

MASTER'S THESIS | LUND UNIVERSITY 2014

Real-time video streaming with HTML5

Marcus Lindfeldt, Simon Thörnqvist

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2014-27



Real-time video streaming with HTML5

Marcus Lindfeldt

`marcus.lindfeldt@gmail.com`

Simon Thörnqvist

`simon.thornqvist@gmail.com`

August 20, 2014

Master's thesis work carried out at Axis Communications AB.

Supervisors: Andreas Jonsson, `andreas.jonsson@axis.com` – Branko Subasic,
`branko.subasic@axis.com` – Klas Nilsson, `klas.nilsson@cs.lth.se`

Examiner: Roger Henriksson, `roger.henriksson@cs.lth.se`

Abstract

The internet is quickly evolving and our expectations of web applications and websites have increased tremendously in the last couple of years. Although modern web-browsers are more capable then ever they are still falling behind when it comes to real-time video streaming. Traditionally, presenting a real-time video stream on the web was synonymous with installing and using third-party software such as Adobe Flash or ActiveX.

The purpose of this thesis is to investigate the feasibility to provide a plugin-less real-time video experience with the usage of HTML5 and its new features with a main focus on the Media Source Extensions API and the H.264 video format.

Three different approaches are presented with evaluations of how they perform in terms of latency, bandwidth, compatibility, CPU and memory usage. Ultimately we will show that it is indeed possible for real-time streaming of H.264 using only native browser API's.

Keywords: Real-time streaming, HTML5, Media Source Extension API, Websockets, H.264, MJPEG

Acknowledgements

We would like to express our gratitude to our supervisors Branko Subasic and Andreas Johnsson at Axis Communications for their support and invaluable feedback. We would like to thank the team at Axis Communications who conducted the initial research in this area. In addition, we thank our Lund University supervisor Klas Nilsson for his input and support.

Terms & Abbreviations

Terms

H.264 Proprietary video compression format

WebRTC Browser to Browser Real-Time Communication technology

Abbreviations

API Application Programming Interface

DASH Dynamic Adaptive Streaming over HTTP

HTML HyperText Markup Language

MSE Media Source Extensions

Contents

1	Introduction	9
1.1	Problem definition	9
1.2	Problem statement	10
1.3	Obstacles	10
1.4	Related work	11
1.5	Contributions	11
2	Technologies	13
2.1	Protocols	13
2.1.1	Transmission Control Protocol (TCP)	13
2.1.2	User Datagram Protocol (UDP)	14
2.1.3	HyperText Transport Protocol (HTTP)	14
2.1.4	Real-time Transport Protocol (RTP)	14
2.1.5	Real-time Streaming Protocol (RTSP)	15
2.1.6	Dynamic Adaptive Streaming over HTTP (DASH)	15
2.1.7	HTTP Live Streaming (HLS)	15
2.2	Video formats	16
2.2.1	Compression	16
2.2.2	Motion JPEG	16
2.2.3	MPEG-4 PART 10 (H.264)	17
2.2.4	ISO Base Media File Format (MP4)	18
2.3	Web technologies	19
2.3.1	JavaScript	19
2.3.2	Media Source Extensions	20
2.4	Browser networking	21
2.4.1	Asynchronous JavaScript and XML (AJAX)	21
2.4.2	Websockets	22
2.5	Tools	24
2.5.1	Gstreamer	24
2.5.2	Chrome DevTools	24

2.5.3	Wireshark	26
2.5.4	MP4 Analyzers	27
3	Analysis	29
3.1	Requirements	29
3.2	Support analysis	30
3.3	Possible approaches	31
3.3.1	Transport via proxy	31
3.3.2	Stream transformation	31
4	Solutions	33
4.1	MJPEG	33
4.2	H.264	34
4.2.1	Broadway H.264 JavaScript and Websockets	35
4.2.2	MP4 fragmentation, MediaSource API and Websockets	36
4.3	Authenticating Websockets	41
5	Evaluation	43
5.1	Test setup	43
5.2	Latency	44
5.3	Bandwidth	45
5.4	CPU & Memory usage	45
6	Discussion & Conclusions	47
6.1	Conclusions	48
6.2	Reflection	48
6.3	Future work	49
	Bibliography	51
	Appendix A Compatibility tables	59
	Appendix B Code Snippets	61
B.1	AJAX Request	61

Chapter 1

Introduction

HTML5 is the latest version of the markup language HTML which is used for structuring and presenting content on the web. HTML5 does not only refer to markup but defines a larger set of web technologies including new JavaScript APIs and CSS styling capabilities. HTML5 specifies a new subset of multimedia elements designed to make it easier to include video, audio and graphics in web applications without the need for third-party plugins.[33]

The video element was meant to standardize video on the web but has been struggling due to disagreements on what video formats and which codecs that should be supported. There has also been issues with providing APIs to support live streamed video which has made third-party plugin vendors hard to compete with, not only are their plugins widespread but they provide functionality to handle several scenarios, including live streamed video. The ability to incrementally stream video in an efficient way has not been possible in HTML5 until recently.

1.1 Problem definition

Traditionally presenting live-streamed video on the web meant installation and usage of third-party software such as Adobe Flash [3] or ActiveX [32]. This was due to no capability of native browser APIs to support a live stream. To be able to use the browser for video streaming without the need for third-party plugins could be considered to be a key usability feature, since it would “just work” out of the box without forcing the user to install or enable anything. At the same time even companies with strict software installation policies can use such a solution.

Real-time streaming imposes certain constraints on what is acceptable in terms of performance and compatibility. A plugin-free solution should have similar performance as a plugin-based one. At the same time it should work across platforms and in different browsers.

Based on these observations our main goals for such a solution have been summarized below. Please refer to Chapter 3 for a more in-depth discussion of the goals and requirements.

Browser only The end user should not have to install any additional software in order to view the real-time stream. A solution should work in all major browsers¹ regardless of the underlying operating system.

Performance The performance should be similar to a plugin-based solution and there should not be any unnecessary network traffic or notable delays.

Compatibility The solution should work in as many browsers as possible.

Quality There should be no additional loss of quality in video or audio compared to plugin-based solutions.



Figure 1.1: Context diagram.

1.2 Problem statement

The purpose of this thesis is to investigate the feasibility to present a real-time video stream from an IP network camera in a browser by utilizing native browser APIs.

This includes a technical study and evaluation of technologies in HTML5 together with browser capabilities for displaying real-time video, as well as to investigate and evaluate the most suitable way to transfer a video stream to the web browser.

1.3 Obstacles

There are several aspects to consider when investigating a possible real-time video streaming solution implementation. A browser based solution gives us additional limitations such as:

New technology Real-time video streaming support for HTML5 is still a very new technology and as such there is lack of proper documentation, there is stability issues and technologies and specifications can change rapidly.

Proprietary conflicts In the field of video formats there is a large gap between the video industry and the browser vendors as the industry generally uses proprietary formats such as H.264 which the browser vendors do not want to support, mainly due to licensing issues.

¹Google chrome, Mozilla Firefox, Microsoft Internet Explorer, Apple Safari

Browser limitations Normally you would be able to build your own client/video player that supports your functionality but since our goal is to stream video directly to the browser we are restricted to the browser's functionality and performance. Additionally browsers themselves have different functionality and performance depending on version and vendor.

Camera performance As with all technology cameras are also limited to its hardware and software. Some cameras may process recorded frames faster than others and we will never be able to achieve better latency and performance than that of the cameras internal speed.

Bandwidth At some point all video frames have to travel across the network. If the network is slow or congested it will take longer for the frames to arrive. Thus we will get a high latency and our stream would no longer be perceived as "real-time".

1.4 Related work

Although all this is fairly new some related work exists. Below is a list of some of the more notable works which has given us insight and inspiration.

dash.js DASH stands for Dynamic Adaptive Streaming over HTTP and dash.js is an MPEG-DASH compatible reference player created by DASH industry forum [13]. It aims to provide open source client-side JavaScript libraries which utilizes the Media Source Extensions API in order to enable streaming of ISO Base Media Files over HTTP and present the stream in a HTML5 video element [14].

Broadway Broadway is a H.264 decoder developed in JavaScript by Michael Bebenita. Broadway is based on an existing Android H.264 decoder which was simplified and then ported to JavaScript. [30, 31]

Dronestream Bernhard Weissshuhn developed a node.js solution which streams video from a remote controlled helicopter with a built-in camera to a proxy server which parses the payload and sends the raw H.264 frames to the browser. The frames are then rendered by Broadway onto the screen. [11]

MP4Box & DashCast MP4Box & DashCast are applications to create fragmented and DASH compliant MP4 files developed by Telecom ParisTech for the GPAC project. [22]

1.5 Contributions

The work of this thesis was divided evenly between the authors. To speed up development Simon took responsibility for the testing environment and implementation of core modules while Marcus was responsible for the overall structure and design.

By always sharing the responsibilities of implementation as well as writing this document the authors felt they could achieve a better result.

Chapter 2

Technologies

This chapter will cover key concepts and technologies related to real-time streaming and browser networking as an introduction to later chapters. It includes brief descriptions of different protocols, video formats, web technologies and browser networking as well as an overview of a set of tools we used for the solutions and performance tests.

2.1 Protocols

Protocols exist so that a computer can communicate with itself or other computers. A protocol is essentially a set of digital rules for data exchange within or between computers. We use the term network protocol when data is exchanged over a network. Additionally transport protocols provide end-to-end or host-to-host communication services for applications.

2.1.1 Transmission Control Protocol (TCP)

The transmission control protocol is a reliable connection-oriented transport protocol designed to provide end-to-end delivery between two hosts in a packet-switched network.

Connection-oriented refers to the fact that TCP needs to first establish a session between the hosts that wish to communicate. This is done with the so called “three-way handshake” which is the process of initiating the connection to enable further communication [39].

As TCP is reliable all bytes received are the same as the bytes sent and in the same order. To enable this TCP provides operations to detect and correct errors in the byte stream. The design is focused around accurate delivery and not timely delivery which make TCP less suitable for real-time applications since it can induce higher latencies if there is a lot of retransmissions. Nevertheless it is still perfectly acceptable to use TCP for real-time communication although it might not be the optimal solution.

2.1.2 User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) is a transport protocol designed to transport packets of data (datagrams) between interconnected computers. UDP enables application programs to communicate with a minimum protocol overhead. This makes UDP incredibly fast and suitable for transporting uncritical packets in real time such as a video or audio frame. At the same time it also makes the protocol unreliable in the sense that it cannot guarantee delivery or duplication protection.[38]

2.1.3 HyperText Transport Protocol (HTTP)

The HTTP protocol [18] is a request-response protocol which simply means that a web client (Internet Explorer, Chrome, Firefox, etc.) sends a request to a web server and the web server responds with the requested data. HTTP operates on top of TCP and thus shares the same advantages and disadvantages.

HTTP has different request methods to accommodate all necessary client/server transactions. The most common methods are GET & POST. A GET method simply requests a server response. A POST method is similar to GET with the exception that you can send a message body with data as well.

Additionally HTTP has a rich feature set of status codes that are sent along with the response to indicate the type of response e.g. 404 Not Found or 500 Internal Server Error.

HTTP is stateless and does not save state between requests. Nowadays many web applications are required to save state between requests. A common use-case is to save a user session after a successful login so that a user does not have to type in a username and password for each click of a link or page reload. To save state you can use a server-side scripting language alongside HTTP which gives you the ability to save state in a file-system or database and retrieve it for other requests when needed.

