Linköping University | Department of Computer and Information Science
Bachelor thesis, 16 ECTS | Computer Engineering
2018 | LIU-IDA/LITH-EX-G--18/066--SE

HTTP Based Adaptive Bitrate Streaming Protocols in Live Surveillance Systems

Daniel Dzabic
Jacob Mårtensson

Supervisor : Adrian Horga
Examiner : Ahmed Rezine

External supervisor : Emil Wittlock

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art. Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Daniel Dzabic
Jacob Mårtensson

Abstract

This thesis explores possible solutions to replace Adobe Flash Player by using tools already built into modern web browsers, and explores the tradeoffs between bitrate, quality, and delay when using an adaptive bitrate for live streamed video. Using an adaptive bitrate for streamed video was found to reduce stalls in playback for the client by adapting to the available bandwidth. A newer codec can further compress the video file size while maintaining the same video quality. This can improve the viewing experience for clients on a restricted or a congested network. The tests conducted in this thesis show that producing an adaptive bitrate stream and changing codecs is a very CPU intensive process.

Acknowledgments

The years at Linköping University have been interesting, fun, educational, and exciting. There have been a lot of people, friends and family, that we shared this journey with. To you we say - Thank You. We don't want to forget the people working at Café Java. You have always been there for us and without your coffee we wouldn't have come this far.

This work could not have been possible without our supervisor at Verisure, Emil Wittlock. We thank you for this oppurtunity and for all the help along the way.

We also thank Adrian Horga for your help and supervision. Your kind words of encouragement helped us in difficult times. We hope your research goes well.

Lastly, we would like to thank our examiner, Ahmed Rezine. You always had a possitive outlook and took a genuine interest to our work.

Daniel and Jacob
August 2018
Linköping

Contents

Abstract iii

Acknowledgments iv

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1
1.1 Background 1
1.2 Motivation 3
1.3 Mission Statement 3
1.4 Limitations 3
1.5 Disposition 4
1.6 Glossary 4

2 Theory 5
2.1 Quality of Service 5
2.1.1 Video Quality 6
2.1.2 SSIM - Measuring Video Quality 6
2.2 Streaming Protocols 7
2.3 Adaptive Bitrate Streaming 8
2.4 Video File Format 9
2.5 Codec 10
2.6 Hardware and Software Encoders 10
2.7 HTML5 Video 11
2.8 Web Server 11

3 Method 12
3.1 Camera 12
3.2 Web Server 12
3.3 Media Encoding 13
3.4 Test Cases 14
3.4.1 Akamai Support Player 14
3.4.2 Test Video 14
3.5 Survey 14

4 Results 17
4.1 Copy 18
4.2 Adaptive Live Streaming 19
4.3 Live Encoding 20

5 Discussion **22**
5.1 Results 22
5.2 Method 24

6 Conclusion **25**
6.1 Future work 25

Bibliography **26**

A Appendix Test Environment **28**

B Appendix nginx **30**

List of Figures

1.1 Simplified diagram of how a user can be connected in different networks and still be able to access the cameras. Each camera has a unique IP address that the user can connect to. 2

2.1 An example of quality resolution 7
2.2 Simplified flowchart for adaptive bitrate streaming. 8

3.1 Flowchart of the nginx web server. 13
3.2 Results from the survey from Verisure's clients when asked what they thought was most and second most important. 15
3.3 Results from the survey from Verisure's clients when asked what they thought was most and second most important. 16

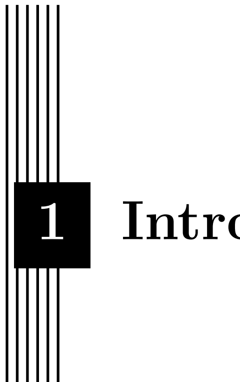
4.1 Diagram of the Copy tests. 18
4.2 HLS and DASH copy compared to the stream from the server. DASH completely overwrites HLS in this figure, since they are identical and DASH is tested *after* HLS. 19
4.3 Live encoding test done with the H.264 codec. The results from the Copy test is also shown here. 20
4.4 DASH ultrafast and veryslow compared to the server video. 21

List of Tables

1.1 Glossary 4

3.1 Results from least acceptable video quality 15
3.2 Results when asked how important bandwidth was for the client 15
3.3 Results when asked what the least acceptable delay was for the client 15

4.1 CPU and memory utilization from Copy tests. 18
4.2 Average SSIM from Copy tests. 19
4.3 Latency between web server and client in a live stream. 19
4.4 CPU and memory utilization on the web server from the live encoding tests 21
4.5 Length of each stream. 21
4.6 Average SSIM index between DASH ultrafast and veryslow when compared to the server video. 21



1 Introduction

Video surveillance is not a new concept. However, these days it is a lot more affordable to get professional grade surveillance for your home as well as your business. A common feature that security companies offer today is the ability to access the surveillance cameras through the internet, so that the user can check up on their home, their business, or their pets. This service is available to the user wherever they are currently located and as long they have a working internet connection. An important aspect with this feature is to keep a good quality of the service while maintaining a sufficient level of security so that no hostile actors are able to access the video stream from the cameras, e.g. through man-in-the-middle attacks. This thesis is not about the security aspects, rather the quality and delay aspects.

One solution for streaming today is using plug-in programs and third-party software that handle playback of the streamed video. One common plug-in program, Flash, is scheduled to be deprecated in the year 2020. This has the potential of becoming a security risk. With modern web browsers, video playback can be achieved without using plug-in programs and only using the tools provided by the web browser.

Alternative methods for streaming video are needed and already available. To ensure that the viewer gets an uninterrupted viewing experience, an adaptive bitrate streaming can be adopted. This way, the quality of the video adapts to the quality of the network the viewer is in. Since video compression is getting better, it is also of interest to analyse how video compression algorithms work and prepare for a future implementation.

Video compression algorithms, or more commonly known as codecs (from encode/decode), have become better at compressing video files to a smaller size without losing too much quality. However, better compressions require better hardware. Newer codecs, such as H.265, are able of compressing the bitrate of a video up to half its size compared to the most widely used codec today [1]. The drawback is that it can require up to ten times the computing power at compression. Another factor to consider is that support is readily available for older codecs, while newer codecs might not have the same support. Fortunately, there are codecs that are open-source and backed by major corporations and gaining more and more support.

1.1 Background

Verisure Sverige AB is a home security company that offers security solutions for consumers. Together with Securitas AB, Verisure offers security solutions to larger companies as well as

small businesses. One service that they provide is video surveillance for homes and workplaces. With this service, their clients can log into Verisure's website and view the cameras connected to their network. This way, for example, a store owner can take a look at their store and see if anything is out of the ordinary while they are away. What they access is a live stream of their cameras. As long they have an internet connection, it doesn't matter where in the world they are. A simplified diagram of how this works can be seen in Figure 1.1.

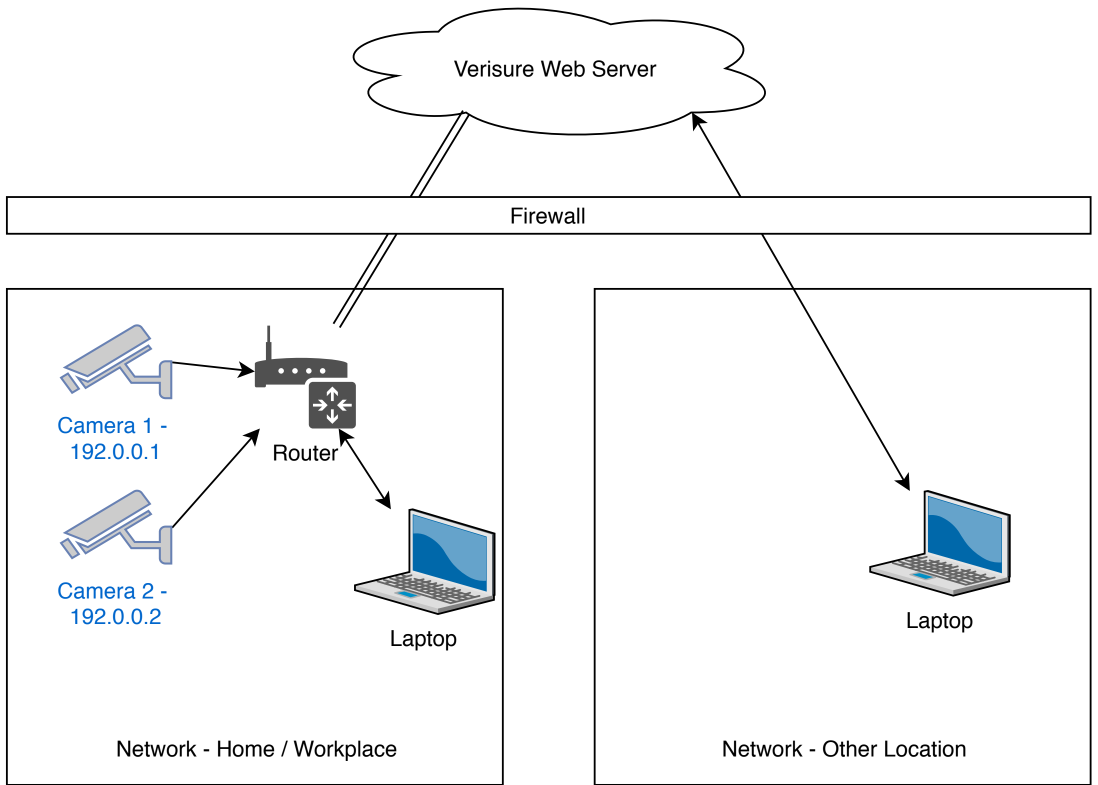


Figure 1.1: Simplified diagram of how a user can be connected in different networks and still be able to access the cameras. Each camera has a unique IP address that the user can connect to.

These cameras also stream their video directly to Verisure's web server. Should an alarm sound at this small store, Verisure's own operators will be alerted. These operators will look at the live video of the cameras to determine what the best course of action is. Since the video is streamed to the web server, a client can access the stream from the server with a secure connection regardless of where they are and what network they are connected to.

The web server constantly receives data from the client's security system. Depending on what type of security system the client is using this data can vary. If the client is using camera surveillance, then the video feed is streamed unencrypted to the web server as can be seen in Figure 1.1. However, the connection between the client and the web server is encrypted, so any attacker getting a hold of this connection would still not be able to view the video stream. This connection can be seen as a tunnel of information being sent between the client and the web server. The tunnel is encrypted, while the data itself is not. In the case where the client is not on the same network as their cameras, they can still access the live feed from accessing Verisure's web server. This connection is different from the former. Here, the video gets encrypted at the server and then sent to the client. To play this video, the client needs to use a plug-in for their web browser, Adobe Flash Player.

The transfer protocol used here is called Real Time Streaming Protocol (RTSP), and other than being encrypted, the video file remains the same as from the source.

