Live streaming via WiFi

Monitoring premature babies

C. Athmer
Q. Chen

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

LIVE STREAMING VIA WIFI

MONITORING PREMATURE BABIES

by

C. Athmer
Q. Chen

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Monday July 2, 2018 at 14:00 PM.

Supervisor:	Prof. dr. ir. A. Hanjalic	
Thesis committee:	Prof. dr. ir. A. Hanjalic,	TU Delft
	Dr. O. W. Visser,	TU Delft
	Ir. K. Rassels,	Eya Solutions

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

PREFACE

This report describes the completion of the TI3806 Bachelor's thesis project by Casper Athmer and Qu Chen to finalize the Bachelor Computer Science program at the Delft University of Technology. The project was done parttime over two quarters instead of the usual fulltime over one quarter.

Over the course of five months a livestreaming solution for an existing multi-platform app has been developed for the Delft startup Eya Solutions with the goal of providing a proof of concept that can eventually be used and adapted for automated monitoring of premature babies in incubators. As the proof of concept will possibly be further developed and improved upon, this document contains recommendations for improvements and partly functions as documentation of the implementation, in addition to functioning as documentation of the development process, including analysis, design and reflection.

We wish to thank our supervisor professor Alan Hanjalic from the Multimedia Computing Group at Delft University of Technology for his guidance and Kianoush Rassels and Sander Jobing from Eya Solutions for their help in understanding the existing server infrastructure with multi-platform front-end and their tips regarding C++ and its libraries.

C. Athmer
Q. Chen
Delft, June 2018

SUMMARY

Eya Solutions, a medical startup, commissioned a project to create a livestreaming proof of concept for its multi-platform medical app so that clients can monitor their premature babies remotely with both a livestream and historical Video on Demand (VOD) from either a smartphone app or from a desktop. The plan is to set up one camera per incubator, so one camera per baby. There was a focus on security through authentication and encryption because of it being sensitive patient data and on affordability through the usage of many Free and Open-Source Software (FOSS) tools combined together, so that it may be usable outside First World countries. Additionally scalability was considered in the design and during the process. Maintainability was also thought about, C++ was written with tests via the Catch test framework and not only was time spent searching for an affordable, high-quality camera, it also had to be easily replaceable by being ONVIF-compliant.

The project was very challenging due to its broad scope. It involved many different parts, programming languages and technologies, so the codebase sent to the Software Improvement Group (SIG) for feedback was three separate repos, and these did not include additional configuration files and bash scripts.

Contents

1

INTRODUCTION

Babies have a lot of urgent needs that need to be fulfilled in a timely manner. This is especially true for prematurely born babies that are put in incubators. But when you have a room full of babies in incubators to watch over, fulfilling these needs in time by stationing one nurse per incubator is inefficient and too expensive. So one nurse monitors multiple incubators and attends to needs of the babies, like feeding, on scheduled intervals.

Since every baby is different some babies may have urgent needs outside of this set schedule that escape the nurse's attention. An extra set of eyes that alerts the nurse that is on duty when a baby seems to need something would be very useful. For example, if a baby starts moving a lot, it indicates that the baby is not sleeping and might be hungry. The parents of a prematurely born baby may want to check on their child often as well, but it is not always possible to be physically present, because it might disrupt the work of the nurses. And analysis of the movement and behaviour of these babies can give additional knowledge which can be used to better help them.

A solution for these problems is to aim a camera at every baby and show a livestream of the camera in an app or website, and to also record and store the video on a server, and show this recorded video on request. Motion detection can be applied to the livestream to send notifications to the nurse when a baby starts moving a lot. And the recorded video can be analyzed for research purposes.

Because of time constraints, this project focuses on the first part of this system, which is creating a working proof of concept providing livestreaming and Video on Demand (VOD) functionality. This system can be built upon to add motion detection, and the recorded video can be used for research purposes.

1.1. STRUCTURE OF THIS DOCUMENT

For this project a lot of initial research was necessary, which is described in chapter 2 and 3. In chapter 2 the problem to solve is defined in detail with the problem overview. This problem definition leads to a set of requirements and relevant research questions. Chapter 3 discusses the answers to the research questions and ends with the final design.

Chapter 4 and 5 describe the complete implementation on a high-level and explain the rationale of certain decisions and deviations from the original design in the final implementation. Chapter 6 describes how the quality of the developed software is guaranteed by doing extensive testing. Chapter 7 evaluates the final product, and offers recommendations for future development to the customer. The report finishes with the conclusion in chapter 8.