2.1.4 Real-time Transport Protocol (RTP)

The real-time transport protocol is a protocol suitable for transporting real-time data such as audio or video. It is merely a transport protocol and does not address resource reservation and does not guarantee quality-of-service (QoS) for real-time services but it is usually used in conjunction with the real-time control protocol (RTCP) to handle these issues [41]. The protocol mainly adds the payload type such as H.264 or MP4 [25] as well as sequence numbers to the payload to detect out of sequence arrival, which is a common incident during IP transmission due to packets taking different routes to their destination. RTP does not have a delivery mechanism to handle multiplexing or checksums [19]. It is usually encapsulated in the lower level UDP protocol but can be used with any suitable underlying transport protocol.

The Secure Real-time Transport Protocol (SRTP) is a profile of RTP which aims to ensure authenticity, confidentiality, message authentication and replay protection of RTP and RTCP traffic. SRTP works by intercepting the RTP packets and forwards identical SRTP packets to the receiver. Similarly at the receiver side RTSP packets are intercepted and RTP packets are sent up the stack. [9]

2.1.5 Real-time Streaming Protocol (RTSP)

RTSP is a protocol which acts as a network remote control for one or several time-synchronized streams of continuous media. Typically a server maintains a session with an identifier for each client, but is agnostic to the underlying transport level connection and the session is not in any way bound by it. RTSP streams can use RTP but does not rely upon the underlying transport.

RTSP was designed to have similar syntax and operations as HTTP so that extension mechanisms to HTTP can also be added to RTSP in most cases.

Although RTSP and HTTP are similar, HTTP does not have the concept of sessions. An RTSP session is stateless and both the client and the server can emit requests to each other. [42]

2.1.6 Dynamic Adaptive Streaming over HTTP (DASH)

Dynamic adaptive streaming over HTTP is a technology for delivering media content in different bit rates and qualities. DASH is an adaptive bit rate streaming technique in which the control lies with the client. Thus the client is responsible for requesting the data and changing quality using the HTTP protocol. [26]

DASH is divided into two main parts, the Media Presentation Description (MPD) and the data segments.

The MPD is an XML-file that describes the format and resource identifiers for media segments. A resource identifier is an ordinary URL with or without a byte range which the client can request with a HTTP GET request. The MPD must supply enough information to provide a streaming service in which the segments can accessed through the scheme of defined resources.

A DASH client can request different data segments based on the current network conditions. This makes it possible to display a smooth stream with the highest possible quality by always requesting segments with the most suitable bit rate. This means that you can automatically lower the quality of the stream if the network temporarily gets congested.

DASH enables the client to stream large files by requesting a small duration at a time, which can both be used as a means to stream a recorded video or a live event.

A web server is required to provide valid DASH compliant media and an MPD file. The client needs to be able to parse the MPD in order to request the correct segments.

2.1.7 HTTP Live Streaming (HLS)

HLS is a technology developed by Apple for streaming audio or video from a web server. It is similar to DASH in the sense that the client is given a manifest file containing URL's to video segments served as small files. The web-server can serve several qualities of the same video and let the client adjust to playing the best suited quality depending on the current network condition. Currently only Apple products and Android devices supports HLS. [5]

2.2 Video formats

The notion of video formats is often a source of confusion. A video format is a generic name for a media container with related codecs. It is often helpful to visualize a video format as a zip archive where the container is the file itself and the codec is a set of instructions describing how the content should be uncompressed and interpreted. Containers and codecs related to real-time streaming and browsers will be discussed in further detail below.

2.2.1 Compression

”Compression is the process of compacting data into a smaller number of bits”
(Iain E. G. Richardson)

In video compression we compress the video sequence into a smaller number of bits. Compression increases the transmission capacity (bandwidth) as less bits have to be transferred over the network. The technology to compress a resource is called an encoder and similarly a decoder decompresses a resource. An encoder with a matching decoder is usually described as a codec. Compression can be either lossy or lossless. There is a trade-off between using a lossless vs lossy codec. A lossless codec does not alter the quality of the resource but is usually not able to reduce the size of the resource as much. A lossy codec will reduce quality upon encoding but can significantly reduce the size of the resource so the choice really depends upon the application. Real-time video streaming requires low latency to be able to provide a continuous stream so lossy codecs are often the preferable option. [40]



Figure 2.1: The encoding/decoding process.

2.2.2 Motion JPEG

In its simplest form Motion JPEG (M-JPEG) is just a set of concatenated JPEG images delimited by a specified delimiter or boundary. Unlike most popular video formats there is no defined standard for M-JPEG so there may be differences in implementation between vendors. The native support from web browsers such as Mozilla Firefox and Google Chrome makes M-JPEG a popular format for video-capturing devices such as IP cameras and webcams.

M-JPEG can be transported by a variety of protocols but HTTP is by far the most common one. When sending Motion JPEG over HTTP the header of a HTTP response contains a multipart [12] content-type (Figure 2.2) that tells the client that the data is separated in multiple parts with a specified boundary. Unlike regular stateless HTTP communication

the TCP connection will not be closed until either party explicitly closes it. M-JPEG also lacks timing information which makes it impossible to sync video with audio.

```
1 HTTP/1.0 200 OK
2 Cache-Control: no-cache
3 Pragma: no-cache
4 Expires: Thu, 01 Dec 1994 16:00:00 GMT
5 Connection: close
6 Content-Type: multipart/x-mixed-replace; boundary=myboundary
7
8 --myboundary
9 Content-Type: image/jpeg
10 Content-Length: 7168
11
12 ...image data...
13
14 --myboundary
15 Content-Type: image/jpeg
16 Content-Length: 7168
17
18 ...image data...
```

Figure 2.2: Part of a M-JPEG stream, images are separated by the multipart boundary `--myboundary`.

2.2.3 MPEG-4 PART 10 (H.264)

H.264 is a standardized video compression format and was designed to provide high video quality with lower bit rates than previous standards. H.264 is highly adopted in the video industry and is used in range of different applications and contexts including Blu-ray video and video streaming on the internet.

H.264 consists of three different types of frames I-frames, P-frames and B-frames.

I-frames or intra frames are independent from all other frames and can be decoded without any reference to any other frame. I.e I-frames contain the entire picture, the first frame of any video is always an I-frame.

P-frames are predictive intra frames which means that they reference previous I-frames or P-frames and only contains the difference from the previous frame.

B-frames are bi-predictive inter frames and are similar to P-frames in the sense that they contain the difference from the previous frame but can also reference future frames. [8]

H.264 frames are organized in Group Of Pictures (GOP). A GOP starts with an I-frame and contains several P/B-frames. As P/B-frames contain the picture expressed as changes in the scene, they are considerably smaller than I-frames; this results in a lower bit rate compared to other compression formats such as MPEG-2.

2.2.4 ISO Base Media File Format (MP4)

ISO Base Media File Format (ISO BMFF) is a general description of a format which serves as the bases for several more specific file formats, among others the media container MP4. ISO BMFF was designed to be a flexible and extensible format in which the presentation of media can be local, as in the system serves the presentation, or the presentation can be received via a network. This section will cover general structure of ISO BMFF as it serves as the basis for the byte stream format covered in section 2.3.2.

File Structure

The ISO BMFF structure is built using boxes or objects and the file is formed as a series of these boxes. The boxes are object-oriented which enables the file to be decomposed into independent boxes in a simple manner. A box in its simplest form defines a type and a length attribute. The type is a unique name to identify the type of box, the length is the total length of the box in bytes including type and length attributes. There are in total 71 boxes defined in the specification, which all can be divided into three main groups of boxes:

Box Contains a type and a length attribute and can be said to be a superclass of all other boxes.

Container box An extension of box and contains a set of related boxes. Container boxes serve as the foundation for the file structure and can contain several other container boxes.

FullBox Extends box and contains information about the version and flags.

Box structure and order

The ISO BMFF specification defines a recommended box order for top level boxes. Some are required to be placed in a certain order while others can be more freely placed based on the nature of the file.

Ftyp Shall be placed before any other box. Defines the file type and brands, major, minor and compatibility brands which the decoder must support in order to decode the data.

Moov Movie Box, general container for all metadata. The moov can be considered to be the entire header of a ISO BMFF and includes information about each of the tracks, the duration and timescale as well as display characteristics. If the file is destined to be progressively downloaded or streamed the moov needs to be in the beginning of the file to ensure that the movie information is downloaded first. If the moov would be placed in end of the file, the entire file would need to be download before playback [29].

Moof Movie Fragment Box. Contains boxes which hold information about the media data such as data offset, where in the presentation a sample should be played. The moof does not contain any actual samples, since those are defined in the mdat. But have boxes which specify size, duration, presentation time. Simply the moof references the mdat. [3]

Mdat Media Data Box. Contains samples of a track; a mdat can contain one or many samples which are referenced in the trun box in the moof.

There are more top-level boxes included in the specification which are not covered here since they are not required for the desired byte stream format covered in Section 2.3.2.

2.3 Web technologies

The modern web consists of several different technologies and APIs. In this section we will introduce a handful of these technologies and especially those relevant to real-time video streaming.

2.3.1 JavaScript

JavaScript was originally developed by NetScape with the name LiveScript as a way to add programs to web pages in the NetScape Navigator web browser. In an attempt to ride the wave of the programming language Java's [37] popularity LiveScript was intentionally marketed as JavaScript which has been a source of confusion ever since. As JavaScript gained popularity it became adopted by other browser vendors and has now become the de facto standard web scripting language.

To formalize the language a standard was written by an organization called ECMA as an attempt to standardize how the language should function. This is why many people refer to the ECMAScript standard when they talk about JavaScript. In practice both names can be used interchangeably as they essentially refer to the same thing.

Although JavaScript is mainly used in web browsers it is not limited to the browser. Nowadays it can be used in many different environments e.g. node.js[27] which is a JavaScript based platform for networking applications such as web servers. [15, 35, 16, 28]

JavaScript is prototype-based which has proven to be a bit confusing for developers experienced in class-based languages. A prototype-based language does not have the concept of a class. It is a style of object-oriented programming in which inheritance is performed by a process where each object has an internal link to another object called its prototype. In turn that object has a link to a prototype of its own and so on. This chain of object links is called a prototype chain. This construction is not as popular as traditional class based languages but is actually quite powerful. It is relatively easy to introduce a classic model on top of a prototype model but it is significantly harder the other way around.

2.3.2 Media Source Extensions

The Media Source Extensions (MSE) is a W3C candidate recommendation standard submitted by representatives from Google, Microsoft, and Netflix. The Media Source Extensions API extends the `HTMLMediaElement` [34], which is an interface with special characteristics shared by all media-related objects such as the `HTMLVideoElement`.

The Media Source extension makes it possible to generate media streams with JavaScript which can be used to playback video or audio. The `MediaSource` object serves as a representation of the source of data to a `HTMLMediaElement`. To attach a `MediaSource` object to a `HTMLMediaElement` the `MediaSource` object needs to create a Blob URI [6] and attach the URI to the `src` attribute of the `HTMLMediaElement`. The `MediaSource` uses buffers called `SourceBuffers` to contain media data that the `HTMLMediaElement` should present in the browser.

The `MediaSource` object uses a `readyState` property to indicate what state the object is in [2]. The specification identifies three states:

Open A media element has opened the source and is ready to accept and append data to its `sourcebuffers`.

