be lost, and we wouldn't know, but the connection would run without delay.

UDP is the foundation for doing real-time communication, but to meet all the specified requirements of WebRTC, we need to support a lot of protocols

[5]Transmission Control Protocol: http://www.ietf.org/rfc/rfc793.txt

and services on top of that.

ICE[6], STUN[7], and TURN[8] are needed to establish a connection in over UDP in WebRTC. Encryption is mandatory in WebRTC, DTLS[9] is used to secure all transfers between peers. SRTP[10] and SCTP[11] are used to multiplex the different streams, provide congestion and flow control, and provide delivery on top of UDP.

4.1.5 RTCPeerConnection

The RTCPeerConnection is responsible for managing the peer-to-peer connection. It manages the ICE for NAT traversal, keeps track of streams, triggers renegotiation when required. It provides an API for generating offer and answer.

To be able to understand RTCPeerConnection we need to understand signaling and ICE.

Establishing a Connection

1. Input devices are opened for capture as the media source. This is done using the getUserMedia API.
2. Now we have to signal the other users that we want to connect to them. using RTCPeerConnection we send an SDP[12] offer to the other clients, which generates an SDP Answer.The SDP here includes ICE candidates. Which allows for firewall traversal. There is a fallback if both clients are on symmetric NATs and a connection isn't possible to use a TURN server that acts like a packet mirror, channeling all the packets through the TURN server.
3. Once connection is successful, a DTLS connection is opened and all the media from input devices are encoded into packets and transmitted using SRTP-DTLS streams.
4. At the destination, the packets are decoded and formatted into a MediaStream.
5. The MediaStream is sent to output devices

[6]Interactive Connectivity Establishment: RFC 5245
[7]Session Traversal Utilities for NAT: RFC 5389
[8]Traversal Using Relays around NAT: RFC 5766
[9]Datagram Transport Layer Security: RFC 6347
[10]Secure Real-time Protocol: RFC 3711
[11]Stream Control Transport Protocol: RFC 4960
[12]Session Description Protocol: RFC 4566

Signaling

While WebRTC does all the routing and connectivity check for us with the ICE protocol. We have to do session negotiation ourselves. To do this we must extend an offer to the receiving peer and we need an answer in return. How can we do this if the other peer is not listening for incoming packets? Choice of signaling application is up to us. The WebRTC standard does not define a signaling protocol, but the key information that needs to be exchanged is the SDP, which specifies the necessary transport and media configuration information necessary to establish a connection. This approach is outlined by JSEP[13]. Assuming we have a shared signaling channel, we can initiate a WebRTC connection.

```
1  var signalingChannel = new SignalingChannel();
2  var pc = new RTCPeerConnection({});
3
4  navigator.getUserMedia(constraints, onStream, error);
5
6  function onStream(stream) {
7    pc.addstream(stream);
8
9    pc.createOffer(function(offer) {
10     pc.setLocalDescription(offer);
11     signalingChannel.send(offer.sdp);
12   });
13 }
```

SDP

WebRTC uses a SDP to describe the parameters of a connection. It represents a list of properties describing the connection, types of media, codecs, bandwidth, and other metadata information.

Here is some of the information that is generated after a call createOffer() has generated the SDP description:

```
1  ...snip...
2  m=audio 1 RTP/SAVPF 111 103 104 0 8 106 105 13 126
3  c=IN IP4 0.0.0.0
4  a=rtcp:1 IN IP4 0.0.0.0
5  a=ice-ufrag:fAYfQM/iWMQPqiHs
6  a=ice-pwd:pgbuPPRdpKq+obC0lyRxVDe/
7  a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-
       level
```

[13] JavaScript Session Establishment Protocol: http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03

```
a=rtpmap:111 opus/48000/2
a=maxptime:60
a=ssrc:2209464108 cname:7oIEPieg3XZzHJdN
a=ssrc:2209464108 mslabel:
    uWu6kVvHhYbbkOtNalf5E2LFgjx4cpGMhnfo
a=ssrc:2209464108 label:2b626a18-c54c-4c1b-9f42-03519
    a9b63f2
m=video 1 RTP/SAVPF 100 116 117
...snip...
```

ICE

In order to establish a peer-to-peer connection, the peers must be able to send packets to each other. This is easy when you know which ip and port to listen to for incoming messages, but hard when you don't know. Normally there would be firewalls and NAT devices between most peers. In a local environment where there is no firewall, we could establish a connection between two peers by appending the IP and port number to the SDP, and forward it to the other peer. What ICE does is getting around these restrictions by doing connectivity checks and route planning between peers. ICE gathers all possible addresses it can in address:port and transport triplets**?**. ICE calls these 'candidates', and once candidates have been gathered, they are ordered in a list based on priority. Highest priorities are assigned to candidates with the least overhead: those that you get from the device itself, the IP 'host' candidates. Next in line are STUN candidates, which are f.ex obtained via UPnP[14]. Finally the 'relayed' candidates that is obtained from TURN servers come, this route is only available when no other route is possible.