2

PROBLEM OVERVIEW

In this chapter the problem to be solved is defined and analyzed to come up with practical requirements and research questions. Section 2.1 gives a detailed problem definition. Section 2.2 describes the set of requirements that are derived from the problem definition. Section 2.3 analyzes the given problem and Section 2.4 declares a set of research questions that are answered with the preliminary research.

2.1. PROBLEM DEFINITION

The customer Eya Solutions wants to focus cameras on incubators in hospitals so premature babies can be monitored remotely. The customer wants to provide both a livestream and Vide on Demand (VOD) for each baby. Video should be viewable by multiple different clients simultaneously (max. 10 clients per camera/baby) in any of the major modern browsers on both desktop and mobile, including Mac OS and iOS. Eventually the customer plans to integrate the videostreams into a multi-platform app, however it is not a must-have for this project to complete integration in this existing frontend. Some latency in the livestream is acceptable.

Because premature babies are very vulnerable, the setup should not hinder the work of the nurses. The customer painted a picture of how nurses would hypothetically act if a bulky device was in the way of their work during an emergency, by throwing the device on the floor. It would therefore be nice if the system was non-invasive and useful to these nurses by alerting them if a baby needs attention. The entire cost of this system must be as low as possible, so the solution can possibly be used in poorer countries in the future.

The system must also be secure because it involves sensitive private data, and to ensure that a medical certification can be acquired in the future. The compressed data stream that is transferred over the internet must be secured to prevent an unauthorized entity from intercepting and viewing the video (a so-called man-in-the-middle-attack).

For every camera the client is able to provide an Intel Edison chip (referred to as Edison from now on) which can be used to process the stream from the camera. The Edison has both a Micro-USB slot and WiFi, so it can receive the stream either through a cable or through the local network, depending on the camera choice. The Edison is also used for other processing tasks (e.g. sensor data readings) which should never be interrupted or delayed, so as little processing as possible for the video stream should be done on the Edison.

When the number of concurrent cameras that use the system increases, the processing load of the server also increases. This should be kept in mind when designing and building the system, to ensure that it can be upscaled relatively easy. Another reason is the customer's desire to expand functionality in the future, which could occupy additional resources per camera.

Hospitals will employ multiple cameras which can put pressure on the available bandwidth in a hospital. Therefore the amount of data passing through the router in a hospital should be as low as possible. It is advantageous if the system works out of the box, which means that as few as possible or no settings need to be changed in the router for it to work (e.g. port forwarding).

2.2. PRACTICAL REQUIREMENTS

The problem definition in 2.1 leads to the following requirements in the form of the MoSCoW-hierarchy (Must have, Should have, Could have, Won't have) in table 2.1. Extra requirements that did not follow directly from the problem definition but became clear because of ongoing discussion with the customer during project development are added to this list as well. If all the must-haves are achieved the project will be considered a success.

Priority	Requirements	Reason
Must have	Cheap overall system	So the system can be used in poorer countries
	Encrypted streams	So the system is more secure
	Small physical footprint	So the system does not hinder nurses
	Provide both livestream and VOD	For monitoring and analysis
	Good quality cameras (e.g. 25 fps)	So it delivers high-quality video for possible analysis
	Cameras with infrared (IR) up to 5m	So there can be 24/7 monitoring of a baby
	Show video in any browser	So video is multi-platform viewable
	Good documentation	So the system can get a medical certification
Should have	Low overal bandwidth usage	So it puts little strain on the hospital bandwidth
	Scalable	So the system can help many babies
	Low latency livestream	For better user experience
	Low resource usage per camera	So the system is cheaper to scale
	Easily replaceable cameras	So the system can be easily maintained/expanded
Could have	Authenticated streams	So the system is more secure
	Few/no router settings changes	So it is easy to setup in the hospital
Won't have	Automated video analysis	So nurses can be assisted
	White colored cameras	So the camera blends in better with hospitals
	Adaptive bitrate streaming	So clients can view video in poor network conditions
	Backup recording when network is down	So video is not lost if the hospital network is poor
	Integration into multi-platform app	So the customer has video added to his app

Table 2.1: Requirements table prioritized according to MoSCoW method

2.3. INITIAL PROBLEM ANALYSIS

Using a helicopter view to look at the problem definition in Section 2.1 it appears this livestreaming project is straightforward, akin to simple video surveillance (scaled to more cameras and more viewers than the average

surveillance solution). A deeper look and taking in account the MoSCoW-hierarchy in Table 2.1 shows that there are some major differences with existing video surveillance solutions.