1.2 Motivation

Today, Verisure streams the video from their web server to the client. Since Flash is a service that will be deprecated by the year 2020, it is important to find a service that can replace it. Especially for companies like Verisure where this can become an issue of security for their clients.

Instead of finding a service for streaming video, we are going to use the features already available in the client's web browsers. With HTML5, modern browsers are more than capable of just playing a video. This way the client doesn't have to download any extra plug-in programs for their browsers, and companies like Verisure don't have to rely on any third-party support.

To improve the clients viewing experience further, an adaptive bitrate stream can be used. This has the potential of adapting the streamed video to better fit the clients' network by lowering the video quality, should their network perform poorly or be congested. Buffering a live streamed video with a bitrate that is higher than what the client' network can handle will result in a stall in video playback. This results in that the video no longer is considered "live". With an adaptive bitrate, stalls in video playback can be reduced.

It is also of interest to see if changing to a different video codec can improve the streaming experience for the client. Changing to a newer codec can result in a smaller file size while maintaining the same video quality. This means that it would require less bandwidth to transmit the same video as before. However, this has a trade-off. A more advanced compression requires more computing power. This can result in a need of upgrading the hardware of the web server to satisfy the increased demand, and that the user can potentially experience a delay in the live video.

But what do Verisure's clients want? We cannot answer for certain that they don't mind waiting for the stream to buffer to get a better video quality, or that they want to be able to see the live video as soon as possible with as little delay as possible. For this, we have sent out a survey to Verisure's clients. With this survey we will better understand what the customer wants and motivate why we configured the server the way that we did.

1.3 Mission Statement

Our primary purpose is to replace the flash player currently in use and keep the same viewing experience. We also want to find a better method to stream live video. This can be achieved by either using an adaptive bitrate, or lowering the bandwidth needed by using a more advanced video compression.

- Does an HTML5 player keep or improve the viewing experience when replacing a Flash player?
- What are the benefits of an adaptive bitrate stream, and what are the added costs when producing one?
- Do the benefits from reduced bandwidth outweigh the increased processing power needed when switching to a more advanced compression algorithm?

1.4 Limitations

The work done in this thesis is a mission from Verisure. Verisure uses several different types of cameras, but we will focus our studies and testing on their IP-cameras. Specifically, their ethernet bound IP-cameras. The survey sent out to Verisure's clients will help us to better understand what we should focus our work on.

1.5 Disposition

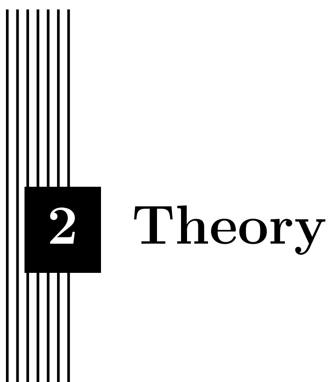
The Theory chapter brings up the different protocols, streaming methods, and video codecs that are relevant to this type of work. The Method chapter presents how a live stream would be received and how it would then be broadcasted using the different protocols, streaming techniques, and codecs that was brought up in the theory chapter. A survey was sent out to Verisure's clients, and it will be discussed at the end of the Method chapter. The Results chapter present how the tests were conducted, what was tested, and the results from these tests. These results later get discussed in the next chapter, Discussion. Lastly, the Conclusion chapter presents what has been learned from this thesis, as well as what can be done in the future.

1.6 Glossary

Table 1.1 below includes a list of abbreviations and terms, as well as their explanation, that will be brought up several times in this thesis.

Term	Explanation
Codec	Coder-Decoder
CPU	Central Processing Unit
DASH / MPEG-DASH	Dynamic Adaptive Streaming over HTTP – A streaming technique for sending video
FLV	Flash Video Format
FPS	Frames Per Second
H.264	Video codec – Most common used today
H.265	Next generation H.264 video codec – Uses a more advanced compression algorithm
HLS	HTTP Live Streaming – A transfer protocol developed by Apple Inc.
HTML5	Hypertext Markup Language – Fifth version
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
MPD	Media Presentation Description – Playlist for DASH
QoS	Quality of Service
RTMP	Real Time Messaging Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SSIM	Structural Similarity Index
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VP9	Video codec
VQMT	MSU Video Quality Measurement Tool

Table 1.1: Glossary



2 Theory

This chapter brings up the technical aspects of the streaming protocols and codecs. The first part of this chapter explains what metrics are used to measure the quality of service for a client receiving live streamed video. The later parts of this chapter explain what a video codec is, and how video can be streamed over a network. How video will be played back to the client will also be explained with small examples.

2.1 Quality of Service

Quality of Service (QoS) is a term used to describe the overall quality of a provided service. Depending on which context it is used in, different metrics apply. When measuring QoS for a live stream, there are many metrics that are subjective to the viewer. There isn't a perfect solution to measure and quantify QoS, however, [2] concluded that the metrics that are most relevant to video streaming are as follows:

- Playback failures
- Start-up time
- Rebuffering
- Video Quality

These four measurements can give us enough quantifiable data and give us a satisfactory measurement of what the client would experience, with the exception of playback failures. Playback failure is more relevant during the development of the streaming service, since it's simply an error if the video stream did not play as it should. Since the video will be played through HTML5 it is possible to obtain error codes and metrics for HTML5 video playback [3]. Start-up time is a measurement of how long it takes for a video stream to start playing once the user opens the video stream. Factors that can increase the start-up time is the available bandwidth in the network [4]. With adaptive bitrate streaming and live encoding of a live stream, it is dependent on how long it takes for the server to generate the video files. The video stream is sent in small chunks over the network. These chunks are kept in a buffer at the clients' video player waiting to be played. If the video is played faster than the buffer can

be filled, playback will stall. This is undesirable and can happen because of congestion in the network. Buffer stalling can be measured in several ways. The three most relevant are:

- Counting the number of times playback is stalled
- The duration of time the playback is stalled
- The frequency of how often buffer stalling events occur

To keep buffer stalling to a minimum, an adaptive streaming is favourable [5]. By lowering the quality of the video stream, the bandwidth needed in the network is reduced. This is a compromise in lowering video quality to lower the need for rebuffering. Adaptive streaming is good if the viewer wants an uninterrupted viewing experience. To better understand what the clients of Verisure want, we have sent out a survey that we will base our decisions to implement adaptive streaming on.

2.1.1 Video Quality

Video quality is a difficult metric to measure, since it can be subjective to the person watching it [6]. It is possible to measure the bitrate of a video, but this is not a good measurement of quality of said video. A high bitrate makes it possible to display more video data, but with advancements in video compression it is possible to almost halve the bitrate in some cases while maintaining the same resolution. Measuring temporal and quality resolution is also possible but can be subjective. A video bitrate is the number of bits needed to process one second of video. This is measured in kilobits per second (kbit/s). The bitrate alone does not give a good measurement of the quality of the video and needs to be combined with the temporal and quality resolution to get a better understanding.

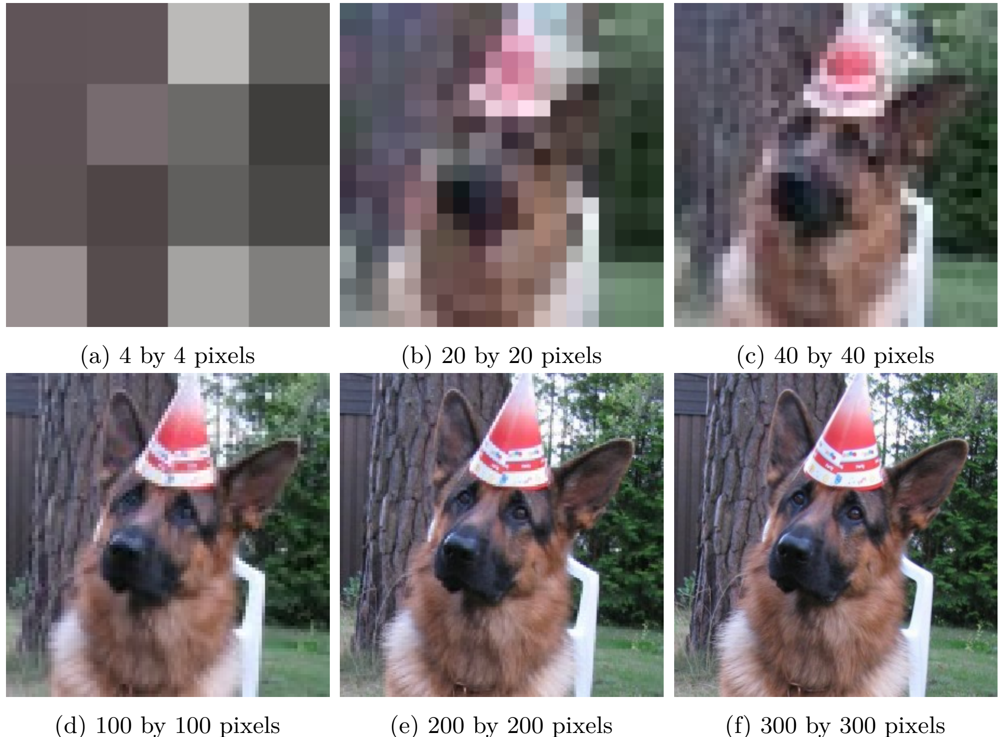
A video is essentially a series of images shown in a fast sequence. With regards to video, an image is called a frame instead. Temporal resolution, or more commonly known as refresh rate, or framerate, is how many frames are shown during one second. This is commonly measured as “frames per second”, or hertz. A low temporal resolution would result in a video that is perceived as rough. A higher refresh rate results in a smoother video.

Quality resolution, also known as pixel resolution, refers to the actual pixel count in a video or an image. More specifically the number of pixels by width and height. Figure 2.1 is a good illustration for understanding how a higher resolution can show more detail. All images inside Figure 2.1 are the same original image but scaled down to a different resolution. They are all displayed at the same size as the original image, but the relative resolution is increased by each image.

Image 2.1a is 4 by 4 pixels. Each individual pixel is very easily distinguished, and it is impossible to see what the image portrays. In 2.1b it is possible to see a shape, but the detail is still not good enough to see what it is. 2.1c has more detail than the previous one and it is possible to see what it depicts, but the finer details are still lost. 2.1d is 100 by 100 pixels, and it is detailed enough to see what the image portrays. 2.1e is 200 by 200 pixels, and the original image 2.1f is 300 by 300 pixels. It is worth noting that at some point the image is detailed enough, and increasing the quality is essentially a waste of extra data needed. Where this point is, is subjective to the viewer since some don’t mind using extra data to increase the quality for a high-definition experience.