4.1.6 Bringing it all together

To summarize the process of creating a peer-to-peer connection:

1. Initialize a shared signaling channel
2. Initialize a PeerConnection Object
3. Acquire local media streams
4. Register local media streams with PeerConnection
5. Gather ICE candidates and generate SDP offer describing connection and send to peer via the signaling channel

[14]Universal Plug and Play: https://tools.ietf.org/html/rfc6970

6. Register remote ICE candidates and initiate connection
7. Receive remote media stream

4.1.7 Secure Communication

Once we have completed the SDP anwers and offers, and traversed NATs, we have come a long way. But WebRTC require that we encrypt all communication. On top of UDP, we have SRTP used for transporting media securely, and DTLS which is used to negotiate secret keys for encrypting media data. This all taken care of by WebRTC, and once we have everyting else in place, we are ready to establish peer-to-peer connections.

4.2 Mobile devices

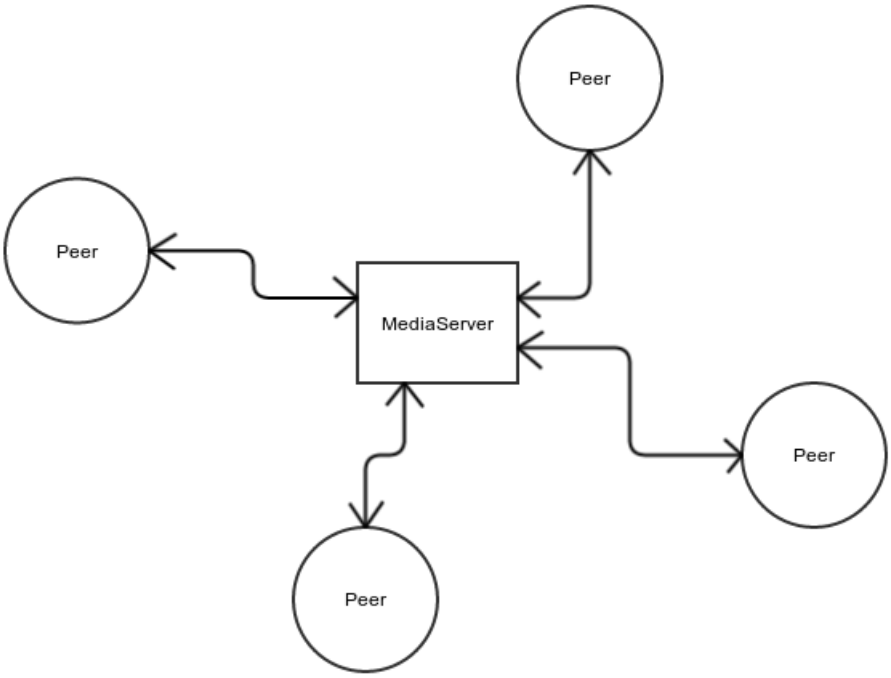
4.3 Related technologies

4.4 Current implementations

4.5 Virtual Arena

Virtual Arena is the application that Visual Solutions has created to do visual collaboration**?**. It supports one-to-one, one-to-many and many-to-many collaborative scenarios. The application uses a MCU[15] that acts as a media server. With this server Virtual Arena can support a lot more incoming and outgoing streams than in a simple peer-to-peer scenario. It also applies mitigation strategies for scenarios with limited bandwidth. Without going into too much detail of how the application is actually put together the main setup looks something like this:

[15]Media Control Unit: An endpoint used to bridge videoconferencing connections



Communication between peers and the media server is done by opening up ports in the firewall to listen for incoming tcp and udp connections. The media server can receives incoming streams and mix it in with the other streams and forward it to all the other peers. All the streams are identified using an SSRC.

4.5.1 Signaling

Virtual Arena uses a proprietary way of doing signaling over RTCP.

4.5.2 Transport

Raw RTP stream over UDP.

4.5.3 Media

Speex for audio and theora for video.

4.5.4 Application sharing

For doing application sharing the application window itself is divided into smaller subsets of rectangles, which are then captured as images. Depending on the bandwidth available, these are sent to the MCU. Then other clients can subscribe to the stream, and decode the images on the client side.