Cost and *security* are arguably the biggest factors that make this project different from existing commercial solutions. While surveillance solutions are ubiquitous and getting cheaper all the time, it is not so easy to serve these streams securely to more than one person without a customized solution. Every client will need his own authentication to access a videostream and every camera its own transport and storage of the stream. And everything must happen securely and cheaply. It is possible to use built-in cloud-based solutions created by manufacturers of IP cameras, but these are not privacy-preserving and often have recurring subscription costs. It is preferred to manage our own private cloud, as this provides better security for sensitive patient's data. It also means easier addition of functionality in the future and smoother integration into the customer's existing medical app.

It is possible to make use of existing infrastructure owned by Eya Solutions, to be specific spare resources on Edisons, with one Edison per camera, and a server running Ubuntu 14.04. Using the Ubuntu server as a private cloud to store and serve the stream securely to a client makes sense when taking cost and security into account. Additionally, by putting this server outside the hospital to serve the clients bandwidth usage in the hospital can be minimized. In this setup increasing the number of cameras increases the bandwidth usage in the hospital, but increasing the number of clients watching streams from the same camera does not.

Another major decision for the architecture is what to use as the hardware used in the hospital (besides the camera). The customer informed us that the Edisons are going to be in the hospital regardless of the livestreaming project, because they are used for other purposes as well. Therefore, if possible, cost wise it would be ideal to also use these Edisons for livestreaming.

2.4. RESEARCH QUESTIONS

Based on the problem definition, requirements and problem analysis, the following research questions have been devised to find a system that meets all the needs previously defined. These research questions were researched and the results can be found in Chapter 3.

WHAT IS THE BEST CAMERA (TYPE) FOR THE SYSTEM?

Two distinct types of cameras are considered: IP cameras and USB cameras. IP cameras transfer their streams over WiFi or Ethernet, while USB cameras transfer their streams through a cable.

This question aims to answer which type of camera is best suited for this project's purpose. While doing this, the most important constraints to keep in mind are that the nurses cannot be interrupted in their work by the camera, which means it should be small and there should be as few cables as possible. The camera should be cheap, support IR Night Vision up to at least 5 meters and it should be easily replaceable.

WHAT STREAMING TECHNOLOGIES SHOULD BE USED?

State-of-the-art streaming protocols should be listed and compared. Based on this comparison the best streaming options for the system can be determined. The most important constraints to consider are:

- The bandwidth usage in the hospital should be as low as possible.
- The video should be playable in all major browsers.
- The user should have the possibility to play a live stream and the recorded stream from a camera.
- The streaming should work without the need to change settings in the hospital router.

HOW TO ENSURE THAT THE STREAM IS ALWAYS TRANSPORTED SECURELY?

Different transport protocols exist for transporting video streams and these transport protocols are secured in different ways. The answer to this question gives an overview of the solutions that exist to ensure secure transport of streams.

HOW SHOULD THE VIDEOSTREAM BE STORED?

The video stream is sent to video storage from where it is served to the client. First, this question aims to find the best type of storage option for serving the video to the client.

Secondly, the question aims to answer on a high level how storage and retrieval of the stored video can be done most efficiently. While answering this question, scalability options are also considered in the event that the chosen solution cannot handle the processing load anymore.

WHAT SHOULD THE ARCHITECTURE OF THE ENTIRE SYSTEM LOOK LIKE?

This question is aimed at giving a high-level overview of all the components in the system and the way in which they interact with each other, to achieve a system as described above. When creating this architecture, special consideration is given to the following issues:

- As little processing as possible should be done on the Edison.
- The load on the router in the hospital should be as low as possible.
- When the stream is sent over the internet it must always be transported securely.
- The entire system should be easily scalable.

3

Research Report

This chapter provides an overview of the research that is done into livestreaming. Section 3.1 discusses which camera type is best to use. Section 3.2 attempts to answer the question which streaming protocol is most appropriate for livestreaming. Section 3.3 details how security can be guaranteed. Section 3.4 discusses various options for storage. And finally, Section 3.5 gives an overview of the complete system architecture that followed from the research.