2.1.2 SSIM - Measuring Video Quality

Traditionally when using objective methods for assessing perceived image quality, one would quantify the visibility of errors between a reference and a distorted image. Structural Similarity Index (SSIM) measures how structurally similar this comparison is instead by assessing the structural degradation of compared images. This is done under the assumption that the human



(a) 4 by 4 pixels (b) 20 by 20 pixels (c) 40 by 40 pixels

(d) 100 by 100 pixels (e) 200 by 200 pixels (f) 300 by 300 pixels

Figure 2.1: An example of quality resolution

eye is better adapted for extracting structural information from a scene [7]. Since compressing an image can result in degradation, SSIM can be applied here as well.

When measuring difference in quality, SSIM gives the reference image a value of 1. The distorted image is then compared to this reference image and depending on the distortion, a lower value is given. The closer the distorted image is to 1, the more structurally similar it is to the reference image. When assessing video quality, each frame from the reference video is compared to each frame from the distorted video.

2.2 Streaming Protocols

A streaming protocol is used to communicate media such as sound and video over a network. There exists a large variety of protocols with different purposes, and all these protocols are in the application layer. Many of the newer protocols are based on Hypertext Transfer Protocol (HTTP). The protocol used for the transportation layer is Transmission Control Protocol (TCP), or User Datagram Protocol (UDP). HTTP and HTTP Secure (HTTPS) are both application layer protocols used for communication and exchange of data over networks. HTTP works on the request/response semantics and is a stateless protocol [8]. To establish the connection to a server it uses TCP/IP protocol. The type of the data being transferred can be controlled in the HTTP header. HTTPS is very similar to HTTP with the difference being that the communication protocol in HTTPS is encrypted by using Transport Layer Security (TLS) [9], or Secure Sockets Layer (SSL) which is now deprecated and should be avoided [10].

Real Time Protocol (RTP) was released in 1996. It is a real-time protocol, which makes it excellent for real time streaming, but makes it slightly unsuited for TCP even though the support exists. RTP uses a protocol called RTP Control Protocol (RTCP), which helps the protocol with monitoring the data and the amount of packet loss [11]. If there is both audio and video they are separated into two different RTP streams with different ports. RTP works by sending packets containing a header, sources, and payload data. The header contains data

such as the number of sources, what kind of payload is being sent, and a timestamp. Real Time Streaming Protocol [12] is a protocol based on RTP. RTSP was released in 1996 and many of the cameras in use at Verisure use RTSP to stream video to Verisure's web server. RTSP uses TCP or UDP to communicate with the server containing the media. RTP is used for the streaming of the Audio and Video. RTSP is used to control media with play, stop, teardown and other similar functions.

Real-Time Messaging Protocol (RTMP) [13] is a protocol developed by Macromedia for streaming video, audio and messages over the internet. It was initially developed as a proprietary protocol for use between a Flash player and a server, and has been released for public use since its acquisition by Adobe in 2005. It works by splitting the streams into fragments with varying size depending on the connection. It has a small overhead as the header for each fragment is only one byte.

The relevant streaming techniques that are specialized for live streaming are HTTP Live Streaming (HLS), Dynamic adaptive streaming over HTTP (DASH), and the older protocol RTSP. The motivation for picking these techniques was because HLS and DASH can be implemented with HTML5 and are becoming more relevant in regard to streaming today. Adaptive bitrate streaming is discussed further in 2.3.

2.3 Adaptive Bitrate Streaming

Adaptive bitrate streaming is a technique utilized to deliver media content with the ability to vary the quality of the video, depending on the client bandwidth. The video stream is divided up to a number of different quality streams, i.e. a low-quality, a medium-quality, and a high-quality stream. Each of these streams are divided further into smaller parts called fragments, or segments. The fragment is selected based on the client's available bandwidth. If the client's network has a high bandwidth, a high-quality fragment is chosen. If the available bandwidth suddenly changes and gets smaller, a lower quality fragment that suits the available bandwidth better will be chosen. This is to avoid long buffering time and stalls in video playback. Figure 2.2 gives a simplified overview on how adaptive bitrate streaming works.

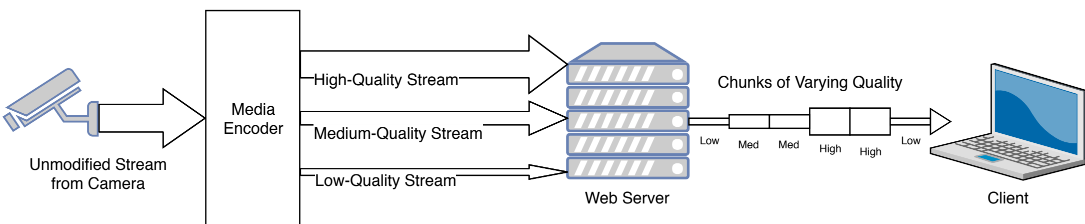


Figure 2.2: Simplified flowchart for adaptive bitrate streaming.

HTTP Live Streaming (HLS) [14] is a protocol developed by Apple Inc, and is a HTTP based streaming protocol. HLS works by dividing up the whole stream into smaller files that can be downloaded over HTTP [15]. Each download is a short fragment of a potentially endless transport stream. A live video stream does not need to have a pre-defined ending, meaning that potentially it can be endless. These fragments are kept track of inside a playlist. The playlist format is derived from the Extended M3U playlist file format and is a UTF-8 text file that contains a Uniform Resource Identifier (URI) for the media, and descriptive tags. These tags describe what type of playlist it is, what the target duration all media segments should aim for, and how long the declared media is since they can be less than the targeted duration. For the client to play video through HLS, the client needs to download the playlist. Then the client downloads the video segments declared inside the playlist. This is handled by the clients videoplayer. An example on how a playlist can look is shown in listing 2.1. This example is taken from [14].

```
#EXTM3U
#EXT-X-TARGETDURATION:10

#EXTINF:9.009,
http://media.example.com/first.ts
#EXTINF:9.009,
http://media.example.com/second.ts
#EXTINF:3.003,
http://media.example.com/third.ts
```

Listing 2.1: M3U playlist example

HLS allows for adaptive bitrate streaming, meaning that the stream can be adapted to the client network without causing any stalls in the video playback. This can be done by using a more complex playlist called a Master Playlist. The Master Playlist is similar to an ordinary playlist but includes a set of variant streams. Each of these variant streams includes a playlist that specifies the same video with a different bitrate and resolution. It is the client's task to monitor the network and switch between the variant streams when appropriate. Each playlist file must be identifiable by the path components of its URI or by HTTP Content-Type.

DASH [16][17], also known as MPEG-DASH, stands for Dynamic Adaptive Streaming over HTTP. It is not a transfer protocol, but a streaming technique that is very similar to Apple's HLS. However, unlike HLS, it doesn't matter what codec the stream is using since it is codec-agnostic. Meaning that DASH can stream any type of encoded streams. This is useful for a company like Verisure, since the cameras that they use could potentially encode their video in different ways. Meaning that they don't have to rely on one particular set of codecs should a better alternative arise.

In the case of DASH streaming, [16] explains that when video is captured and stored on a web server, the video is partitioned into several fragments which contain the media bitstream in the form of chunks. A Media Presentation Description (MPD) is needed to describe the manifest of the available fragments of the video, such as the various alternatives, the URL addresses, and other information. An MPD is needed for each available video on the web server. For the client to play this video it needs to obtain the MPD, which can be delivered by HTTP. By parsing this MPD, the client learns about the video type: its availability, the resolution, and the existence of various encoded alternative bitstreams of the video. For dynamic adaptive streaming to work, the web server needs to produce various bitrates of the video and then serve them. With this information, the client selects an appropriate bitstream and starts streaming the video by fetching fragments using HTTP GET requests. The client continues fetching the subsequent fragments and monitors the network for bandwidth fluctuations. Depending on the network, the client fetches fragments of a lower or a higher bitrate to maintain an adequate buffer. With live broadcasting using DASH, the server will keep capturing and storing new video. These new files will not be described in the current MPD. This means that the server needs to update the MPD, and that the client needs to update the MPD it currently has by sending a new HTTP GET request for a new MPD with regular intervals.

The DASH specification only specifies the MPD and how the fragments are formatted. Delivery and media encoding, as well as how the client adapts to the network, fetches, and plays video are outside of the DASH scope.

2.4 Video File Format

A video file consists of a container such as Matroska, or MP4, to contain video data and metadata in the form of some coding format, i.e. a codec such as those mentioned in chapter 2.5 below. Adobe uses Flash Video (FLV) as a container for sending audio and video. An application is needed to open and play these video files. Since our goal is to play video without

any plugins, a video file that supports playback directly in the browser through HTML5 is needed.

DASH needs to partition the video file into several segments, or fragments as mentioned previously. These segments are then sent to a client. An MPD is needed to describe information about the segments; video resolution, bit rates, timing, and URL. DASH does not need any specific container or encoding, since DASH is a way to present media content between a web server and a client.

HLS works similarly to DASH, as they both need to segment the stream into several parts. HLS needs to download an extended M3U playlist with the required metadata for the various quality streams that are available. The difference is that all media segments in HLS must be in a format described in [14]. This format the media segments need to be in is the MPEG-2 Transport Streams format [18].

2.5 Codec

A codec, short for encoder/decoder, is used to compress video data with the help of a compression algorithm. Each frame in a video is divided into blocks and depending on which codec is used, the number of blocks vary. If no changes were made in one of these blocks from one frame to the next, there is no need to update the data for those blocks. By using this method, the bitrate of the video can be reduced while keeping the same overall quality depending on which kind of codec is used. A common method for video compression is to use a two-dimensional vector, called a motion vector. The motion vector is used to estimate the motion of the blocks for each frame. How it is used is dependent on the codec. If an extreme compression algorithm is used the overall quality of the video will be reduced by leaving compression artifacts. Some factors that can affect the video compression are the frame rate of the video, the complexity of each frame, the pixel density of the video, and the amount of colour displayed.

The H.264 codec is the most commonly used codec when it comes to video [19] and is currently used by a majority of Verisure's cameras as well. H.264 works by dividing each frame into blocks. The size of each block range between 4 by 4 pixels up to 16 by 16 pixels. These block sizes are fixed and do not change throughout the video. Having a smaller block size allows for finer detail at the cost of a larger file size, while a larger block size can leave noticeable artifacts in each frame of the video. This codec divides each frame into one of three types of frames: I-, P-, and B-frames. I-frames are key frames and are self-contained. P-frames are prediction frames and only store information on which block has changed from the previous I-frame which it uses as a reference. The B-frame uses two reference frames, one in the past and one in the future to create a frame with the changes.

Other notable codecs that we intend to test are H.265 and VP9. H.265 is the successor to H.264 and was released in 2013. One of the main differences between the two is how they divide each frame into blocks. H.265 is capable of dynamically allocating blocks with different sizes in one frame, with a maximal block size up to 64 by 64 pixels. Meanwhile H.264 is locked to one block size throughout each frame of the video with a maximum size of 16 by 16 pixels. This allows H.265 to compress each frame much higher with very little loss in detail compared to H.264 [20]. VP9 works much like H.264 and H.265 with a max block size of 64 by 64 pixels. It was developed as an alternative codec since H.264 and H.265 are not royalty-free.