4.5.5 Security

While WebRTC provides security such as DTLS on top of SRTP, Virtual Arena only uses raw UDP streams because nothing more is needed. It operates in a closed business environement, so transport level encryption is not needed, because unidentified peers are not allowed inside the network anyways. The closed environment makes technologies like ICE, DTLS, and srtp unnecessary.

4.6 Summary

Chapter 5

Industrial Case

This thesis is written in cooperation with Visual Solutions.

Visual Solutions is a Norwegian company in the BB Visual Group. Their primary business is within the integration of operations for the oil and gas industry. They have offices in Houston, US, Brighton, UK and a headquarter in Bergen, Norway.

Visual Solutions contribute tailor made solutions enabling and improving collaboration and cross-disciplinary interaction across organisation units and geographic locations.

One of their applications Virtual Arena is a powerful and interactive tool that allows IP-based application sharing within collaborative sessions. Virtual Arena supports one-to-one, one-to-many and many-to-many collaborative scenarios. With support for high-performance application sharing, audio and video communication from an advanced 3D shared scene.

5.1 Visual Solution's Vison

In todays world everybody owns some kind of mobile device. If a user could enter a collaborative session from the web browser of their own simple mobile device, this would be of great benefit to the customer. The user would not have to install any additonal plugins or software to their device. Many new API's are currently being developed to improve communication and hardware access in the web browsers. The W3C has drafted a document that defines a set of API's to allow media to be sent to and from another browser or device implementing the appropriate set of real-time protocols.

Visual solutions wants to investigate how far development has come with these new API's and if their vision can be solved by using them in their

application.

The problem can be defined in two sub-problems:

Problem 1
Is it possible to use these new API's together with Visul Solutions current application? How can these new API's be implemented with their current software solution Virtual Arena?

Problem 2
It is in Visual Solution's interest to gain knowledge of how far the support for these new API's have come today in desktop browsers and on mobile devices. This would give an indication to whether or not there is anything to gain from investing in further development.

Chapter 6

Problem Description

As the use of mobile devices increase in business use, Visual Solutions wants to explore the possibility of developing collaboration tools that run directly in the browser using the new API's drafted by the W3C. This would allow their solutions to extend to more platforms. This chapter aims to explain the problems and what can be done to overcome them.

As described in the previous chapter 3. Visual Solutions has chosen to look at the new Web APIs that are under development for doing collaboration services. While there long have been Flash, and other services like Microsoft Silverlight's Smooth Streaming and Apple's HTTP[1] Live Streaming. WebRTC seems to have some important advantages over the other technologies.

- Universal mobile support
- Easy integration, with a high focus on security.
- No need to download clients or plugins.

By looking over the drafts done by W3C, WebRTC seems to be well suited for doing collaboration directly in the web browser.

6.1 Problem statement

WebRTC seems to be the best solution for doing video conferencing on the web, and both desktop and mobile web browsers have some degree of support which improves on a nightly basis.

[1]Hypertext Transfer Protocol: https://tools.ietf.org/html/rfc2616

While other solutions like Flash may have better support at the moment, there is a clear indication that the web is moving towards open standards. And a lot of effort is being put into WebRTC at the moment by the big players like Google and Firefox to make this the new standard, however there are still disagreements about which audio/video codecs is going to be used as a standard.

There has been done a lot of functional testing to see if the experiments work. What kind of features and codecs work and what does not. This is done beacuse all the browsers that support WebRTC only support different subsets of the features specified by the W3C.

Find out if WebRTC is a mature and usable technology to be used in a business environment.

Figure out what will happen in the near future with the support for WebRTC on different platforms?

6.2 Research question

This thesis will try to answer the following question:

How can Visual Solutions best integrate WebRTC with their current application Virtual Arena for doing collaboration?

Supporting questions:

- What is the current support for WebRTC in the different browsers?
- How well does WebRTC connections scale with more MediaStreams and peers, and if this can be improved using other models than the standard P2P model
- How will the support for this technology improve in the near future?

6.3 Research method

The research method used in this thesis, is a quantitative experimental method, using systematic investigation and analysis of the different implementations in browsers of the drafted APIs by the W3C. Doing experiments to examine the actual workings of the implementations to uncover support for the features.

Practical experiments consists of test cases that was run in Chrome, Firefox, and Chrome for Android. These experiments were performed because there is a lot of incomplete information on the current support.