Closed The source is not attached to a media element.

Ended The source is attached to a media element, but stream has ended.

The state is set to open after being attached to a `HTMLMediaElement` and a `SourceBuffer` has been added. It will keep being in the state open as long as no errors occur or `endOfStream()` is called.

Attaching a `MediaSource` object to a `HTMLMediaElement` without adding a `SourceBuffer` will keep the `readyState` in closed. If `readyState` was open but an error occurred on any of the `SourceBuffers` the `SourceBuffer` will be removed from the source and its state set to closed [2].

Byte Stream Format

In section 2.2.4 a general description of ISO BMFF was covered; this section will cover more specifically what requirements Media Source Extensions has on a provided bytestream. Although MSE accepts other formats than ISO BMFF they are out of this scope and will not be covered.

Initialization segment The initialization segment is a sequence of bytes that contains the information required to decode media segments. In the byte stream specification an initialization segment consists of a single `ftyp` and a single `moov`. The `ftyp` box cannot contain a major brand or a compatible brand which is not supported by the user agent, if it does the `MediaSource` will change its state to ended and throw an “invalid access error”. The `ftyp` box need to come before the `moov`.

The `moov` is required to have a movie extend box (`mvex`) in its container; this is to signal that Movie fragments are to be expected. The boxes and fields in the `moov` can not violate the requirements specified by the `ftyp` major or compatible brands.

Media segment A media segment is required to have a moof followed by one or more mdats. The moof needs to contain a track run box (traf), which in turn needs to have a track fragment decode time box (tfdt) and a track run box (trun). The tfdt provides the absolute decode time i.e. when on a media timeline a sample should be decoded. The trun references the samples in the mdat and can not reference other samples which are not present in the mdat. The trun specifies a property `dataOffset` which is the offset in bytes where media data is located in the mdat.

The initialization segment needs to be appended before any media segments. Generally an initialization segment is only appended once and followed by one or more media segments. However it is possible to append another initialization segment mid-stream, if for instance one would like to display another stream or change the properties of the current stream [1].

2.4 Browser networking

The web today consists of much more than just HTTP. Browser APIs such as the WebSocket API and webRTC¹ fill their own niche and can accomplish tasks which HTTP can not. HTTP is however the cornerstone in browser networking and in many modern web applications several technologies are used to provide the end user with the best experience possible.

2.4.1 Asynchronous JavaScript and XML (AJAX)

Asynchronous JavaScript and XML or AJAX is a web technology that allows you to communicate with a server via the `XMLHttpRequest` [4] object. The `XMLHttpRequest` object can send and receive a variety of formats such as JSON, XML and HTML. The technology became popular largely because of its asynchronous nature which enables you to send and receive data without refreshing the page. This lets you update portions of a page when an event occurs, for example when a user clicks a button or link.

To communicate with the server you will need an `XMLHttpRequest` JavaScript object.² This object was originally developed by Microsoft and later adopted by all major browser vendors such as Google and Mozilla. It has been standardized by the W3C since 2004. AJAX is not a duplex technology and does not support two way communication that is required by real-time services such as chat applications or video streaming. Even so this can be accomplished with a less optimal solution called polling.

Polling

Polling is the process of periodically asking the server if it has any updates. The client simply creates AJAX requests at a periodic interval to check for updates. If the server has any new data it is sent to the client otherwise the server sends an empty response as illustrated in Figure 2.3. Polling is simple and well supported but also very inefficient as

¹For more information on webRTC: <https://developer.mozilla.org/en-US/docs/WebRTC>

²See appendix B.1 for a complete AJAX fetch example.

you frequently send unnecessary requests to see if there is any new data. This could be remedied by increasing the duration between each request (long polling) but that will cause delayed delivery of updates and would not be very useful for real-time communication. An example of polling with the jQuery JavaScript library is shown in Listing 2.1.

```
1 function poll() {  
2   // create an ajax request  
3   $.post('poll/example.html', function(data) {  
4     // process results here ...  
5   }).always(function () {  
6     // wait until we get a response  
7     // before issuing another request  
8     setTimeout(poll, 5000);  
9   });  
10 }
```

Listing 2.1: Polling with jQuery

2.4.2 Websockets

Websockets is a technology which enables full-duplex communication over TCP. Websockets are commonly used in web applications via JavaScript. The Websockets technology consists of two parts, the protocol and its API. The protocol was standardized by the IETF as RFC 6455 [17] in 2011 and the API specification [24] is currently being finalized by the W3C. [23]

Websockets was primarily designed to be used with web browsers and web servers but can be used by any client or server application. Even though it has not been around that long it has gained a lot of popularity due to the fact that it enables you to push arbitrary data to the client, see Figure 2.3. Historically this has been achieved by an abuse of the HTTP protocol and continuously polling the server for updates.

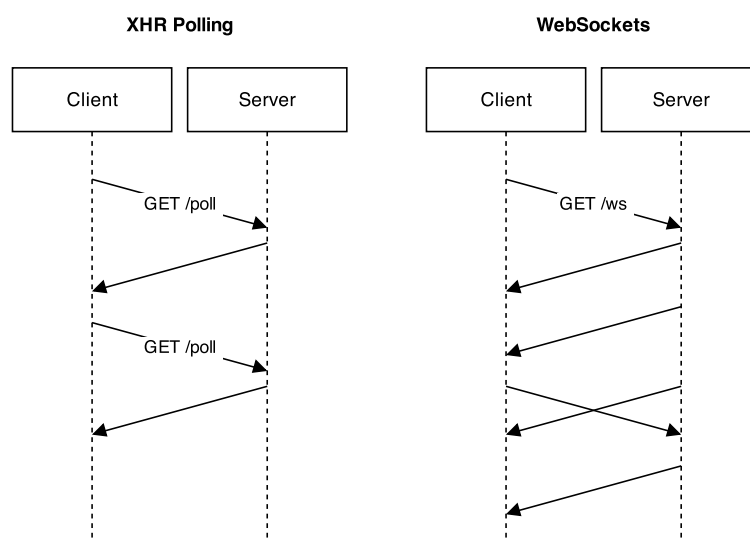


Figure 2.3: XHR Polling vs Websockets client-server communication

Two-way communication is essential for real-time applications such as chat applications or games and the ability to push data to all clients also makes it possible to stream media.

WebSocket communication between a client and a server is performed via a message-oriented API. Messages can be of either UTF-8 strings or binary data. The WebSocket protocol does not pose any constraints on the payload, since internally only the length and the payload are tracked.

When a message is sent an `onmessage` event will only trigger once all data has been received. The received message is then automatically converted to a string, or a binary object.

Websockets is implemented in most modern browsers and mobile devices however older browser versions may lack support for sending and receiving binary objects and are thus limited to strings only.

2.5 Tools

Our solution was built with a set of open-source tools and libraries. A brief introduction to some of the more important tools related to the solution is given below.

2.5.1 Gstreamer

GStreamer is an open-source multimedia framework. It allows for creation of a variety of media components such as video and audio streaming. The framework is modular with an extensive set of plugins to handle most media related tasks such as encoding, decoding, transcoding, muxing and otherwise transform video and audio to other formats and streams. At its core GStreamer use the concept of elements. An element is an object that provides some sort of functionality and several elements can be linked together to create a so called pipeline (Listing 2.2).

```
$ gst-launch-1.0 -q rtspsrc latency=200 location="rtsp://<servername>/axis-media/media.amp" ! rtpH264depay ! H264parse ! decodebin !
vp8enc ! webmmux ! filesink location=cam.webm
```

Listing 2.2: A typical gstreamer pipeline, transcoding H264 to vp8.

Typically a pipeline always starts with a source and ends in a sink. A source and sink can be anything from a file to a network or webcam stream. Additionally the pipeline can branch out to several sinks (Figure 2.4) e.g. if you would like to separate the video from the audio and process them differently. Naturally the reverse is also possible i.e. combining several sources into one sink. The process of branching and combining streams is commonly called multiplexing and demultiplexing or short muxing and demuxing.

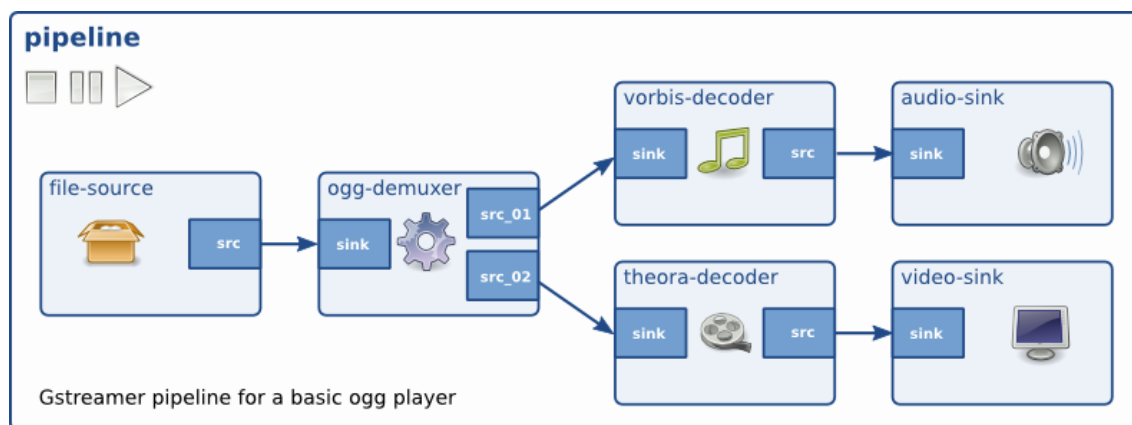
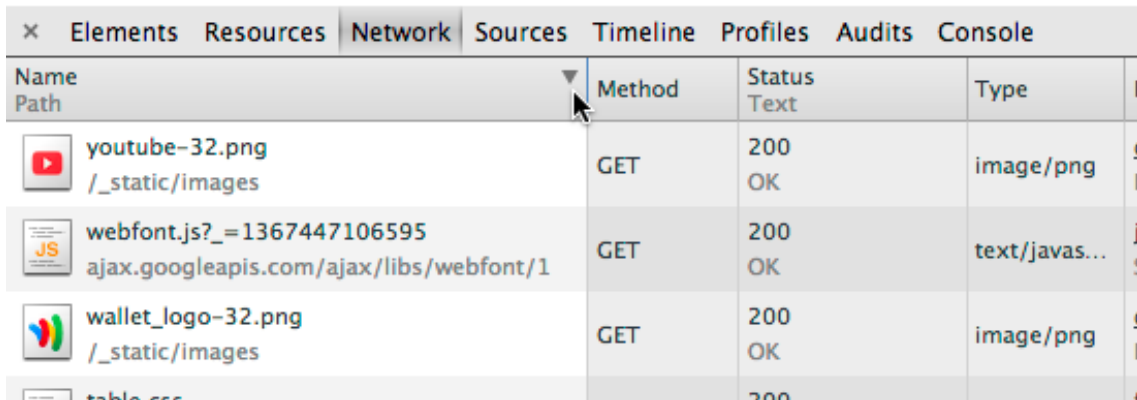


Figure 2.4: Gstreamer demultiplexing pipeline.

2.5.2 Chrome DevTools

The Google Chrome browser has, like many other browsers, built in development and debugging tools [21] that allows access to the internals of the browser and your application.

Among other things we can audit CPU and memory usage, monitor network traffic and track errors. Another useful feature is the ability to access a JavaScript console to debug and edit JavaScript.