2.6 Hardware and Software Encoders

To encode media there are two methods: using either a hardware encoder or a Software encoder. A software encoder has the benefits of being easy to use. The drawback of this solution is the time needed to encode the video as the encoder will have to share resources with other programs running on the computer. Hardware encoders use specialized and dedicated

hardware to encode media and are very efficient. The drawback is that it is less flexible than a software encoder, but in return the encoding will often be much faster.

2.7 HTML5 Video

HTML5 is the fifth and current version of the HTML standard, a mark-up language used to present content on the internet. It is developed and maintained by the World Wide Web Consortium, W3C [3]. HTML5 is supported in all modern web browsers to a large extent, and it is backwards compatible with older browsers. Browsers automatically handle unknown elements, and because of this it is possible to make older browsers handle newer elements correctly. One of the reasons for its development was to support multimedia without the need for third party support. Companies like Apple have called for more open standards like HTML5 to replace services like Flash [21]. This would allow for cross platform compatibility as well, by utilising HTML5, JavaScript, and CSS. JavaScript, or JS, is a scripting language and allows for interactive and event-driven web pages and applications. CSS is a tool that separates presentation and content, allowing for more flexibility when designing a website and application. Together with JS and CSS, it is possible to play videos with HTML5. With the <video> element in HTML5 it is possible to embed a media player, and with the <source> element it is possible to specify whether the video is sourced locally or externally. The below examples are taken from Mozilla MDN web docs[1].

```
<video>
        <source src="my.mp4" type="video/mp4">
</video>
```

This is an example of playing a video in a web browser without any JS or CSS. In the above example the <source> element specifies that the path to “my.mp4” is local, and that it is an MP4 video. Most modern browsers would be able to run this. To play a video from a URL and not from a local source, simply write:

```
<video>
        <source src="http://example.com/my.mp4" type="video/mp4">
</video>
```

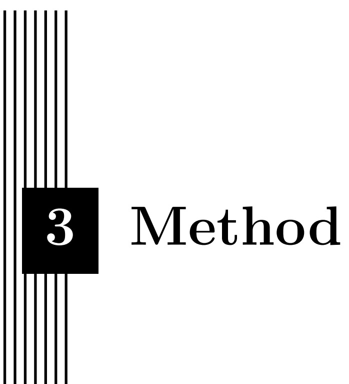
It is possible to add several sources to the same video player, in case the browser in use does not support a certain file extension. This allows for higher flexibility to ensure playback of a video should the client’s browser prefer one type of video over the other.

```
<video>
   <source src="my.mpd" type="application/dash+xml">
   <source src="my.m3u8" type="application/vnd.apple.mpegurl">
</video>
```

2.8 Web Server

A web server is either software on a server or dedicated hardware combined with software. Its purpose is to deliver and store files which can be delivered with protocols such as HTTP, and host webpages. An example of a static content would be a video file, while dynamic content would be the exchange rate for pounds, meaning it will update the information before sending it to the client requesting the information. They are often hosted on the internet or inside of an intranet of a company.

[1] The examples are taken from here: https://developer.mozilla.org/en-US/Apps/Fundamentals/Audio_and_video_delivery/Setting_up_adaptive_streaming_media_sources (2018-06-02)



3 Method

This chapter explains the method used to validate the streaming qualities of the tested solutions. Each subchapter explains the purpose and why we decided on using that method. It includes subchapters for the test environment and the test cases, and the survey that we sent out to Verisure's customers to better understand what they want. The results from all these tests will be shown in Chapter 4.

3.1 Camera

The cameras we used and tested were the most common camera in use today at Verisure, the AXIS M1054 Network Camera.

For testing and comparisons, it is better to use a pre-recorded video file rather than a live stream. This is because a live stream can be difficult to replicate and may not have the same conditions between tests and comparisons. Therefore, to get a proper comparison for how different codecs can change the output it is favourable to use the same video for each test.

It is important to note that the server cannot improve the incoming stream and must work with whatever it receives from the camera. The outgoing stream from the server can only be as good as the outgoing stream from the camera.

3.2 Web Server

To host our web server, we used and nginx 1.14.0. It was chosen for our web server because of its ease of use and the amount of available documentation for it. It also had the feature of being able to include third-party modules for different purposes. The web servers' task is to receive incoming streams, convert these streams to different quality streams for HLS and DASH, and serve them through HTTP to the client.

The web server is responsible for creating the playlist files for DASH and HLS. The nginx web server cannot do this on its own and needs an additional module to achieve HLS and DASH streaming capabilities and playlist creation. A module that offered this was the nginx RTMP module. This module allows nginx to stream media through RTMP, but it also has the added functionality of being able to create and serve HLS and DASH files and their respective playlists for playback through HTTP.

With the help of nginx it is possible to set how long each fragment will be and how long the playlist will be. This can help to control the time it takes to create a playlist and how prone it will be to freezing. How we configured nginx and the RTMP module is explained in appendix B.

For the media encoding on the web server, we used FFmpeg. This is used to manipulate media files, and can be used to do small changes like limit the bitrate of a video to bigger changes such as changing the codec of the original video file. It also has the functionality of streaming a video live. For the tests we sent a video file located on the servers' local storage to the nginx program. The nginx program runs several modules and applications within itself. Figure 3.1 shows a flowchart of the applications we created and how the web server receives and sends a video stream.

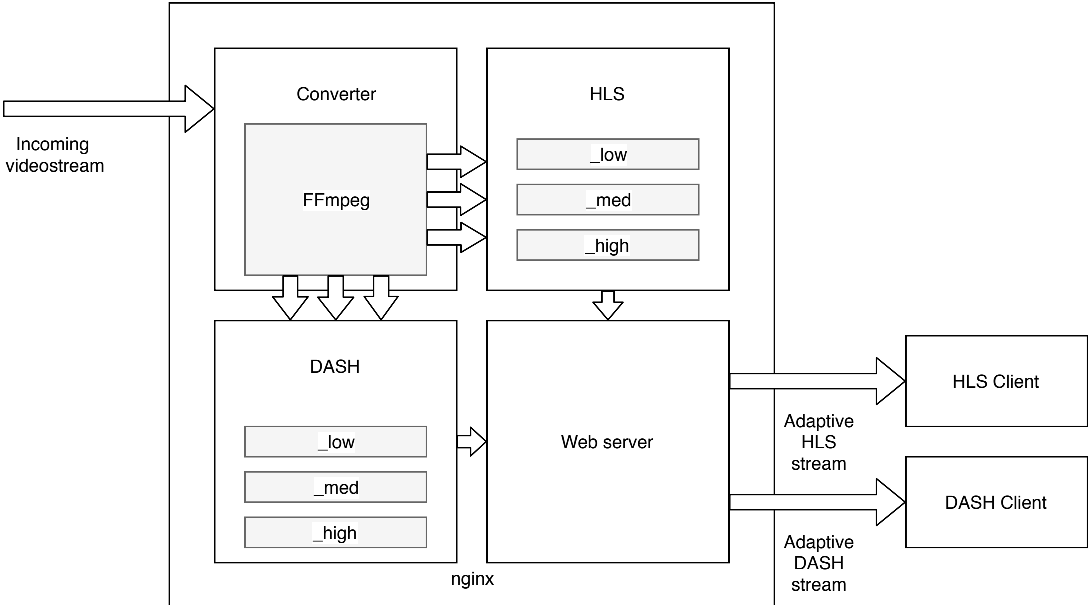


Figure 3.1: Flowchart of the nginx web server.

An incoming stream first gets sent to the converter application that was created inside of the RTMP module. When the converter detects an incoming stream, it executes an FFmpeg command where it converts the incoming stream six times: three times for the DASH application and three times for the HLS application. The three streams to the DASH or HLS application are the same video but in different quality, both in bitrate and resolution.

The nginx software will broadcast live video as long as there is an incoming video stream. If the incoming stream stops, the broadcast stops immediately. All fragments get removed after an unspecified duration, unless specified otherwise in its configuration. It is possible to serve Video on Demand with nginx, but that is not in the spirit of live video broadcasting and is therefore not included.

The hardware specifications for the tested web server as well as the web server at Verisure can be seen in appendix A.

3.3 Media Encoding

With FFmpeg we were able to change the codec on the incoming stream when sending it to the DASH/HLS modules from the converter application, as seen in Figure 4. This was done by creating an application inside of the RTMP block called app. This application doesn't save anything; it simply takes an incoming stream and converts it to different sizes and then sends it to either the DASH or the HLS application.

3.4 Test Cases

The tests were done for both HLS and DASH in adaptive and non-adaptive configurations. The metrics that were tested:

- CPU and memory load
- File size and bitrate of the finished stream
- Video quality of the stream

Information about the CPU load was obtained with the top command. This command displays all running processes and the current CPU usage for each process. Each stream was recorded with FFmpeg on a client receiving the stream. This tells us the size of each stream. This file could then be compared to the original video sent to the server. The video quality will be measured with SSIM, and SSIM will be measured using the MSU Video Quality Measurement Tool (VQMT)[2].

3.4.1 Akamai Support Player

To test the video stream, we used Akamai support player for both HLS and DASH. Both video players use JavaScript to enhance their functionallity and allows them to load HLS and DASH playlists. Akamai is a content delivery and cloud service company that offer a reference player to test video streams. Both video players offer real-time information about the stream, such as current buffer length and download speed. This player was used instead of creating an HTML5 player as it was easier to gather the information about the stream.

3.4.2 Test Video

The test video we used for testing was a recording from one of the AXIS cameras used with a resolution of 720p and a framerate of 25 FPS. The recording was 60 seconds long with a clock in the corner and featured 45 seconds of no movement, and 10 seconds of movement, followed by another five seconds of no movement.

A live stream was tested as well. This was used to measure the delay between the camera and the client when the camera streams video through the nginx web server. The AXIS camera used is able to send live video and add a clock in the corner of the streamed video. By watching the video from the camera next to the incoming stream from the server, it is possible to measure the delay by reading the clock from the camera and comparing it to the incoming stream.

3.5 Survey

A survey was sent out to the clients of Verisure to better understand what the average client thought was important when it comes to watching live video. A total of 162 people answered the survey, and 133 of them currently had camera surveillance.

When it comes to streaming video, it is difficult to achieve a high video quality while maintaining a low latency together with a low bandwidth. It is possible to choose two of these options, but the third option will suffer negatively. Therefore, it was of interest to see which of these options the client's thought was most important. The survey asked the clients to rank what they thought was most important and what was second most important. The results can be seen in Figure 3.2.