Many smaller experiments were conducted in the begginning to gain knowledge of the workings of WebRTC. Lots of articles, blogs, books and papers were read.

Looking at trends in the development, this thesis will also be predicting support for the APIs in the near future.

6.4 Existing work

There exists a lot of tutorials on creating very simple implementations of WebRTC on the client side, usually setting up one-to-one connections using a third-party service such as Pusher or OpenTOK for doing signaling. More advanced guides shows how you can setup a Node.js server to do signaling using WebSockets.

This is great, but to be able to do what we need, we need to create an advanced server that acts like a client. While most commercial solutions that could potentially be of some help most are closed. But there is one open source project called Licode which utilizes the WebRTCs native libraries and is compatible with the standard and protocols. It's written in C++ and it acts like a MCU. It's open source, but not very well documented. It also doesn't offer what we need in terms of including external RTP[2] streams, but it says in their roadmap that this is planned for the future.

There is no good service on the web that provides live data on how far the support has come on different browsers. There is one service that will check for WebRTC and WebSockets connectivity on http://www.checkconnectivity.com/ You will find blogs and articles that provide tables showing who has implemented which APIs, but most will be outdated. The best solution is to test yourself using your own code or read the different browsers roadmaps.

If WebRTC is not supported in a users browser, there may be possible for the user to either update their browser or adjust a flag in the browsers experimental settings.

[2]Real Time Protocol: defines a standard packet format for delivering audio and video over IP networks.

6.5 Expected results

- From this thesis we want to have provided a solution to how Visual Solutions can integrate WebRTC with Virtual Arena. We want to provide information about all the necessary technologies that need to be implemented.
- We would also like to map the current support for WebRTC on different browsers.
- Also we would like to look at how well WebRTC connections scales with more MediaStreams and peers, and if this can be improved using other models than the standard P2P model.

Chapter 7

What is the current state of support of Real-time Communication Between Browsers

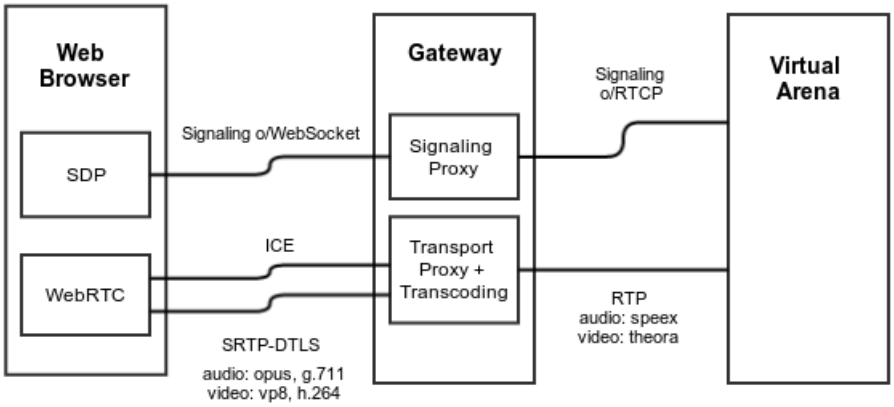
Chapter 8

How can we integrate WebRTC with Virtual Arena?

How can we integrate RTC in the Web with Virtual Arena? There are two solutions to this problem the way I see it. The first is very obvious and complicated, but probably the best solution. The second is more of an experimental hack rather than a viable solution.

8.1 The Gateway Solution

The obvious solution is to create a gateway using WebRTC and Visual Arena to turn any WebRTC enabled browser into a client. The gateway will allow the web browser on your preferred device to make and receive connections from Virtual Arena. The gateway would have to contain three modules: a Signaling Proxy, a Transport Proxy, and a Media Transcoder. The global architecture would look something like this:



8.1.1 The Signaling Proxy

Since WebRTC does not define any signaling protocol, one is free to use something custom made. But in this approach, the key information that needs to be exchanged is the SDP, which specifies the necessary transport and media configuration information necessary to establish a connection. This approach is outlined by JSEP. So the role of the Signaling Proxy module would be to extend the custom signaling protocol already used in Virtual Arena to include the necessary metadata provided in the SDP. It would go something lke this:

The MCU sends an offer via the signaling method, then on the client the remote party would install it using the setRemoteDescription() API.

```
1  ...snip...
2  m=audio 1 RTP/SAVPF 111 103 104 0 8 106 105 13 126
3  c=IN IP4 0.0.0.0
4  a=rtcp:1 IN IP4 0.0.0.0
5  a=ice-ufrag:fAYfQM/iWMQPqiHs
6  a=ice-pwd:pgbuPPRdpKq+obC0lyRxVDe/
7  a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-
       level
8  a=rtpmap:111 opus/48000/2
9  a=maxptime:60
10 a=ssrc:2209464108 cname:7oIEPieg3XZzHJdN
11 a=ssrc:2209464108 mslabel:
       uWu6kVvHhYbbkOtNalf5E2LFgjx4cpGMhnfo
12 a=ssrc:2209464108 label:2b626a18-c54c-4c1b-9f42-03519
       a9b63f2
13 m=video 1 RTP/SAVPF 100 116 117
14 ...snip...
```

8.1.2 The Transport Proxy

The WebRTC specification make support for ICE and SRTP-dtls mandatory. The problem here is that Virtual Arena uses raw RTP streams, it does not need the added security layers that WebRTC defines. It is up to the Transport Proxy to convert the media streams to allow these two worlds to interoperate.

How can we take advantage of ICE and it's security? By modifying a constraint in the IceTransport object we can modify which candidates the ICE engine is allowed to use. We can indicate that the engine must use only relay candidates. This can be used to prevent leakage of IP addresses.

8.1.3 The Media Transcoder

The WebRTC specification defines these mandatory codecs:

- Audio: opus and g.711
- Video: ?

There are still discussions on the topic of which video codec should be standard. The choice is between VP8 and H.264. The H.264 codec was recently made free by Cisco, so now both choices are royalty free. H.264 is the most widely deployed and currently has the best hardware support, but both Google and Firefox has decided to use VP8 in their WebRTC implementations.

Virtual Arena uses Speex for audio and Theora for video. So these would have to be transcoded to the appropriate formats.

This is one of the advantages of utilizing a MCU because you can add support for both H.264 and VP8 and be able to create a session between both Chrome, Virtual Arena, and Bowser.

8.2 The Experimental Solution//TO DO

A server that acts like a client. This could be done using the native client library. One problem with the native library is that it as far behind the web APIs in terms of development. This is because all the focus is on browser-to-browser scenarios. There are a couple of services out there that uses the native libraries, but most are commerce. There is one open-source solution that has the ability to create a server that acts like client, but we need more than that. We need to be able to attach remote incoming RTP streams to a peer connection. Signaling is already done on the server side so that

should not be any different from the previous solution. What we need is one gateway server that can listen to UDP-packets from Virtual Arena and act as a peer in WebRTC. This is not an ideal approach, but pretty cool if it should work.

For a single person that acts as a PeerConnection in WebRTC to listen in on a conversation from Virtual Arena, one would need to initiate a fake RTC Connection using two peers, then create a MediaStream from the incoming UDP packets and inject that MediaStream into the RTC Connection. On the returning side one would have to take the MediaStreams and break them into pure RTC packets with an SSRC identifier in the header and send them in return to the MCU.

First problem is receiving the incoming conversation from Virtual Arena this can be done setting up a socket that listens for incoming UDP packets. This is not a problem and can even be done in a pure Javascript environment using node.js

Then one would have to create a MediaStream Object from these packets. This is not currently possibly using chrome API's or Firefox. In chrome it is possible to create a pure audio MediaStream using the new WebAudio API, and in Firefox it should be possible to create a MediaStream from video using getStreamUntilEnded(), but this is currently broken. In the future however this should be possible using the drafted captureMedia APIs.

It is however possible to inject external MediaStreams into an RTC Connection.

For returning data from a PeerConnection to the MCU on would have to record the stream and return the data over a an UDP connection. This should be done using the proposed MediaStream Recording APIs, but none of the browser have currently implemented these yet.

Chapter 9

What will happen with the support of Real-time Communications in the near future?

Chapter 10

Discussions and Conclusion

summarize your thesis again as in the introduction. Describe how your evaluation revealed that your system is successful. Describe future work in this area.

We have looked at some basic theory about streaming media in the web browser, and explained our experiments and their capabilities. In chapter x we looked at the testing data collected and did an analysis of the results. The results were discussed and conclusions were made. This chapter will summarize the key elements of the results, look at what contributions have been made, and suggest some future work.

To simulate a low latency environment all early experiments were done on a local network. After collecting data using different mechanisms of protocols and streaming media, they were compared with the results from the same experiments using a simulated external network. Data about traffic, latency, and cpu-load were gathered and analyzed. Then we discussed the results with regard to inconsistencies, trends, synergies between different factors, and potential conclusions that could be made.

Contributions to reseach

Future Work Furter testing when different vendors have agreed on a common platform

Appendix A

Appendix Title