Name	Path	Method	Status	Text	Type
youtube-32.png	/_static/images	GET	200	OK	image/png
webfont.js?_=1367447106595	ajax.googleapis.com/ajax/libs/webfont/1	GET	200	OK	text/javas...
wallet_logo-32.png	/_static/images	GET	200	OK	image/png
table.css		GET	200		

Figure 2.5: Chrome DevTools. Network monitoring.

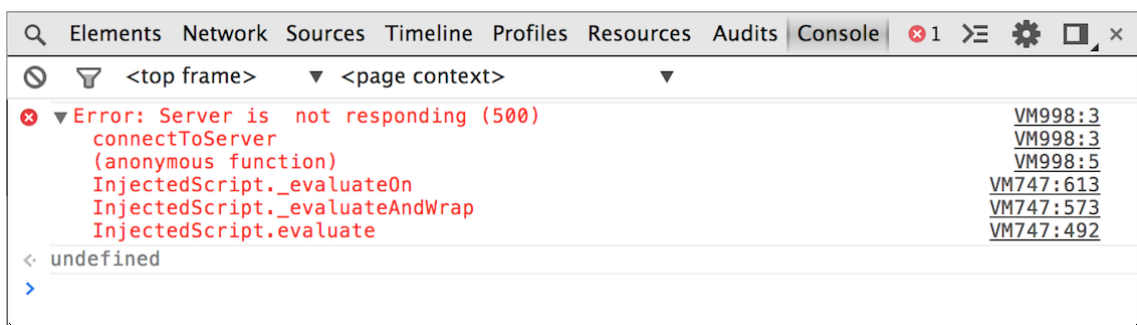


Figure 2.6: Chrome DevTools. JavaScript console.

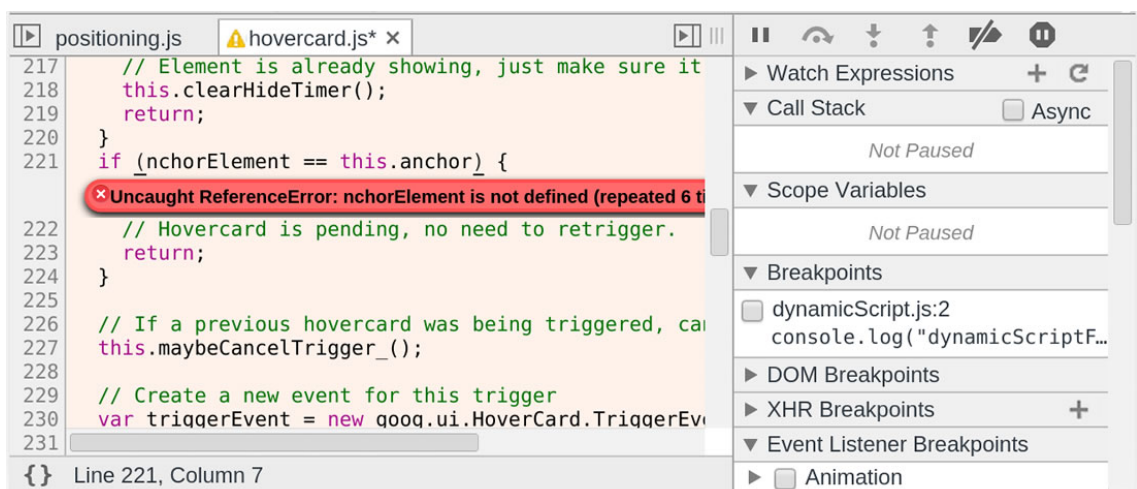


Figure 2.7: Chrome DevTools. Exception tracking.

2.5.3 Wireshark

Wireshark [20] is one of the most popular network protocol analyzers. It is considered the de facto standard across many industries and educational institutions. Wireshark is very useful as a debugging and error tracking tool for inspection of protocols and network packets at a microscopic level.

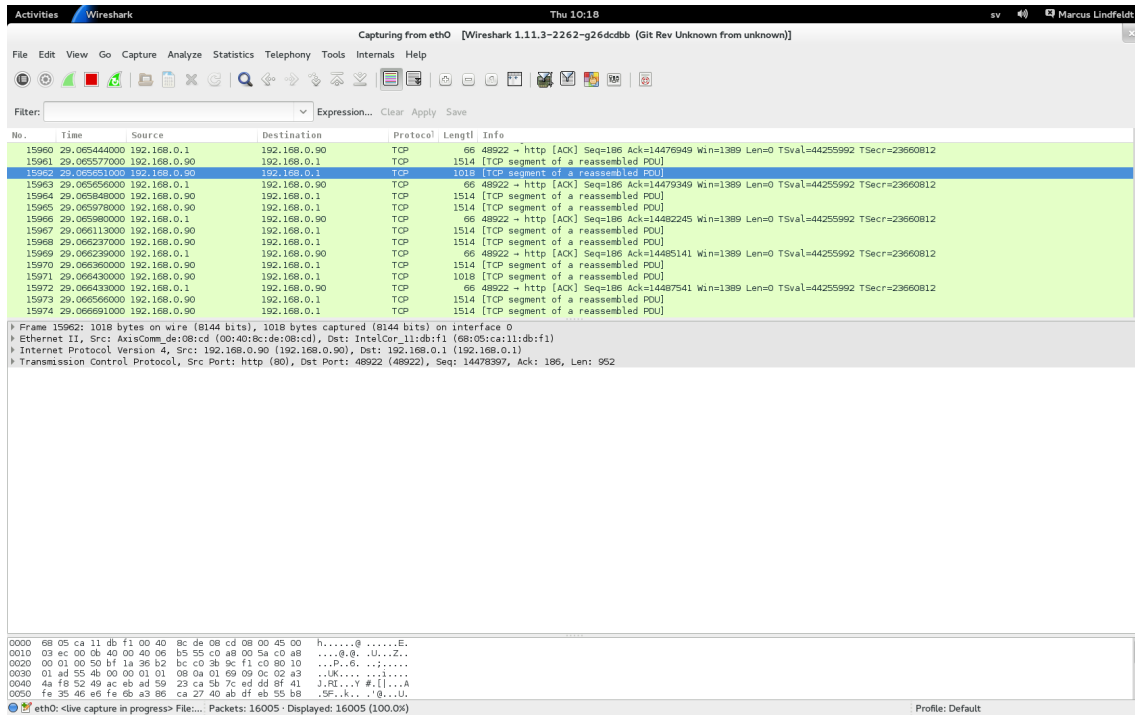


Figure 2.8: Wireshark capture of a motion jpeg stream.

2.5.4 MP4 Analyzers

The MP4 media format is quite complex and easy to get wrong when implementing your own version. Therefore it was essential to use several different MP4 analyzers such as Isoviewer [43] (Figure 2.9) to validate and inspect the internals of the MP4 media format.

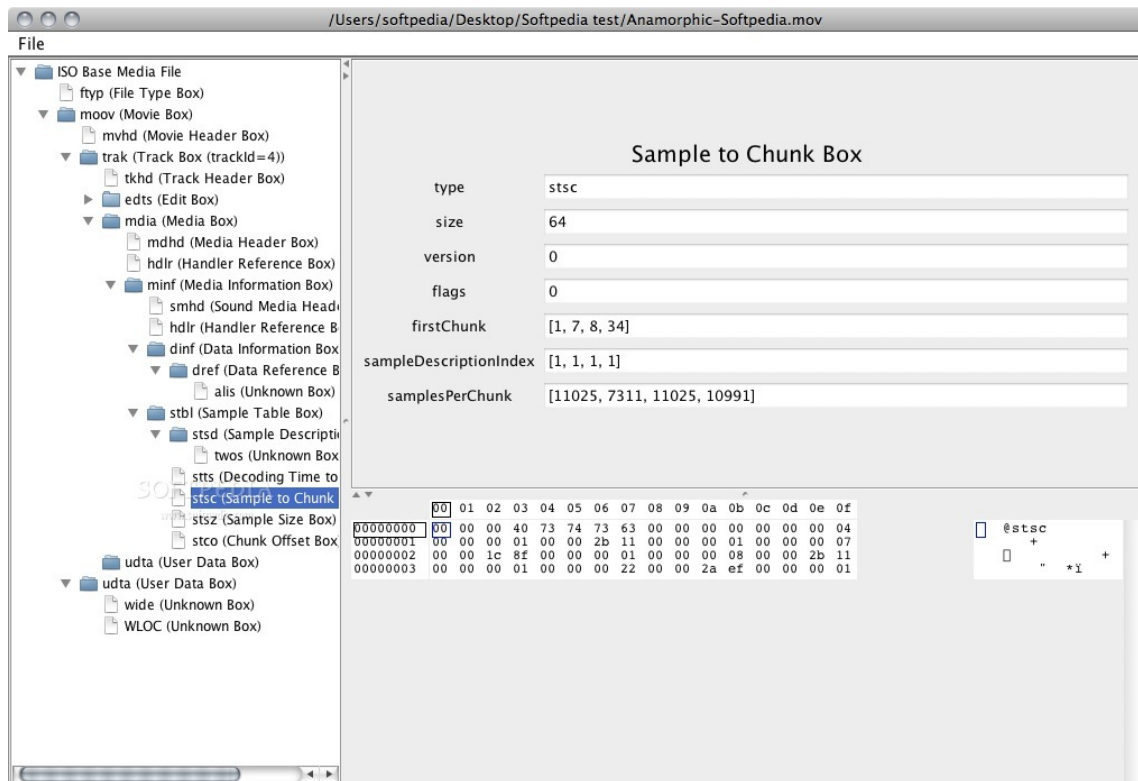


Figure 2.9: Isoviewer.

Chapter 3

Analysis

To design an acceptable solution we need to have a better understanding of the problem. In this chapter we will list all requirements set by us and Axis Communications. We will also discuss different ways on how to approach the problem and what browser support we can expect.

3.1 Requirements

Browser compatibility Since there is a variety of browsers in use today it is important that the solution is cross-platform which basically means that it should work independently of the browser implementation and underlying operating system. Cross-platform support is often an unrealistic requirement when using cutting edge technology as we are dependent on the browser implementation and its functionality and that functionality varies depending on the browser vendor and version. However, it is possible to provide the functionality to those who have the ability to use it and provide fallback solutions for those who does not. As a last resort you should at least provide a hint that the user should update his browser. Regardless you should always try to provide a working solution for the majority of the market but it will always be a trade off between “cutting edge” features and backwards compatibility.

OS & device independent As previously mentioned a solution should work regardless of the underlying operating system or device as long as a compatible browser is installed.

Latency In relation to real-time video streaming latency is the delay from which a camera recorded a video frame to when it is displayed on screen. In real-time streaming we aim for instant feedback e.g. when someone waves their hand in front of the camera you would like to see the hand waving on screen instantly and not five seconds later. To achieve instant feedback we need low latency which means that we need to send

out the video frames as soon as they are ready and the frame size has to be small and optimized so the frame can travel faster across the network. Our solution should have a latency below 300 ms.

Bandwidth Bandwidth is the amount of data that can be carried from one point to another, expressed in bits per second or multiples of bits per second e.g. kbit/s, Mbit/s, Gbit/s etc. The bandwidth is usually the bottleneck in streaming applications. With a low bandwidth it will take longer for the data to travel across the network which implicates that you will get a high latency and long delays between recording and presentation of video frames. With this knowledge in mind it is important to design a solution where each frame is small in size without losing too much quality.