[2]MSU VQMT can be found here: `http://www.compression.ru/video/quality_measure/vqmt_download.html` (2018-08-20)

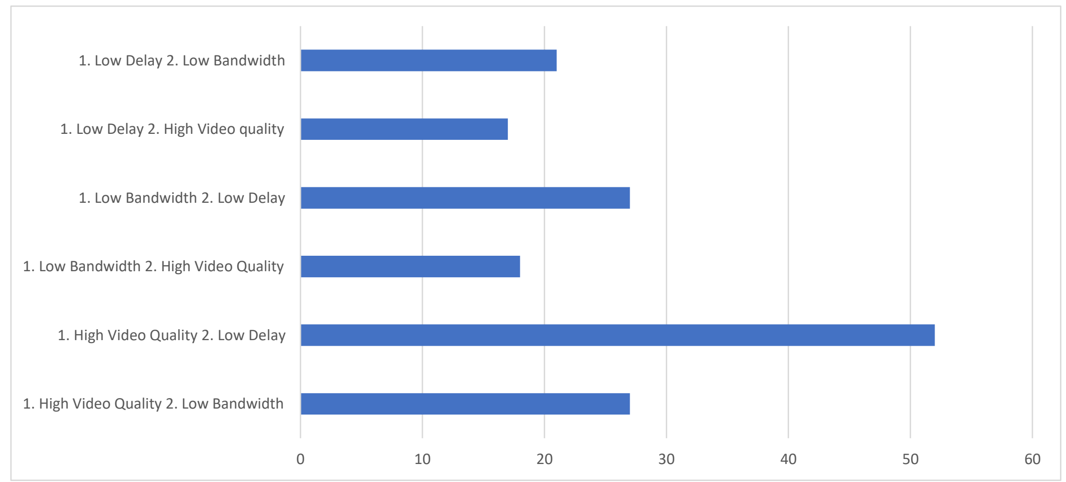


Figure 3.2: Results from the survey from Verisure's clients when asked what they thought was most and second most important.

When asked what the clients thought what the least acceptable video quality was on a scale of 1 to 10, where 1 was only being able to distinguish between shapes and 10 being able to count individual hairs of a cat, the weighted average from 162 clients was 6.88. Table 3.1 shows how Verisure's clients answered.

SCORE:	1	2	3	4	5	6	7	8	9	10
AMOUNT:	8	11	8	7	14	7	23	33	14	37

Table 3.1: Results from least acceptable video quality

When asked about how important bandwidth usage was on a scale 1 to 10, where 1 was not important and 10 most important, the weighted average answer was 6.67. Table 3.2 shows how the results for this question.

SCORE:	1	2	3	4	5	6	7	8	9	10
AMOUNT:	10	3	13	5	10	20	31	29	23	18

Table 3.2: Results when asked how important bandwidth was for the client

When asked what the least acceptable delay was from the camera to the client was on a scale of 1 to 10, where 1 was instantaneous and 10 was up to 2 minutes of delay, the weighted average answer was 4.05. Table 3.3 shows how the clients answered this question.

SCORE:	1	2	3	4	5	6	7	8	9	10
AMOUNT:	27	44	24	17	4	9	6	4	16	11

Table 3.3: Results when asked what the least acceptable delay was for the client

Lastly, the survey asked the clients if they preferred a variable video quality that can adapt to the available bandwidth in their network, or if they preferred a constant video quality, or if they didn't know which they preferred. This is shown in Figure 3.3.

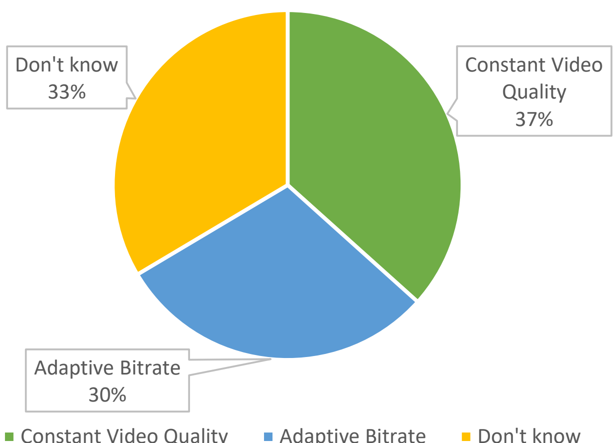


Figure 3.3: Results from the survey from Verisure's clients when asked what they thought was most and second most important.

From these results in Figure 3.2 it is possible to see that video quality is most important, but when asked what the least acceptable quality was on a scale of 1 to 10 most people answered between 7 and 10. A low delay was also preferred, but the video quality should not suffer as a consequence of it. Some people didn't mind a delay up to 2 minutes long, but most people wanted as little delay as possible.

Figure 3.3 shows that one third of the people questioned did not know what they wanted when asked if they preferred a constant video quality or a variable video quality. It is possible that many don't understand what the difference is between an adaptive video and a video that doesn't vary in quality, or that this question wasn't specific enough. High quality video requires larger bandwidth, and when played in a low-quality network it will produce several stalls. This in turn results in a delay, something that the clients did not want.

4 Results

First, a baseline from Verisure's server was needed to compare the nginx configuration with. This was done by streaming 223 seconds of live video from a camera at Verisure and saving it locally on a “client” computer. This camera stream went through Verisure's system of servers and conveniently, the incoming stream was also saved at Verisure's server. It was then possible to compare if the video received by the client had any degradation by measuring the SSIM index between the client's video to what the server broadcasted. Later, the camera stream that the server saved was then used as an input to the nginx web server on the test environment. Appendix A show that the test environment was similar to the server that Verisure used. The nginx web server allows input from several different sources, one of which is using FFmpeg to stream a static video file as if it was a live video, basically simulating a camera streaming to the web server. This way, it was possible to ensure that each test used the same video with almost identical conditions. The tests were divided into three categories:

- Copy
- Adaptive Live Streaming
- Live Encoding

The Copy test entailed that the web server received an incoming stream and sent it without modifying the video in any form. The only work that the web server did was cutting the incoming stream into DASH and HLS fragments and serving them through HTTP. It is then possible to view the stream in an HTML5 player. The Adaptive Live Streaming test is meant to see if using nginx together with the tested hardware is a viable solution for creating adaptive bitrate streams. It was interesting to test and measure how and adaptive bitrate stream would behave and adapt to changes in the network. Last part was Live Encoding. We tested how different encoding speeds could affect the streamed video. These three tests can answer the three mission statements from chapter 1.3.

FFmpeg was used on the client machine to record the incoming live stream. A maximal duration was set to 60 seconds.

To get an objective measurement of video quality VQMT was used to measure the SSIM index between the different video streams. Figure 4.2 and Figure 4.4 were created using VQMT.

The results from chapter 4.1, 4.2, and 4.3 are from the nginx web server on the test environment. The nginx web server was installed at Verisure using appendix B as a guide. An informal comparison was done to compare the results from the test environment, and the results were similar.

4.1 Copy

The first test was to see if the nginx web server is a viable live streaming solution that can be used to replace Verisure's flash player. This was the simplest configuration of the web server, as it just takes the incoming stream and serves it as DASH and HLS files. No other conversion takes place, meaning that the impact on the web server is minimum. A live stream was tested as well to measure the latency between the web server and client. Figure 4.1 shows the framerate, the length, and the bitrate of each video.

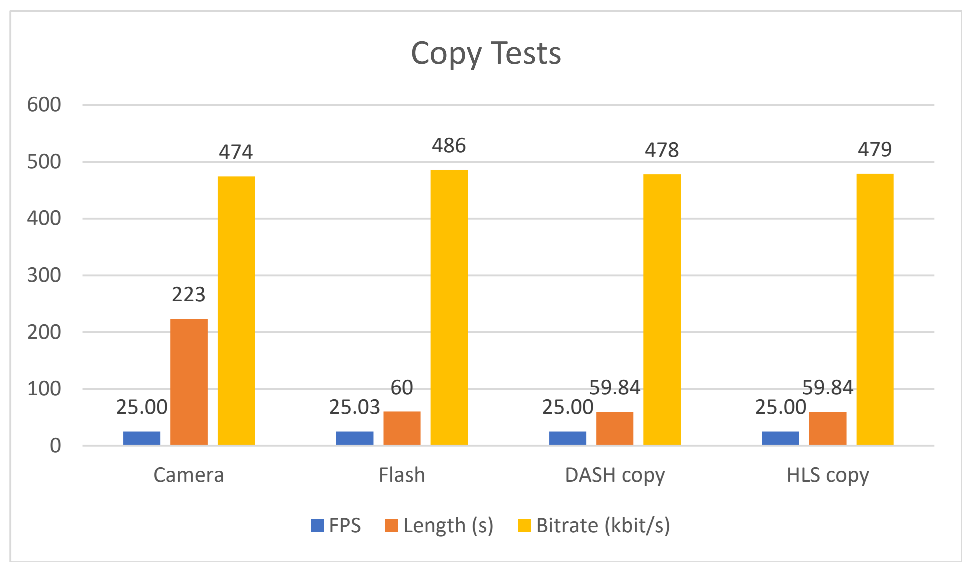


Figure 4.1: Diagram of the Copy tests.

The bitrate of the DASH copy and HLS copy streams are very similar to the original bitrate from the camera. Both DASH and HLS used approximately the same amount of processing power and memory space during the copy test as seen in Table 4.1.

	CPU %	Memory %
HLS Copy	1.0%	3.20%
DASH Copy	0.7%	3.20%

Table 4.1: CPU and memory utilization from Copy tests.

Figure 4.2 shows the result from the SSIM Test when comparing the HLS and DASH stream to the stream outputted by the server. The X-axis shows what frame is compared, and the Y-axis is the SSIM index of that frame. Table 4.2 shows the average SSIM index of both the HLS and DASH stream. Worth noting is that the DASH stream completely overlaps the HLS stream in this test, since they are almost identical. Total number of frames tested for both DASH and HLS was 1469 frames.

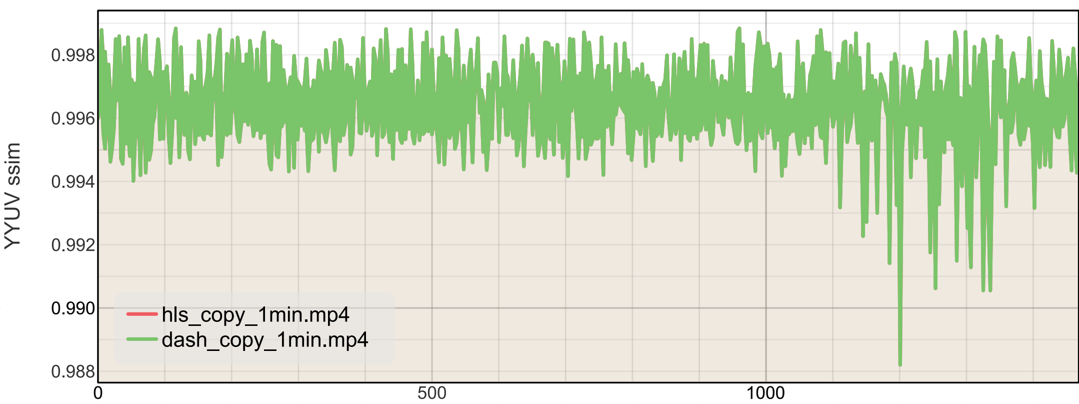


Figure 4.2: HLS and DASH copy compared to the stream from the server. DASH completely overwrites HLS in this figure, since they are identical and DASH is tested *after* HLS.

	Average SSIM:
HLS Copy	0.9964244962
DASH Copy	0.9964244962

Table 4.2: Average SSIM from Copy tests.

As seen in Table 4.2, the HLS and DASH streams are identical in quality to each other but differ slightly from the server stream. For it to be a perfect copy, the SSIM index of each frame should be 1. With a stream copy, an average difference of 0.0036 SSIM units was achieved.

When testing the latency of the web server, a side-by-side comparison was done between the current streaming solution at Verisure and the nginx web server. The camera outputted a clock to show the current time. This time was then compared to the Flash player, the Akamai HLS player, and the Akamai DASH player. All comparisons were done on the same client machine. For both HLS and DASH streams, a playlist length of 3 seconds was used, and a fragment length of 1 second was used. Table 4.3 shows the live stream latency:

	Latency (s):
Flash	<1
HLS copy	8
DASH copy	4

Table 4.3: Latency between web server and client in a live stream.

4.2 Adaptive Live Streaming

This test was done to see if it is possible to switch to a lower quality bitrate stream if the client's network doesn't allow for high bitrate streaming. Unfortunately, the hardware that this was tested on was not powerful enough to handle reencoding of a video stream and broadcasting it live.

When testing, the HLS and DASH clients tried to load a playlist that had not yet been created. Once the playlist was created, the fragments that the video players tried to download were not completed. Once these fragments were completed, they were almost instantly deleted by the nginx clean-up process that removes old fragments to save space on the server. During testing it was sometimes possible to see one or two fragments that the video player was able to load, but later the stream would still crash. Because of this it was not possible to record

any video to do any testing or do any other meaningful measurements. This will be further discussed in the Discussion chapter.

The Live encoding tests go in deeper detail to what happens on the web server during encoding of a live stream and can help explain what the limiting factors are for creating an adaptive bitrate stream.

4.3 Live Encoding

The Live encoding test was done to see if the client can benefit from a different codec or a different encoding speed to achieve a greater compression of the video, while maintaining the same video quality.

The first series of tests were done using the H.264 codec and the same video resolution, but with different encoding speeds. The incoming stream to the web server was the original camera stream recorded by the Verisure server. The nginx web server received this stream and reencoded it with a different encoding speed and then served it through either HLS or DASH depending on the test. The client receiving also recorded the reencoded stream. The three encoding speeds that were tested with FFmpeg were ultrafast, medium, and veryslow. The tests were performed once for each speed for both HLS and DASH, for a total of 6 tests. Figure 4.3 shows the bitrate of each stream after changing the encoding speed, as well as the bitrate from the previous Copy tests.

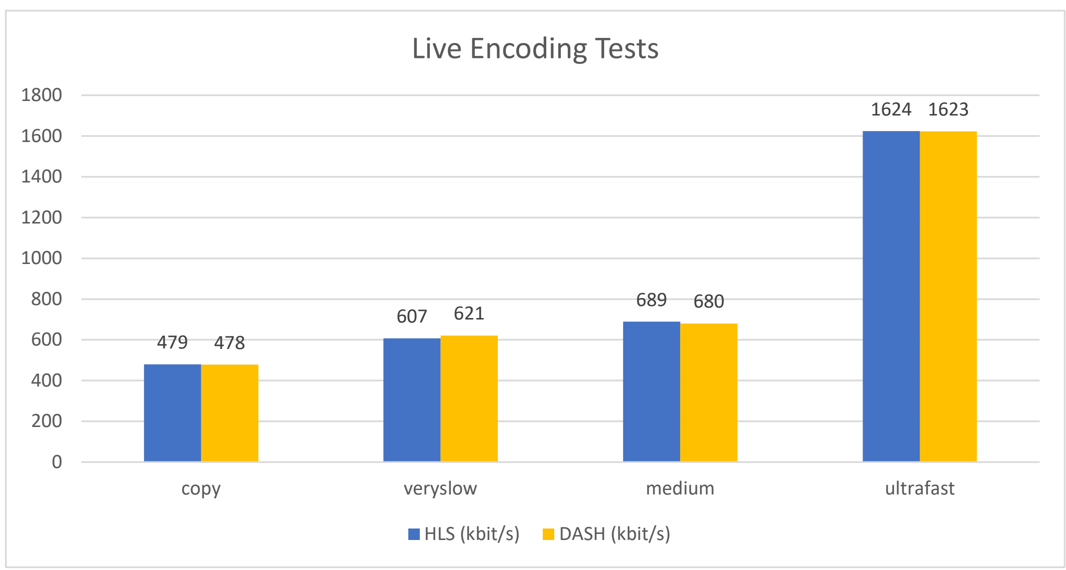


Figure 4.3: Live encoding test done with the H.264 codec. The results from the Copy test is also shown here.

Figure 4.3 shows that the ultrafast encoding speed increased the bitrate for the original stream by almost 4 times the amount when compared to the copy stream. None of the different encoding speeds were able to reach the same bitrate as the copy stream. One factor that can possibly explain this could be the limitations in the hardware. The web server could not produce HLS and DASH streams at the same speed as it receives an incoming video stream, meaning that it needs to buffer more video in the memory. This can be seen in Table 4.4. The length of each stream was drastically reduced as well. When the web server is done receiving video, it will stop its encoding processes and stop broadcasting. For example, the client was only able to record 30 seconds of HLS veryslow video, even though the web server was receiving 223 seconds of video. Table 4.5 shows all the different lengths of video for each encoding speed.

	CPU	Memory
HLS veryslow	98.70%	38.90%
HLS medium	98.70%	20.40%
HLS ultrafast	30.00%	6.10%
DASH veryslow	98.70%	38.90%
DASH medium	98.70%	20.50%
DASH ultrafast	26.60%	6.00%

Table 4.4: CPU and memory utilization on the web server from the live encoding tests

	Length (s)
HLS veryslow	30
HLS medium	50
HLS ultrafast	60
DASH veryslow	23
DASH medium	50
DASH ultrafast	60

Table 4.5: Length of each stream.

Figure 4.4 shows the result from the SSIM Test when comparing the DASH ultrafast and DASH veryslow stream from the client to the stream outputted by the server. The total number of frames compared is 598. Table 4.6 shows the average SSIM index value.

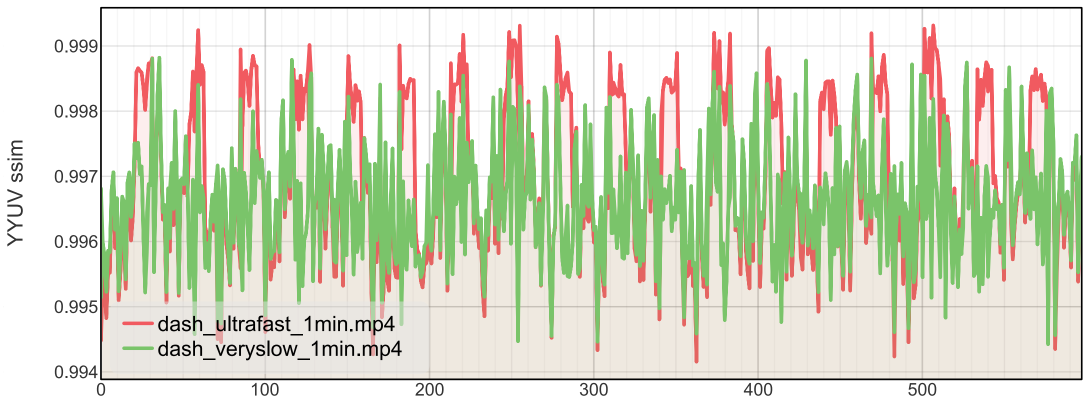


Figure 4.4: DASH ultrafast and veryslow compared to the server video.

	Average SSIM:
DASH ultrafast	0.9970669746
DASH veryslow	0.9966068864

Table 4.6: Average SSIM index between DASH ultrafast and veryslow when compared to the server video.

One flaw with using RTMP for streaming is that the stream is limited to using flash video format. This means that the stream is limited to using codecs that flash video can handle. Unfortunately, the list of codecs that it can use does not include any newer codecs such as H.265 or VP9. It was not possible to change the format to one that can handle newer codecs and still send it through RTMP. This will be discussed in more detail in the next chapter.

5 Discussion

The previous chapter presented the results. This chapter will analyse and comment on the results as well as the method that was used to obtain them. The mission statement is answered at the end of chapter 5.1.

5.1 Results

The results from the Copy tests show that using either HLS or DASH with an HTML5 video player is a suitable replacement for the Flash Video Player. There are no visible degradations of the video, and the average SSIM index for both HLS and DASH is close to the reference stream. One downside to HLS and DASH streaming is the delay between the camera and the client. For HLS and DASH to be able to stream video, a playlist and a video fragment need to be created. With nginx, the playlist is created once the web server has been receiving video equal to the length of one fragment. For HLS this took a longer time. The Flash player seemed to be able to receive video immediately over RTMP in comparison. Video quality and delay was a big concern for Verisure's clients. Many of them wanted as little delay as possible. The delay of the stream could be changed depending on what length the fragments and playlist were set to. Small fragments and playlists would result in a minimal delay, but it would also result in having a minimal buffer. The video player would in turn become more error-prone as it will not have any kind of forward buffer to rely on in case there would be a sudden change in quality of the network.

The results from the Adaptive Live Streaming test were inconclusive. To create an adaptive bitrate stream, there needs to be copies of said stream with different bitrates that the video player can switch between. Creating these different streams proved difficult, since creating a video stream with a lower resolution and bitrate required a reencoding. The Live Encoding test showed how much of a strain a software encoding was on the web server when encoding one single stream. Adaptive bitrate streaming requires several streams to be encoded in parallel. The hardware used in these tests were not powerful enough for this task. This resulted in that the stream had a massive delay, since the playlist told the client video player to download fragments that had not yet been created. When trying to save the streamed video on the client video player for further testing, the video file would be corrupted in the saving process. Different playlist and fragment lengths did not in any way change how the adaptive stream

would behave. The client video player could load the first fragment after a very long delay, and then it would stall.