Quality The quality of video and audio is always important for the overall experience. Therefore we should not alter the quality of the initial stream in any way.

CPU usage Even though the camera can be considered a small computer it has limited processing power so the solution has to be lightweight. It is not feasible to let the camera manipulate the video stream into several different formats just to fit our needs as this would heavily increase the camera's workload.

Mobile devices Mobile devices such as smartphones and tablets are becoming increasingly popular and a solution should work on these devices as well.

3.2 Support analysis

Before a prototype can be developed we need to investigate the video support in different browsers. The browser is considered to be our client and that unfortunately imposes some restrictions on what we can and cannot do compared to a plugin-based solution where you tend to have access to a wider range of features and functionality. IP cameras usually have support for either MJPEG and/or H.264 so these formats will be our main focus.

MJPEG MJPEG has native support in most modern browsers (Table A.1) by simply putting the url to the MJPEG stream in the `src` attribute of the `img` element like Listing 3.1.

```

```

Listing 3.1: Motion JPEG stream HTML markup.

H.264 The support for H.264 is more limited. Some browsers support the format via the `HTMLVideoElement` when packaged in an MP4 container as seen in Table A.4. As H.264 is a proprietary format some browsers rely on OS or hardware support to avoid patent issues. This support is based on recorded media but for real-time streaming we need to continuously append a bytestream to the source element which is only possible with the new Media Source Extensions API. This technology is still in development and is as of today not very well supported, see Table A.3.

3.3 Possible approaches

There are two apparent problems with designing a solution for browser based real-time streaming: the transport of data and transformation of video into a suitable format. Browsers do not have any support for RTP and RTSP so we need to find alternate ways of transporting the stream to the browsers. And as discussed we are bound to the browser and its support for different video formats so we need to transform the stream into a suitable format if it is not already supported.

3.3.1 Transport via proxy

As the cameras we use are not able to transport H.264 over HTTP we need some sort of proxy that tunnels the stream to the clients. The proxy can be put either outside or as a part of the camera as shown in Figures 3.1 and 3.2. The proxy can then relay the data with any suitable protocol e.g. Websockets or HTTP.

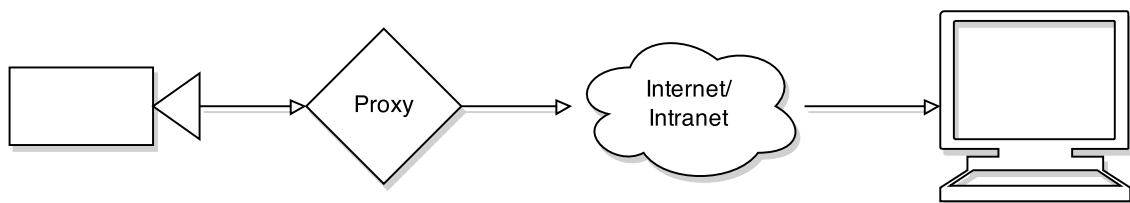


Figure 3.1: External proxy server.

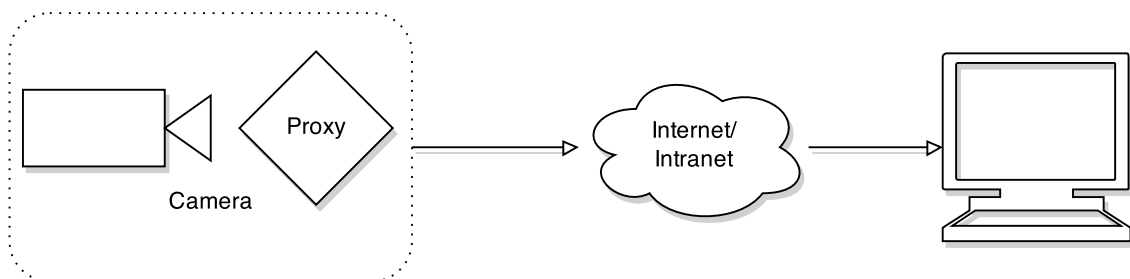


Figure 3.2: Proxy integrated with the camera.

3.3.2 Stream transformation

As discussed in section 3.2 we have a varying degree of browser support for video streams. To make the browser accept the stream we need to transform it into a suitable format. This can be done either by transcoding or multiplexing.

Transcoding

One of the easiest solutions would be to simply transcode the stream into multiple formats for a wide support across different devices and browser. Transcoding is the process

of decoding and re-encoding the stream into another format. The transcoding could be done either by the camera itself, a private media server or in the cloud with services like Amazon Elastic Transcoder. An approach like this would be fairly easy to implement and would have a very high compatibility across browsers and devices but the problem is that transcoding is very CPU intensive, results in loss of quality and induces relatively high latencies which makes it unsuitable for real-time streams.

Multiplexing

Multiplexing or muxing is a method where multiple streams is combined into one. In media streaming this basically means combining video and audio into a container. As seen in Table A.4 the only browser support for H.264 is when the stream is muxed into an MP4 container. By doing this we can skip the hassle of transcoding and simply package the original stream into a suitable container.

Chapter 4

Solutions

Solving the problem of real-time video streaming with HTML5 could be approached in a variety of ways as discussed in the previous chapter. Our main focus has been put in three different solutions: IE support for MJPEG via websockets, client side H.264 decoding and MP4 fragmentation. All our solutions are based on websockets which is currently not supported by the cameras we use for testing. As Websockets is not authenticated in the same manner as HTTP we also briefly discuss token based WebSocket authentication.

4.1 MJPEG

Unfortunately there is no official support for Motion JPEG in any version of Internet Explorer. The problem mainly lies in Internet Explorers inability to handle multipart data because more or less every browser for the last decade has the ability to show regular JPEG images. Theoretically it should be possible to feed single images in rapid succession into the HTML `img` element to get something that looks like a real-time video stream.

To achieve this we will not be able to send the images directly via HTTP as this would require the browser to make a request for each image as previously illustrated in Figure 2.3 which quickly becomes inefficient when sending somewhere between 15-30 frames/images per second. The usage of Websockets would be a better solution as it would enable you to push data directly to the browser. The only caveat is the need to send binary data which is a fairly new addition to the WebSocket protocol. Browsers capable of handling binary Websockets is described in Table A.2.

As it turns out it is possible to display a real-time MJPEG stream in Internet Explorer 10+ by using a single `img` element and a couple of rows of JavaScript. This is achieved by extracting and sending single image frames from the camera via Websockets and then feeding them to the `img` element's `src` tag as a binary large object. A naive implementation is shown in Figure 4.1. This works but has several issues. First of all it depends on the proxy to split the stream into individual frames (see Listing 4.1 for a Gstreamer example)

and furthermore it does not wait for an image to finish rendering before it tries to render another. It would be better if we could wait until an image has been rendered and simply throw away any received images during that time to keep in sync as well as splitting the stream client-side to ease the cameras workload or use some form of adaptive streaming solution.

```
1 // Fetch the img element from the DOM
2 var element = document.getElementById('camera');
3 // Create a new WebSocket
4 var socket = new WebSocket('ws://<servername>:<port>');
5 // Set it to interpret the binary data
6 // as a binary large object (blob).
7 socket.binaryType = 'blob';
8
9 // fire function when a message arrives on the websocket
10 socket.onmessage = function(event) {
11     try {
12         // temporarily store the binary data in a variable,
13         // each websocket message contains exactly one jpeg image
14         var data = event.data;
15         // insert the binary data into the image src tag
16         // by creating a DOMString containing an
17         // URL representing the image.
18         element.src = URL.createObjectURL(data);
19         // clear the ObjectURL to free memory
20         URL.revokeObjectURL(data);
21     } catch (error) {
22         // Catch and print any errors
23         console.error("Oops! ", error);
24     }
25 };
```

Figure 4.1: Motion JPEG in IE via Websockets.

```
gst-launch-1.0 rtspsrc latency=50 location="rtsp://<servername>/axis-
media/media.amp?videocodec=jpeg" ! rtpjpegdepay ! <appropriate sink
>
```

Listing 4.1: Gstreamer pipeline, Motion JPEG frame splitting.

4.2 H.264

Axis cameras can output video encoded in H.264. The benefit of using H.264 is that frames can be compressed effectively and in turn utilize the bandwidth more efficiently. The most commonly used browsers have support for H.264 in combination with MP4, with the exception of Opera [36].

4.2.1 Broadway H.264 JavaScript and Websockets

Broadway.js is a JavaScript H.264 decoder developed by employees at Mozilla. It was based on the H.264 decoder from Android and compiled to JavaScript using emscripten¹. Broadway was not intended to be a production ready solution to enable real-time streaming of H.264 in HTML5 but more of a showcase what JavaScript is able to do. Broadway actually works surprisingly well, and one could theoretically use it for supporting real-time streaming if the user device is powerful enough.

Even though Broadway has been optimized to run certain calculations directly on the GPU, it is still very resource heavy and reliant on good hardware to perform well.

Broadway accepts H.264 in annex b format which is decoded and results in a buffer containing the decoded frame as well as the height and width of the frame. This is used to determine the which portion of the buffer should go in which colorspace. The buffer is then drawn as different textures in a `canvas` element and rendered on to the screen.

```

1 function onPictureDecoded(buffer, width, height) {
2
3   if (!buffer || !this.render) {
4     return;
5   }
6   var lumaSize = width * height;
7   var chromaSize = lumaSize >> 2;
8
9   this.webGLCanvas.YTexture.fill(buffer.subarray(0, lumaSize));
10  this.webGLCanvas.UTexture.fill(
11    buffer.subarray(
12      lumaSize,
13      lumaSize + chromaSize
14    )
15  );
16  this.webGLCanvas.VTexture.fill(
17    buffer.subarray(
18      lumaSize + chromaSize,
19      lumaSize + 2 * chromaSize
20    )
21  );
22  this.webGLCanvas.drawScene();
23 }
```

Figure 4.2: Drawing a decoded frame from `broadway.js` to a HTML5 canvas element with the usage of WebGL.

Figure 4.2 shows an example callback for handling a decoded H.264 frame. By calculating the luma size and chroma size the picture buffer can be sub arrayed properly to return each part of the YUV color space.

The benefit of this approach is that it only requires that the browser supports the Canvas API, however the WebGL powered canvas is a lot faster since the GPU can be used to render the picture. Users with no WebGL capabilities may suffer from performance issues.

¹LLVM-to-JavaScript compiler, <https://github.com/kripken/emscripten>

Due to the decoding being done mostly in JavaScript and because of the fact that the canvas needs to be repainted very frequently it causes high CPU usage. This can be troublesome for devices which run on battery as it will decrease battery life. Even though there may be support for mobile devices it is not a feasible solution for video in the browser as it is very hardware dependent, very resource heavy and constrained to a certain format of H.264.

An Axis camera has the capability to output H.264 but not in annex b. To successfully use Broadway the stream needs to be transcoded to H.264 in annex b before sending the frame to the browser via a websocket message. The message is sent in binary and accepted in the browser and feed to the Broadway decoder which produces a decoded frame. Listing 4.2 shows an ffmpeg pipeline to output H.264 in annex b from an Axis camera.

```
ffmpeg -i "rtsp://<servername>/axis-media/media.amp?resolution=320x240"
        -r 30000/1001 -b:a 2M -bt 4M -vcodec libx264 -pass 1 -coder 0 -bf
        0 flags -loop -wpredp 0 -an -bsf:v H264_mp4toannexb -f rawvideo
        pipe:
```

Listing 4.2: FFmpeg pipeline, RTSP H.264 to annex b

There are several drawbacks of this approach mainly the transcoding of H.264 to H.264 in annex b which is a waste of CPU time and will introduce high latency. The latency imposed by decoding the frame in Broadway is heavily dependent on the capabilities of the user agent and user device.