The Live Encoding test showed that using software for encoding is an extremely CPU intensive task. When using the ultrafast compression speed for the H.264 codec, it was possible to encode it live, i.e. it was possible to send a reencoded stream with minimal delay. However, the ultrafast encoding is made to encode a video as fast as possible without doing any real compression. Testing revealed that the bitrate of the ultrafast encoded stream was about four times the size of the copy stream from the first tests. Even when using the medium and veryslow encoding speeds the bitrate was still higher than the copy stream. It is worth noting that the copy stream was already hardware encoded by the camera. This is where a new codec could prove useful, as the camera is limited to using H.264 when using hardware encoding. Using H.265 with a software encoder could potentially lower the bitrate even further. However, limitations in the nginx RTMP module did not allow for testing of different codecs. Modern codecs do not work with the file format associated with RTMP. The web server receives the incoming camera stream to the converter application through RTMP. The converter application in turn sends reencoded streams to the HLS and DASH applications through RTMP as well. The HLS and DASH applications broadcast video streams through HTTP afterwards. It is when sending reencoded streams to the HLS and DASH applications where the process halts.

Using ultrafast encoding could have created an adaptive bitrate stream much faster, since the standard encoding speed is medium if nothing else is specified. Changing the encoding speed was not tested. However, the Live Encoding tests still show that this would result in an increased bitrate, which is not in the spirit of adaptive bitrate streaming.

The results obtained from the tests in chapter 4.1, 4.2, and 4.3 answer our mission statement from chapter 1.3 as follows:

- Does an HTML5 player keep or improve the viewing experience when replacing a Flash player?

Yes. With HTML5 we can still view video in *almost* the same quality. The SSIM test show that it is not a “1 to 1” copy, however a SSIM score of 0.996 is remarkably good. The delay however is noticeably increased. This is a drawback when using HLS or DASH because of the need of creating a playlist.

- What are the benefits of an adaptive bitrate stream, and what are the added costs when producing one?

Producing an adaptive live stream was so CPU intensive that the server could not send any streams to the client. Creating the different quality streams from the incoming video stream is a very CPU intensive process. A new stream with a different bitrate requires a reencoding, which leads to increased CPU usage. When creating three streams with different bitrates, the CPU started stalling and nginx could not create the necessary files in time. As a result, the client could not view any video. This is a limitation in hardware, and could produce different results with more powerful hardware.

- Do the benefits from reduced bandwidth outweigh the increased processing power needed when switching to a more advanced compression algorithm?

No, not with the tested hardware. The hardware encoder that the camera uses outperform the software encoder at its highest compression setting. However, this result could be different on a more powerful CPU. The RTMP module for nginx can’t use newer codecs such as H.265. A H.265 software encoder could have reduced the bitrate even more, but the encoding time could be increased as well [20].

5.2 Method

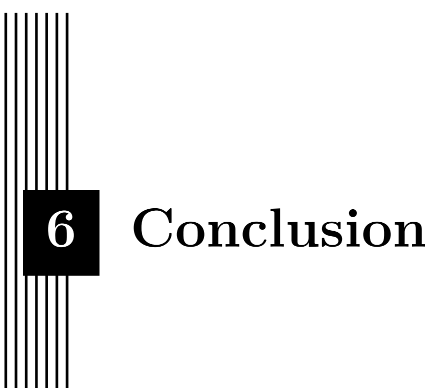
It is possible to criticize the choice of web server for this type of work. There are many commercial web servers that are better suited for streaming video. The nginx web server was chosen because it was open-source, and for its ease of use and available documentation, as well as its ability to load third-party modules. However, using nginx together with the RTMP-module proved to be limiting. There are other modules for nginx that can create HLS and DASH streams, such as the TS-module. This module has the ability to receive and serve video streams through HTTP, meaning a different video file format could be used. However, this module is still in early development and it does not yet have the functionality to execute commands outside of its own module. This means that the converter application that was created and used inside the RTMP-module could not be created inside the TS-module, as it requires FFmpeg for software encoding.

Should a different web server have been chosen, there would still be limitations from the hardware that was used. Software encoding is a CPU intensive process and could have benefitted from more powerful hardware. This could have produced better results when creating an adaptive bitrate stream.

There are alternatives to Apple HLS and MPEG-DASH for adaptive bitrate streaming, such as Adobe HDS and Microsoft Smooth Streaming. However, Adobe HDS requires a Flash player, and because of this it is unsuitable for this work. Both HLS and DASH have broader support than Microsoft's Smooth Streaming.

During testing, the same video file that was recorded from a camera was used. It could have been of interest to test other types of video, each with different complexities.

Using FFmpeg to record was a simple solution, and one where it was possible to set the target duration of each recording. The drawback was in case something would fail, i.e. a missing video fragment as in the case of the Adaptive Live Streaming tests, the recording process would fail, and the recorded video would get corrupted and unplayable. There were cases where FFmpeg did not record the full duration, such as the HLS and DASH copy tests where the last second got cut off. Other solutions for recording the live video stream to the client could have been using a screen recording software. But this is flawed, since a screen recording software only records what is shown on the screen. The SSIM index of DASH veryslow was relatively close to the source video, but the bitrate the video had was almost 4 times as large. This information would not have been picked up by a screen recording software.



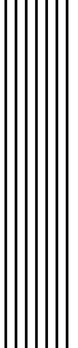
6 Conclusion

Adaptive live streaming is an interesting method to improve perceived quality of service for a client, should they be in a restricted network where streaming live video is a challenge. By simply switching the next video fragment to one from a lower quality stream, a playback stall can be avoided at the cost of reduced video quality. The tests done in this thesis show that this is not that simple, as hardware that is suitable for streaming live video may not be suitable for producing an adaptive bitrate stream. This needs to be considered when implementing such services.

Using HTML5 for video playback allows for a larger audience, since modern web browsers are capable of using it without the need for downloading any extra software and is a suitable replacement for Flash.

6.1 Future work

In the future, development of the different modules for nginx may have progressed to a point where they are better suited for adaptive live streaming. Advancements in encoding could mean a greater support for more advanced codecs, as well as improvements in software encoding. The work done in this thesis could be revisited with hardware that is better suited for encoding live video. Either by using more powerful hardware for the web server or by using hardware encoders.



Bibliography

[1] J. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand. "Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC)". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (Dec. 2012), pp. 1669–1684. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2012.2221192.

[2] Jon Dahl. *The four elements of video performance*. Aug. 16, 2016. URL: https://mux.com/blog/the-four-elements-of-video-performance/ (visited on May 3, 2018).

[3] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, Sangwhan Moon, Erika Doyle Navara, Theresa O'Connor, and Robin Berjon. *HTML 5.2 - W3C Recommendation*. Dec. 14, 2017. URL: https://www.w3.org/TR/html5/ (visited on Apr. 27, 2018).

[4] Ren Wang, G. Pau, K. Yamada, M. Y. Sanadidi, and M. Gerla. "TCP startup performance in large bandwidth networks". In: *IEEE INFOCOM 2004*. Vol. 2. Mar. 2004, 796–805 vol.2. DOI: 10.1109/INFCOM.2004.1356968.

[5] Vengatanathan Krishnamoorthi. "Efficient HTTP-based Adaptive Streaming of Linear and Interactive Videos". PhD thesis. Linköping University, Faculty of Science & Engineering, 2018, p. 72. ISBN: 9789176853719. DOI: 10.3384/diss.diva-143802.

[6] Shahriar Akramullah. "Digital Video Concepts, Methods, and Metrics". In: Berkley, CA: Apress, 2014. Chap. Video Quality Metrics, pp. 101–160.

[7] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (Apr. 2004), pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: https://tools.ietf.org/html/rfc2616 (visited on May 7, 2018).

[9] E. Rescorla. *HTTP Over TLS*. 2000. URL: https://tools.ietf.org/html/rfc2818 (visited on Aug. 3, 2018).

[10] R. Barnes, M. Thomson, A. Pironti, and A. Langley. *Deprecating Secure Sockets Layer Version 3.0*. 2015. URL: https://tools.ietf.org/html/rfc7568 (visited on Aug. 3, 2018).

[11] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. 2003. URL: https://tools.ietf.org/html/rfc3550 (visited on Aug. 3, 2018).

[12] H. Schulzrinne, A. Rao, and R. Lanphier. *Real Time Streaming Protocol*. 1998. URL: https://tools.ietf.org/html/rfc2326 (visited on Aug. 2, 2018).

[13] H. Parmar and M. Thornburgh. *Adobe's Real Time Messaging Protocol*. Dec. 2012. URL: https://www.adobe.com/content/dam/acom/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf (visited on Aug. 2, 2018).

[14] Roger Pantos and May Williams Jr. *HTTP Live Streaming*. 2017. URL: https://tools.ietf.org/html/rfc8216 (visited on Aug. 3, 2018).

[15] "Apple Inc., HTTP Live Streaming Overview". In: *Apple Inc* 1 (2016). URL: https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/Introduction/Introduction.html (visited on May 3, 2018).

[16] I. Sodagar. "The MPEG-DASH Standard for Multimedia Streaming Over the Internet". In: *IEEE MultiMedia* 18 (Nov. 2011), pp. 62–67. ISSN: 1070-986X. DOI: 10.1109/MMUL.2011.71. URL: doi.ieeecomputersociety.org/10.1109/MMUL.2011.71.

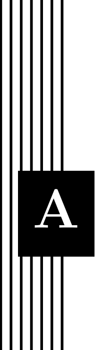
[17] Iraj Sodagar. *White paper on MPEG-DASH Standard*. 2012.

[18] Aaron Colwell, Adrian Bateman, and Mark Watson. *MPEG-2 TS Byte Stream Format*. Dec. 2, 2013. URL: https://www.w3.org/2013/12/byte-stream-format-registry/mp2t-byte-stream-format.html (visited on Aug. 8, 2018).

[19] Benny Bing. *Next-Generation Video Coding and Streaming*. 1st ed. New Jersey: John Wiley & Sons, Incorporated, 2015.

[20] M. A. Layek, N. Q. Thai, M. A. Hossain, N. T. Thu, L. P. Tuyen, A. Talukder, T. Chung, and E. Huh. "Performance analysis of H.264, H.265, VP9 and AV1 video encoders". In: *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. Sept. 2017, pp. 322–325. DOI: 10.1109/APNOMS.2017.8094162.

[21] Steve Jobs. *Thoughts on Flash*. Apr. 2010. URL: https://www.apple.com/hotnews/thoughts-on-flash/ (visited on May 3, 2018).



A Appendix Test Environment

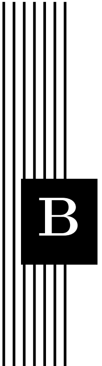
This is the server that was used during all the configurations and tests.