Table 4.1: Browser compatibility: Canvas.

Feature	Chrome	Firefox	IE	Opera	Safari
Standard - HTML Canvas Support	4	2	9	9	3.1

4.2.2 MP4 fragmentation, MediaSource API and Websockets

To use the Media Source Extension the byte stream needs to be in ISO BMFF. Unfortunately to our knowledge there is no IP camera that sends the stream in this way. Normally MP4 is used to package data into a single file but MP4 is a very flexible container format and allows for fragmentation which means that the ISO BMFF byte stream can be created “on the fly”. The H.264 needs to be encapsulated into an ISO BMFF byte stream which in turn can be fed into the HTML5 video tag by creating a `MediaSource` object attached to the video element. A browser without support for the Media Source Extension API has no possibility of transporting the stream into an underlying H.264 decoder.

The approach is to first create a video tag and attach a `MediaSource` and to create a `SourceBuffer` as seen in listing 4.3.


```

1 // get video element and create a media source object
2 var el = document.getElementsByTagName('video')[0],
3     ms = new MediaSource(),
4     sourceBuffer;
5 //event listener for when the media source is ready to accept data
6 ms.addEventListener('sourceopen', function() {
7     sourceBuffer = this.addSourceBuffer('video/mp4; codec="avc1.42E01E"
8     ');
9 }, false);
10 // attach media source to the video element
11 el.src = URL.createObjectURL(ms);

```

Listing 4.3: Setting up a video element with a MediaSource object

Once the callback from `sourceopen` returns the `MediaSource` will change ready state from closed to open which means it is ready to accept data. To minimize latency the initializing segment can be appended as soon as the `MediaSource` object is ready.

The H.264 stream is encapsulated into media segments consisting of a moov and a mdat which is created “on the fly” resulting in a continuous ISO BMFF byte stream. Each media segment is appended to the `SourceBuffer` in the same manner as the initializing segment and played within the video element in the browser.

There are two general approaches to generating the correct ISO BMFF byte stream. You can either create the fragments client-side, i.e. when H.264 arrives in the browser, or server-side before sending the frame. A client-side approach requires every client to fragment the data where as in a server-side approach you only have to fragment the stream once and broadcast the same stream to all clients.

Server-Side with mp4dashmux

MP4Dashmux is a plugin for Gstreamer which creates MPEG-DASH compatible fragmented MP4. It is currently not included in the stable releases and requires a build from the latest Gstreamer development release in order to function.

The benefit of this plugin is that it can just be hooked in with the rest of the gstreamer pipeline and will output the type of fragmented MP4 that is compatible with MSE. There are however issues with using this plugin, it provides the general functionality needed but still lacks the ability to provide small enough segments to minimize delay.

When using `mp4dashmux` one must provide either a duration, which is the duration of each fragment or split each fragment on I-frames. The delay imposed by splitting on a duration is dependent on how long each frame is. For instance consider a video which is produced at 30 fps, each frame is thereby 33.33 *ms*. The lowest duration achievable with `mp4dashmux` is 200 *ms*, which means that every fragment will contain 6 or more frames. The other option is to split fragments on I-frames, which is highly dependent on the GOP size. A GOP size of 1 will be as efficient as MJPEG with every frame being the entire picture. The default setting of the GOP size for an Axis camera is 32, which means that with 30 fps there will be a little more than a second delay for each fragment. Evidently we can not provide video using `mp4dashmux` efficiently without imposing a delay of at least 200 *ms*.

There is also the fact that there will be an overhead for each packet we send, as the raw H.264 is smaller in size than the encapsulated MP4. The mdat will only contribute to a

4 byte overhead but the moov will be significantly larger and is dependent on the amount of samples included in the mdat. The moov consists of several inner boxes and will have a minimum overhead of 96 bytes as shown in Table 4.2. On streams with low movement we can get P-frames with a size less than 100 bytes which means that the fragmentation could result in over a hundred percent size increase of P-frames.

Table 4.2: Fragment header size.

Atom	Size
moof	8 bytes
mfhd	16 bytes
traf	8 bytes
tfhd	20 bytes
tfdt	12 bytes
trun	32 bytes
Total size	96 bytes

Client-Side in JavaScript

A solution entirely in JavaScript introduces several challenges which need to be considered. Even though the JavaScript engines have seen major improvements in the recent years, JavaScript will not run as fast as native code on a computer or device. Running inefficient code on a computer might be fine, since today's computers pack a lot of power. Mobile devices on the other hand do not have as powerful CPUs and GPUs and inefficient JavaScript might cause noticeable performance issues.

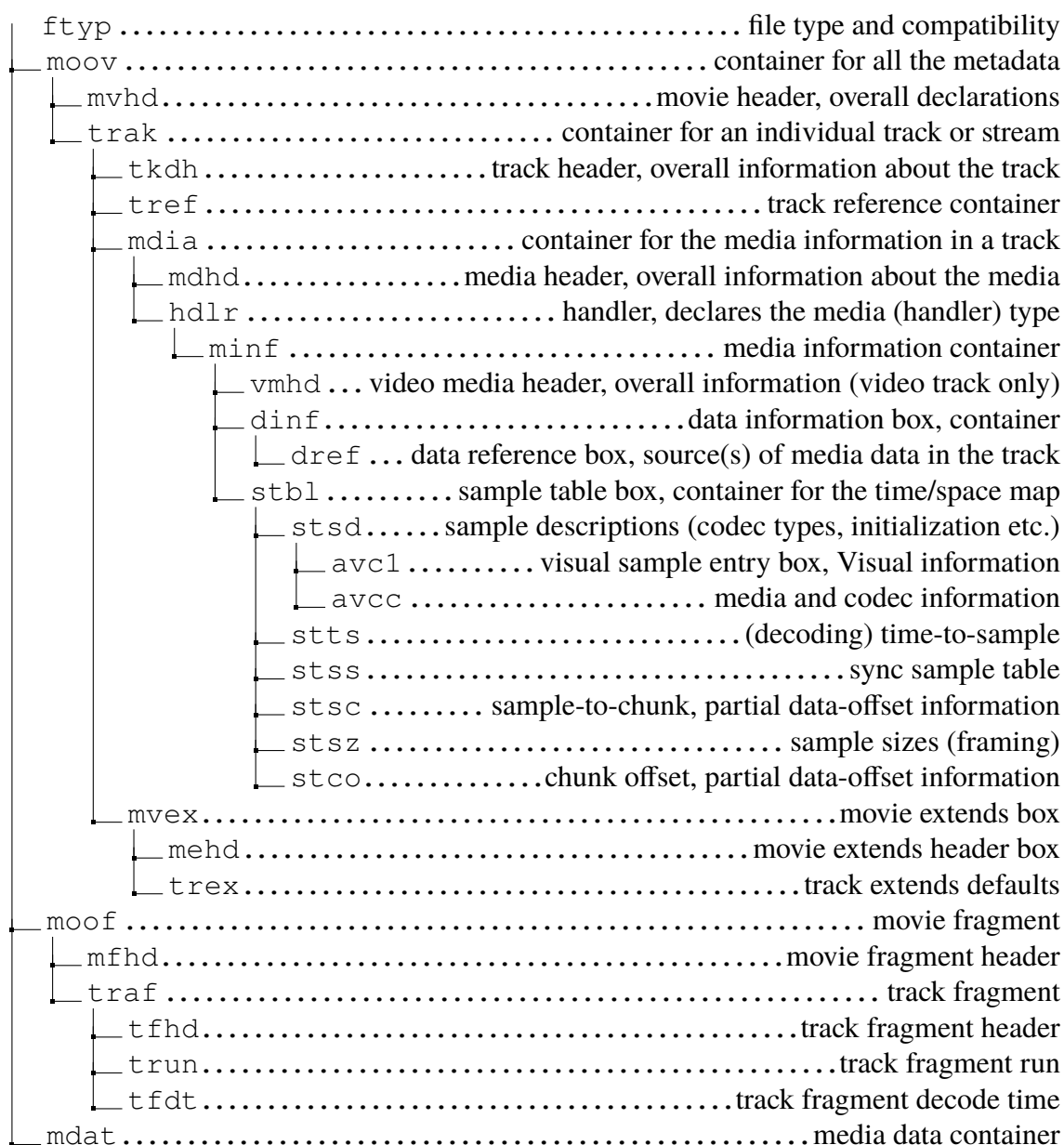
The approach is the same as discussed in the introduction for this chapter only we perform the encapsulation entirely on the client with JavaScript. The `MediaSource` is very picky about which ISO BMFF boxes are contained in the initialization segment and media segments as well as their properties. Figure 4.3 lists the boxes that are required to be included in order for the `MediaSource` to accept the byte stream.

Some boxes have properties which are UTF-8 strings. This causes some problems; as JavaScript strings are UTF-16 we could do conversion between UTF-16 to UTF-8 but it would mean unnecessary computation time would be spent on doing the conversions alone. Instead we define everything that has to do with strings in hexadecimal and assign them to human readable constants.

The ISO BMFF container consists of three main boxes which serve as top level containers for other boxes, moov, moof and mdat.

The moov contains boxes which define meta data and general information about the video of the file. Most of the boxes are of little interest to us since their properties do not affect the ability to render video and thus can be set to default values. However they are still needed to provide the correct ISO BMFF structure discussed in 2.3.2. The boxes that do interest us are the ones that contain properties which are needed to decode the video data.

The most important boxes in the moov are `avcc` and `avc1`. `Avcc` contains the sequence parameter set (SPS) and picture parameter set (PPS) that are used to decode the H.264. The SPS contains parameters which apply to a sequence of coded video pictures and the

**Figure 4.3:** Required ISO BMFF boxes

PPS applies to the decoding of individual pictures inside a sequence [10]. The `avc1` box contains the maximum width and height of the video stream as well as horizontal and vertical resolution.

The initialization segment can be created and appended to a `MediaSource` object before initiating any communication with the camera proxy. Using this technique will not only remove the imposed latency by creating and appending the initialization segment but also ready the source to accept further data.

Each packet sent from the camera proxy will contain one frame and each frame needs to be packetized in a media segment. The motivation for this is simple by only including one frame in each media segment the frame will be appended to the source as fast as possible. This will of course generate a lot more JavaScript objects which are kept in memory, but they are automatically garbage collected rather quickly as they fall out of scope.

A frame will start with an 8-bit field specifying the length of the frame followed by another 8-bit field with the frame type. If the frame is an I-frame we need to set the `firstSampleFlag` property in the `trun` to mark that this frame should not use default flags. By setting first sample flags we enable the decoder to detect that there is a key frame which is followed by one or more P-frames. Typically the `firstSampleFlag` contains information that frames preceding this frame will have a duration and size of their own. Setting the first sample flags for an I-frame will generate the same result as setting the same flags for all of the frames in a GOP.

Each moof in a media segment need to have information of the duration and size of each sample as well as where in time the decoder should start decoding the frame. For each media segment the `baseMediaDecodeTime` property of the `tfdt` box needs to be incremented. The `baseMediaDecodeTime` determines where on a timeline the decoder should start when decoding the samples and is incremented by the total duration of all samples preceding the current one. As for the `mdat` it only serves as a container for the H.264 data and the `trun` in the moof references the sample sizes, duration and where in the `mdat` the actual data starts.

Once the media segment is created it is appended to the `MediaSource` object in the same manner as the initialization segment. By controlling the way we create each fragment we can generate different buffer models depending on what we want to accomplish.

4.3 Authenticating Websockets

As we are dealing with security cameras it is important that the videostream is properly protected. It is a common misconception that just because you have an authenticated HTTP session the WebSocket would be secure as well. The WebSocket should be seen as an entirely different protocol.

Fortunately there is a couple of generally accepted solutions for WebSocket authentication. Our preference is the token based solution as illustrated in Figure 4.4.

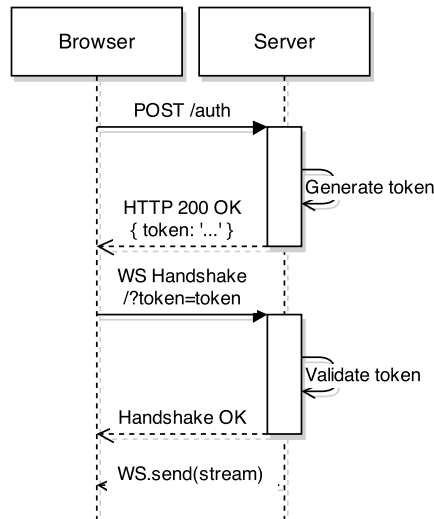


Figure 4.4: Token based WebSocket authentication.

Basically the server generates and returns a unique time-based token to the client when a user logs in using some ordinary authentication mechanism like HTTP Basic Auth. A WebSocket connection is then established and the token is sent as the first part of a handshake. The server verifies the received token with the one generated and if they match, the handshake will be completed and the server will start streaming the video.

Chapter 5

Evaluation

Solutions are great and all but how good are they? To answer that question we need to evaluate and perform measurements on our solution so we can get some metrics on exactly how good it is. In this chapter we will concentrate on evaluating latency and bandwidth as well as CPU and memory usage.

5.1 Test setup

There are several outside factors that can cause differences in tests and measurements. We evaluated our solution in an isolated environment without any outside network traffic and with the following hardware:

- Network Camera
 - AXIS Q6035 PTZ Dome Network Camera
- Switch
 - NETGEAR ProSafe GS108P 10/100/1000 8-port PoE switch
- Computer
 - Intel® Core™ i7-4770 CPU 3.40GHz
 - 2x8GB Kingston DDR3 1600Mhz
 - Nvidia GeForce GT 640
 - 1 Gbit/s Ethernet

5.2 Latency

The latency is measured end-to-end by simply enabling the timestamp on the camera and filming the screen while taking screenshots. The difference between the two timestamps in the picture (Figure 5.1) gives us a latency with an accuracy of one hundred of a second. Measurements of a couple of different resolutions and frame rates can be seen in Table 5.1. The measurements is based on hundred observations each.

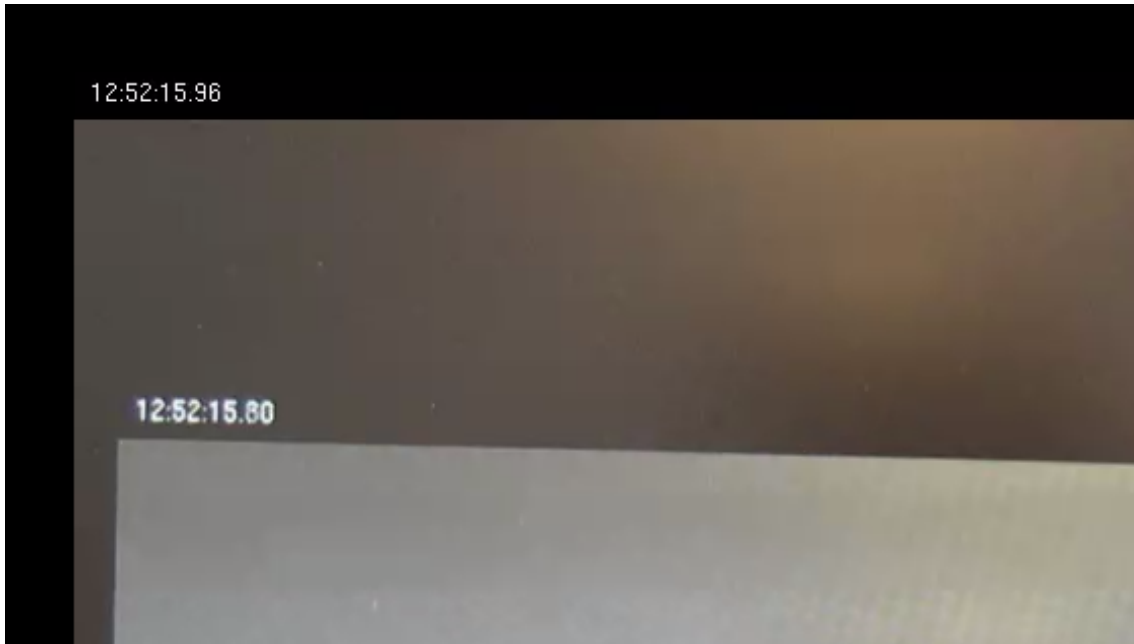


Figure 5.1: Latency measurement method.

Table 5.1: Average latencies.

	12 fps	30 fps	50 fps
800x450	250.4 ms	157.2 ms	141.5 ms
1280x768	251.1 ms	181.1 ms	144.5 ms
1920x1080	331.1 ms	255.4 ms*	N/A

* Measured at 25 fps because of camera limitations.

5.3 Bandwidth

As discussed in Chapter 3, bandwidth is the amount of data that can be carried from one point to another. Our solution carries the exact same video stream as an RTP solution would which means that the only difference is the protocols used to transport the data. In Figure 5.2 we can see that our solution is only 16 bytes larger compared to when transporting the stream via RTP. This means that it will not have as good bandwidth as with RTP but as the overhead is so small it can be considered negligible.

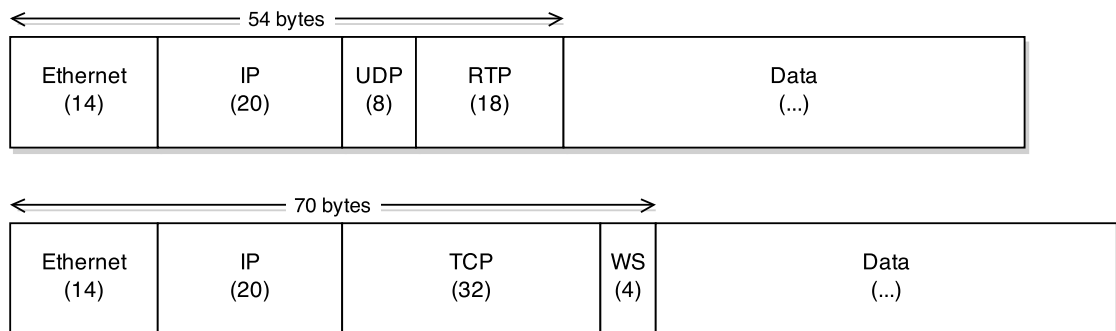


Figure 5.2: Packet header comparison between RTP and Websockets.

5.4 CPU & Memory usage

As the Media Source Extensions API has the ability to use the underlying operating system and hardware for decoding it performs very well compared to JavaScript based decoding solutions such as Broadway. In our test environment the CPU load for the Google Chrome browser never exceeded 15% when streaming video with a resolution of 800x450 at 30 fps. Chrome does store frames in memory for a while but they are eventually automatically garbage collected. With the same stream as previously mentioned the memory usage was in the range of 5-25 MB.

Chapter 6

Discussion & Conclusions

In this thesis we investigated the feasibility to provide a plugin-less real-time video experience with the usage of HTML5 and its new features and APIs.

In chapter 3 we made an in-depth analysis of the requirements and goals of such a solution. By testing different solutions we can conclude that it is indeed possible to provide real-time streaming without plugins but that the solutions are not always ideal. Our main focus lied in investigating the H.264 video format and the Media Source Extensions API but we also looked at some other possible solutions. Mainly MJPEG with Websockets and H.264 decoding in JavaScript. Unfortunately these other solutions fall short mainly because of bandwidth and CPU usage.

MJPEG sends a lot of redundant data with each frame in comparison with H.264 and thus wastes bandwidth. MJPEG has been around for a long time and has a wider browser support but it has never been natively supported in Internet Explorer. Our solution for MJPEG in Internet Explorer is dependent on the browser support for Websockets which fortunately is fairly good.

Decoding H.264 in JavaScript proved to be a very CPU intensive operation. This solution depends on binary Websockets and canvas but the performance would be severely limited in browsers without WebGL support. Through observation we saw that we had latency of a couple of seconds and so this solution was not a suitable alternative, as it did not meet our latency requirement and fell short on performance as well.

We have shown that by using the Media Source Extensions API together with client-side MP4 fragmentation we can achieve low latency real-time video streaming well under our requirement of 300ms and with a high bandwidth. Today's computers, high-end phones and tablets are powerful enough to perform the MP4 fragmentation in JavaScript, so there is no need to perform this step server-side. This solution can definitely compete with third-party plugins and there is still plenty of room for improvements.

The Media Source extensions is still very new, and thus the browser support is not as good as one might wish. At the time of writing there is support for the Media Source Extension in Google Chrome, including Chrome on Android devices, and Internet Explorer

11 on Windows 8.1. Mozilla is currently implementing the API in Firefox and Apple recently announced that OSX 10.10 (Yosemite) will include support for MSE in Safari.

We clearly see that the Media Source Extensions are here to stay. However the current browser support does not fulfill our requirement of compatibility: a solution which works in all major browsers on every platform.

6.1 Conclusions

To summarize we can conclude that for best browser support MJPEG is still the way to go but we feel that the best solution in general is client side MP4 fragmentation combined with the Media Source Extensions. We also think that it is reasonable to fall back to plugin based solutions in older browsers for better support but our hope that such solutions will be unnecessary in the future.

The Media Source Extensions is a very versatile and well thought out API, which covers a lot of use cases. Still, the amount of work put in to actually utilize MSE for real-time streaming is a lot and we hope that browser vendors will eventually propose APIs specifically designed for real-time streamed media.

6.2 Reflection

Much of our work has been covering the absolute latest technology in regard to JavaScript APIs and browser capabilities. Of course this has been exciting but new technology often means lack of proper documentation and involves a lot of trial and error.

This thesis provided us with an opportunity to push the limits, and test both the capabilities, of cutting-edge browsers and the JavaScript scripting language. We are fortunate that we started this thesis when we did as there has been a lot of movement around the development of Media Source Extensions during the last couple of months. MSE has changed and improved a lot but above all it has gained popularity both from the community and the browser vendors. Our belief is that the Media Source Extensions is definitely worth evaluating and it is just a matter of time before the browser support is good enough to justify a migration to an MSE based real-time streaming solution.

6.3 Future work

During this thesis we have got a lot of ideas on how our solution could be extended to provide new functionality or to improve the existing solution. The most important ones are summarized below.