- OS: Ubuntu 16.04 xenial
- Kernel: x86_64 Linux 4.4.0−127−generic
- Shell: bash 4.3.48
- CPU: Intel Xeon CPU E5−2630L v2 @ 2.4GHz
    - Architecture: x86_64
    - CPU(s): 1
    - Thread(s) per core: 1
    - Core(s) per socket: 1
    - Socket(s): 1
    - CPU MHz: 2399.998
    - BogoMIPS: 4799.99
    - Virtualization: VT−x
    - Hypervisor vendor: KVM
    - Virtualization type: full
    - L1d cache: 32K
    - L1i cache: 32K
    - L2 cache: 256K
    - L3 cache: 15360K
- RAM: 992MiB

The information disclosed for Verisure's server:

- OS: CentOS 6.x

- CPU: Intel @ 2.4GHz
    - Architecture: x86_64
    - CPU(s): 1
    - Thread(s) per core: 1
    - Core(s) per socket: 1
    - Socket(s): 1
    - CPU MHz: 2400.000
    - BogoMIPS: 4800.00
    - Virtualization: VT−x
    - Hypervisor vendor: VMware
    - Virtualization type: full
    - L1d cache: 32K
    - L1i cache: 32K
    - L2 cache: 256K
    - L3 cache: 20480K
- RAM: Not disclosed.



B Appendix nginx

This appendix explains how we configured the nginx web server on our test environment through a command-line interface. Verisure used CentOS 6, but our test environment used Ubuntu 16.04. They were able to follow this guide by simply replacing the commands that are specific to Ubuntu 16.04, to commands available on CentOS. The following steps were done on a new installation of Ubuntu 16.04. Root access is needed where required.

Installing nginx

Since this was a new installation of Ubuntu 16.04, we had to get the latest updates and the proper libraries for nginx to work properly. To add modules for nginx, `make` and `gcc` is required. FFmpeg is a tool that will be used frequently as well.

```
apt-get update && apt-get install -y libpcre3 libpcre3-dev \
libssl-dev gcc make ffmpeg
```

To download the right nginx packages with `apt`, it is required to source where from these packages will be obtained from. This was done with the command below:

```
printf "\n\ndeb http://nginx.org/packages/ubuntu/ codename nginx \
\ndeb-src http://nginx.org/packages/ubuntu/ codename nginx" >> \
/etc/apt/sources.list
```

Where *codename* is the version of Ubuntu you are running. In this case it was *xenial.* To eliminate warnings and to authenticate the nginx repository signature, it is necessary to add the key used to sign the nginx packages to the `apt` program keyring.

```
wget https://nginx.org/keys/nginx_signing.key -O - | \
sudo apt-key add -
```

Next part is to download nginx through `apt`. Checking for updates at this point can be beneficial.

```
apt-get update && apt-get install -y nginx
```

If there were no errors encountered, nginx should be installed but not running. By writing `nginx -v` in the terminal it is possible to see what version of nginx is installed. It should be the latest stable version which was version 1.14.0 at the time when this appendix was written. Next part is installing modules.

Installing modules for nginx

All modules need to be compiled into nginx to be able to work. In case you would like to add a new module, nginx would need to be recompiled. However, nginx version 1.11.5 introduced binary compatibility for dynamic modules[3] . This meant that it was possible to add modules dynamically by compiling the module with the open source release of nginx, and then loading the compiled module into your nginx program. It is important to make sure that the open source version is the same version installed through `apt`. Since we used nginx 1.14.0, we downloaded the same version from GitHub, as well as the module we needed for DASH and HLS streaming. Below are the following steps we used.

```
wget http://nginx.org/download/nginx-1.14.0.tar.gz
git clone https://github.com/ut0mt8/nginx-rtmp-module.git
tar -xzvf nginx-1.14.0.tar.gz
cd nginx-1.14.0/
./configure --with-compat --add-dynamic-module=../nginx-rtmp-module
make modules
cp objs/ngx_rtmp_module.so /etc/nginx/modules
```

`make modules` compiles the module with nginx and creates a file called `ngx_rtmp_module.so` that is then copied into the modules directory of the installed nginx program. Loading this module is as simple as adding `load_module modules/module_name.so` to the nginx.conf file, where `module_name` is the name of the module. This is shown in the last part of this appendix document.

Configuring nginx

After all necessary modules are added, it is time to configure the `nginx.conf` file. This file controls all the different directives of the nginx process. The file is located in the `/etc/nginx/` directory. The `nginx.conf` file that we used is at the end of this appendix. The only difference is that the domain name is replaced with `example.com`. The file contains comments that provides some explanation to what each directive does.

When configuring nginx, there is a command that can be used to test the configuration file without running nginx. nginx tests the file syntax and then tries to open files referenced in the configuration file. This is done with `nginx -t`. If there are any errors, nginx will point this out. In our configuration, we wanted all files to be located in `/tmp/streams/`, so the directory `streams/` needed to be created. This was done with:

```
mkdir /tmp/streams
```

At this point nginx was configured and needed to restart its process. This was done with the `systemctl` tool:

```
systemctl restart nginx
```

At this point the nginx process will be running.

PID-bug

It is possible to encounter an error when running the nginx process for the first time. A race condition can occur between systemd and nginx. systemd expects a PID-file to be created before nginx creates it. This results in nginx not being able to start its process. This was solved using the following workaround:

[3] An overview and a guide can be found from the official nginx website: `https://www.nginx.com/blog/compiling-dynamic-modules-nginx-plus/` (2018-07-25)

```
mkdir /etc/systemd/system/nginx.service.d

printf "[Service]\nExecStartPost=/bin/sleep 0.1\n" > \
/etc/systemd/system/nginx.service.d/override.conf

systemctl daemon-reload
systemctl restart nginx
```

This bug was not encounter during testing on CentOS.

nginx.conf file

The `nginx.conf` file has been split between several pages. Line-numbering is kept from page to page.

```
1  user  nginx;
2  worker_processes  1;
3
4  error_log  /var/log/nginx/error.log warn;
5  pid        /var/run/nginx.pid;
6  load_module modules/ngx_rtmp_module.so;
7
8  events {
9    worker_connections  1024;
10 }
11
12 http {
13   include /etc/nginx/mime.types;
14   tcp_nopush on;
15   tcp_nodelay on;
16   client_max_body_size 16M;
17
18   server {
19
20     server_name localhost, .example.com, www.example.com;
21     access_log /var/log/nginx/access.log;
22
23     add_header Cache-Control no-cache always;
24     add_header Access-Control-Allow-Origin * always;
25     listen 80;
26
27     location / {
28       root /usr/share/nginx/html;
29     }
30
31     location /time {
32       return 200;
33     }
34
35     location /dash {
36       types {
37         application/dash+xml mpd;
38         video/mp4 mp4;
39       }
40       alias /tmp/streams/dash;
41     }
42
43     location /hls {
44       types {
45         application/vnd.apple.mpegurl m3u8;
46         video/mp2t ts;
47       }
48       alias /tmp/streams/hls;
49     }
```

```
50   }
51 }
52
53 rtmp_auto_push on;
54 rtmp {
55 # More information can be found on github for each directive.
56 # https://github.com/ut0mt8/nginx-rtmp-module/blob/dev/doc/directives.md#table-of-
       contents
57   server {
58     access_log /var/log/nginx/rtmp_access.log;
59     listen 1935;
60     chunk_size 4096;
61
62 #   This is where the stream will enter the web server.
63     application app {
64       live on;
65 #     Switch between non-adaptive to adaptive streams here by removing comments on
       the "libx264" commands. Remember to comment out the "copy" commands.
66
67       exec ffmpeg -i rtmp://localhost:1935/app/$name
68         -vcodec copy -an -f flv rtmp://localhost:1935/hls/$name
69         -vcodec copy -an -f flv rtmp://localhost:1935/dash/$name;
70 #       -vcodec libx264 -an -vf -s 640x360 -f flv rtmp://localhost:1935/hls/
       $name_low
71 #       -vcodec libx264 -an -vf -s 848x480 -f flv rtmp://localhost:1935/hls/
       $name_mid
72 #       -vcodec libx264 -an -vf -s 1280x720 -f flv rtmp://localhost:1935/hls/
       $name_high
73 #       -vcodec libx264 -an -vf -s 640x360 -f flv rtmp://localhost:1935/dash/
       $name_low
74 #       -vcodec libx264 -an -vf -s 848x480 -f flv rtmp://localhost:1935/dash/
       $name_mid
75 #       -vcodec libx264 -an -vf -s 1280x720 -f flv rtmp://localhost:1935/dash/
       $name_high;
76     }
77
78     application dash {
79 #     Toggle live mode on/off. (one-to-many broadcasting)
80       live on;
81 #     Toggle the DASH application on/off.
82       dash on;
83 #     Toggles nested mode. A subdirectory of 'dash_path' is created for each stream.
        Playlist and fragments are created in that subdirectory. Default: off.
84       dash_nested off;
85 #     NOTE: When sending a single stream, dash_nested needs to be turned off
86 #     Set the DASH playlist and fragments directory. If directory does not exist it
       will be created.
87       dash_path /tmp/streams/dash;
88 #     DASH cleanup: nginx cache manager process removes old DASH-fragments and
       manifests from the DASH directory. Default: off.
89       dash_cleanup on;
90 #     DASH clock: using 'http_head' forces client to fetch header 'Date' from URI.
       dash_clock_helper_uri points to this URI.
91 #     This is only needed for the DASH application.
92       dash_clock_compensation http_head;
93       dash_clock_helper_uri http://localhost/time;
94
95       dash_fragment 3s;
96       dash_playlist_length 9s;
97
98 #     dash_variant is needed when sending multiple bitrates. dash_nested needs to be
        turned on. bandwidth specifies the maximum bandwidth allowed
99
100 #     dash_variant _low bandwidth="64000";
101 #     dash_variant _mid bandwidth="128000";
102 #     dash_variant _high bandwidth="256000";
```

```
103     }
104
105     application hls {
106 #       Toggle live mode on/off. (one-to-many broadcasting)
107         live on;
108 #       Toggle the HLS application on/off.
109         hls on;
110 #       Toggles nested mode. A subdirectory of 'hls_path' is created for each stream.
        Playlist and fragments are created in that subdirectory. Default: off.
111         hls_nested off;
112 #       NOTE: When sending a single stream, hls_nested needs to be turned off
113 #       Set the HLS playlist and fragments directory. If directory does not exist it
        will be created.
114         hls_path /tmp/streams/hls;
115         hls_fragment_naming sequential;
116 #       HLS cleanup: nginx cache manager process removes old HLS-fragments and
        manifests from the DASH directory. Default: off.
117         hls_cleanup on;
118
119         hls_fragment 3s;
120         hls_playlist_length 9s;
121
122 #       hls_variant is needed when sending multiple bitrates. hls_nested needs to be
        turned on. BANDWIDTH specifies the maximum bandwidth allowed
123
124 #       hls_variant _low BANDWIDTH=64000;
125 #       hls_variant _mid BANDWIDTH=128000;
126 #       hls_variant _high BANDWIDTH=256000;
127     }
128   }
129 }
```