Audio Our focus have been video only. It would be beneficial to provide support for audio as well. And to our understanding the approach to enable audio is similar to that of video, to provide the right type of ISO BMFF boxes and to differentiate what the frame contains.

Client-side recording MP4 fragmentation in JavaScript offers us the ability to generate valid MP4 files in the browser without the need for server side processing. This means one could record video directly in the browser by storing the video frames in a suitable matter for instance in the browsers local storage.

SPS/PPS As of now we have no way to fetch SPS and PPS from the camera. In our experiments we used hard coded values for different resolutions and cameras. The SPS and PPS could be sent as a initialization packet before actually sending the video.

Encrypted Media Extensions Beside the Media Source Extension there is an API called Encrypted Media Extensions (EME) which offers the capability for content protection. Using this with encrypted media will make the browser recognize that media is encrypted and thus requires authentication to view content.

Optimization We have not performed any optimization on functions used in our solution. Especially functions concerned with reading and writing to the buffer can be performed much more efficiently.

Bibliography

- [1] Aaron Colwell, Adrian Bateman, Mark Watson. ISO BMFF Byte Stream Format. <https://dvcs.w3.org/hg/html-media/raw-file/default/media-source/isobmff-byte-stream-format.html>.
- [2] Aaron Colwell, Adrian Bateman, Mark Watson. Media Source Extensions. <http://www.w3.org/TR/media-source>.
- [3] Adobe. Adobe flash player. <http://www.adobe.com/se/products/flashplayer.html>.
- [4] Anne van Kesteren, Julian Aubourg, Jungkee Song, Hallvord R. M. Steen. XMLHttpRequest Level 1. <http://www.w3.org/TR/XMLHttpRequest>.
- [5] Apple Inc. HTTP Live Streaming Overview. <https://developer.apple.com/library/ios/documentation/networkinginternet/conceptual/streamingmediaguide/Introduction/Introduction.html>.
- [6] Arun Ranganathan, Jonas Sicking. File API. <http://www.w3.org/TR/FileAPI/>.
- [7] Axis Communications. Browser support. http://www.axis.com/techsup/cam_servers/tech_notes/browsers.htm.
- [8] Axis Communications. Video compression. http://www.axis.com/products/video/about_networkvideo/compression.htm.
- [9] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. Updated by RFCs 5506, 6904.
- [10] Ben Mesander. The h.264 sequence parameter set. <http://www.cardinalpeak.com/blog/the-h-264-sequence-parameter-set/>.

- [11] Bernhard Weissshuhn. GitHub Repository, node-dronestream. <https://github.com/bkw/node-dronestream>.
- [12] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies. RFC 1341 (Proposed Standard), June 1992. Obsoleted by RFC 1521.
- [13] DASH Industry Forum. DASH Industry Forum. <http://dashif.org/>.
- [14] DASH Industry Forum. GitHub Repository, dash.js. <https://github.com/Dash-Industry-Forum/dash.js/wiki>.
- [15] Douglas Crockford. Javascript: The world's most misunderstood programming language. <http://javascript.crockford.com/javascript.html>.
- [16] Ecma International. Ecma-262 edition 5.1, the ECMAScript language specification. <http://www.ecma-international.org/ecma-262/5.1/>.
- [17] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.
- [18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.
- [19] Behrouz A. Forouzan. *Data Communications and Networking*. McGraw-Hill, 4 edition, 2007.
- [20] Gerald Combs. Wireshark. <http://www.wireshark.org/about.html>.
- [21] Google. Google chrome developer tools. <https://developers.google.com/chrome-developer-tools/>.
- [22] GPAC. Gpac. <http://gpac.wp.mines-telecom.fr/>.
- [23] Ilya Gregorik. *High Performance Browser Networking*. O'Reilly, 4 edition, 2013.
- [24] Ian Hickson, Google, Inc. The WebSocket API. <http://dev.w3.org/html5/websockets/>.
- [25] IANA. Real-Time Transport Protocol (RTP) parameters. <http://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml>.
- [26] ISO. Information technology — Dynamic adaptive streaming over HTTP (DASH) - part 1: Media presentation description and segment formats. ISO 23009-1:2012, International Organization for Standardization, Geneva, Switzerland, 2012.
- [27] Joyent, Inc. Node.js. <http://nodejs.org/>.
- [28] Marijn Haverbeke. Eloquent javascript second edition. http://eloquentjavascript.net/2nd_edition/preview.

- [29] Maxim Lekov. Understanding the mpeg-4 movie atom. http://www.adobe.com/devnet/video/articles/mp4_movie_atom.html.
- [30] Michael Bebenita. Broadway.js - h.264 in javascript. <http://haxpath.squarespace.com/imported-20100930232226/2011/10/28/broadwayjs-h264-in-javascript.html>.
- [31] Michael Bebenita. GitHub Repository, Broadway. <https://github.com/mbebenita/Broadway>.
- [32] Microsoft. What is an ActiveX control? <http://www.microsoft.com/security/resources/activex-what-is.aspx>.
- [33] Mozilla Developer Network and individual contributors. HTML5. <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>.
- [34] Mozilla Developer Network and individual contributors. HTMLMediaElement. <https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement>.
- [35] Mozilla Developer Network and individual contributors. Javascript. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- [36] Mozilla Developer Network and individual contributors. Media formats supported by the HTML audio and video elements. https://developer.mozilla.org/en-US/docs/HTML/Supported_media_formats.
- [37] Oracle. Java. <http://www.oracle.com/technetwork/java/javase/overview/index.html>.
- [38] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980.
- [39] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.
- [40] Iain E. Richardson. *The H.264 Advanced Video Compression Standard*. Wiley, 2 edition, 2010.
- [41] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD), July 2003. Updated by RFCs 5506, 5761, 6051, 6222, 7022.
- [42] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.
- [43] Sebastian Annies. GitHub Repository, isoviewer. <https://github.com/sannies/isoviewer>.

Listings

2.1	Polling with jQuery	22
2.2	A typical gstreamer pipeline, transcoding H264 to vp8.	24
3.1	Motion JPEG stream HTML markup.	30
4.1	Gstreamer pipeline, Motion JPEG frame splitting.	34
4.2	FFmpeg pipeline, RTSP H.264 to annex b	36
4.3	Setting up a video element with a MediaSource object	37
B.1	XMLHttpRequest fetch data example - W3C	61

Appendices

Appendix A

Compatibility tables

Table A.1: Browser compatibility: Motion JPEG.[7]

Feature	Chrome	Firefox	IE	Opera	Safari
Basic support	18	14	Not supported	(yes)	4

Table A.2: Browser compatibility: Binary Websockets.

Feature	Chrome	Firefox	IE	Opera	Safari
Standard - RFC 6455 Support	16	11	10	12.10	6.0

Table A.3: Browser compatibility: MediaSource

Feature	Chrome	Firefox	IE	Opera	Safari
Basic support	(Yes)	25.0*	11*	Not supported	Not supported

Table A.4: HTML5VideoElement, see [36] for a more detailed description

Feature	Chrome	Firefox	IE	Opera	Safari
Basic support	3.0	3.5 (1.9.1)	9.0	10.50	3.1
<video>: VP8 and Vorbis in WebM	6.0	4.0 (2.0)	9.0	10.60	3.1
<video>: VP9 and Opus in WebM	29.0	28.0 (28.0)	?	?	?
<video>: Theora and Vorbis in Ogg	(Yes)	3.5 (1.9.1)	Not supported	10.50	3.1
<video>: H.264 and MP3 in MP4	(Yes)	Partial	9.0	Not supported	Not supported
<video>: H.264 and AAC in MP4	(Yes)	Partial	9.0	Not supported	3.1

Appendix B

Code Snippets

B.1 AJAX Request

```
1 function processData(data) {
2     // taking care of data
3 }
4
5 function handler() {
6     if(this.readyState == this.DONE) {
7         if(this.status == 200 &&
8             this.responseXML != null &&
9             this.responseXML.getElementById('test').textContent) {
10            // success!
11            processData(this.responseXML.getElementById('test').textContent);
12            return;
13        }
14        // something went wrong
15        processData(null);
16    }
17 }
18
19 var client = new XMLHttpRequest();
20 client.onreadystatechange = handler;
21 client.open("GET", "unicorn.xml");
22 client.send();
```

Listing B.1: XMLHttpRequest fetch data example - W3C

Real-time video streaming with HTML5

Popular summary

Marcus Lindfeldt & Simon Thörnqvist

October 4, 2014

1 Introduction

Everyday, people watch and stream video on the web, it may be from video sharing websites such as YouTube and Vimeo or simply watching a friend's video on Facebook. Today, most videos on the internet are pre-recorded but there is an increasing trend to broadcast live events such as concerts, news and sports events.

Traditionally web browsers did not have the capability to display video without the use of installed third-party software. Third-party software may impose security risks and it is therefore common for companies to not allow installation of such software. At the same time it requires a user to manually install the software which can be considered negative from a usability point of view.

With technical advances in modern web browsers there is no longer a requirement to use third-party plugins for watching video as the browser provide this capability. However there are no built-in support for live video streaming. Our goal was to combine new browser technology with network cameras to investigate the feasibility to provide real-time video directly from a network camera to the browser. This would provide users with a professional and seamless experience since they would not be required to install any additional software.

We carried out our work with the help of Axis Communications AB. Axis has for a long time tried to find a way to effectively utilize the capabilities of a web browser to display real-time video streams from their network cameras.

2 Technical background

Live and real-time is not the same thing. A stream can be considered live even if it has a latency of several seconds where as a real-time stream has considerably higher performance requirements.

New browsers provides the capability to use a technology called Media Source Extensions (MSE). MSE enables you to incrementally download small segments of a video instead of downloading the entire file before playback begins. This means faster loading times and the ability to stream large videos by requesting small fragments. It also enables you to dynamically change the quality of the video based on the network conditions.

MSE can be used to perform real-time streaming by using the fact that a small fragment of a file is basically just a stream of bytes when it travels over the network.

Axis current plugin based solution has a latency below 0.3 s and a requirement was to develop a solution equally as good. Axis network cameras use the h.264 video format and so the video frames needs to be packetized in a media container to be compatible with MSE.

3 Approach

When evaluating the problem we decided that the best approach was to simply build a prototype. MSE is very strict about the format of the video stream. By using test driven development and modular design we could develop and test small pieces of our implementation in isolation. This enabled us to overcome the difficulties of creating a valid video stream.

By experimenting with MSE we concluded that an entire fragment needs to be buffered before it can be played back. In theory, by minimizing the amount of frames in a fragment we can obtain a lower latency since each fragment contain the absolute minimum of data.

4 Solution

The previously described technique proved to be very efficient and we managed to display real-time video with a latency of 0.15 s. Which is as good as Axis current plugin based solution.

To limit the amount of data that travels on the network, the raw video was transferred to the browser and the desired video fragments was then created with the browser scripting language, JavaScript. The advantage of this approach is that the camera can do what it does best, record and transmit video while the computational power of the client is used to perform the packetization. This solution limits the workload of the camera and is at the same time cross-platform and cross-browser.

5 Conclusions

Our solution shows that it is possible to achieve pluginless, real-time video streaming with a latency below 0.3 s by using the Media Source Extensions API. However, the amount of work put in to actually utilize the Media Source Extensions API for real-time streaming is significant but it is likely that browser vendors will eventually propose solutions specifically designed for real-time streamed media.