3.1. High Definition Video Capture and Formatting

There are many cameras and types of cameras to choose from. This section answers the research question "What is the best camera (type) for the system?" Section 3.1.1 first discusses which *type* of camera is best suited for the project and why, comparing camera types on price/quality, cable placement, size, IR, onboard compression and replaceability. Section 3.1.2 then elaborates how this replaceability is achieved using ONVIF. Finally, Section 3.1.3 recommends a suitable camera to fulfill the requirements of the project.

3.1.1. Most suitable camera type

An important requirement of finding the best camera type for the project is taking into account the price of the camera. As the requirements detail, the camera must be relatively cheap to acquire and easy to replace in the future.

A quick survey of existing camera types for sale shows that so called USB cameras (webcams) and IP cameras are the cheapest options. Webcams generally send video data to the client through a cable, usually USB, and IP cameras send the data via Ethernet or WiFi. USB cameras can be cheaper than IP cameras, however both types are pretty cheap, having many options below €20.

Next, it is important to look at which camera types have the least amount of cables. Cables in the wrong places might interfere with activities of the nursing personnel, which was an important concern expressed by the customer. Comparing USB cameras to IP cameras, WiFi IP cameras offer the most flexibility in optimizing the placement of a connected Edison as they offer streaming over a wireless connection to the Edison. If one uses an USB camera it must be connected via USB, and a Wired IP camera must be connected with an Ethernet cable to the router or Edison. For all camera types it holds that they require a power source, usually provided via a cable. IP cameras can minimize the number of cables the most: Wired IP cameras by using PoE (Power over Ethernet), combining data and electricity transmission in one cable, and WiFi IP cameras by using wireless transmission for the data and one cable to provide the electricity.

Other properties are applicable to both camera types. There are small USB cameras, small IP cameras, USB cameras with IR, IP cameras with IR, USB cameras with H.264 compression support and IP cameras with H.264 compression support. The last one, H.264 support is not a hard requirement to have for the camera, but it is useful to have a camera with that property, if one wants to minimize bandwidth usage of video data, keeping in mind future scalability. H.264 compression support directly on the camera itself comes in handy, because then one does not have to encode the raw stream itself and expend valuable processing power in case one wants to scale the number of cameras. H.264 is a video coding standard which aims at halving the bit rate of video streams while providing similar good quality video as earlier standards [1], thus allowing for HD-quality streaming.

The final requirement to compare USB cameras and IP cameras on is replaceability. Here IP cameras are the clear winner, as USB cameras lack the global open industry standard that is ONVIF. The existence of the ONVIF protocol for IP cameras is a major reason to pick an (ONVIF-compliant) IP-camera. By creating an architecture that uses an ONVIF-compliant IP camera, one is not locked into a certain vendor indefinitely, and if the camera needs to be replaced in the future, one could just acquire a newer ONVIF-compliant IP camera, even from a different manufacturer, with the best price/quality ratio at that moment of time in the future, and the architecture should still be functional. Thus ONVIF-compliance takes care of the replaceability issue and Section 3.1.2 explains how this requirement works in practice, by elaborating on the details. In conclusion, after comparing USB cameras and IP cameras, an ONVIF-compliant WiFi IP camera is most likely the best camera type for this project.

Furthermore, this choice makes the most sense, because generally one does not use an USB camera for surveillance, unlike an IP camera, although there are ways to convert one camera type into the other.

3.1.2. ONVIF PROVIDES FOR BETTER REPLACEABILITY

Open Network Video Interface Forum (ONVIF) was established to remove interoperability issues between digital surveillance solutions, at a time when there were many proprietary surveillance solutions available. As of October 2016 there were nearly 8400 ONVIF-compliant products available on the market, according to statistics shown on the ONVIF website [2].

The core of ONVIF consists of the ONVIF profiles. These ensure that compliant devices and clients are compatible. For example a device that conforms to profile S is compatible with a client that conforms to profile S. There are 6 profiles right now: S, C, G, Q, A are the first 5 profiles and the 6th one, T, is the newest proposed profile. In this ordering S is the oldest profile, with the most basic features. The profiles after S are newer, with more advanced features supported. So Profile S covers basic video streaming functionalities and with newer profiles there are additional functionalities, for example support for video storage in profile G. A device can support one or more profiles at the same time [3].

For this project the minimum functionality that is needed is the ability to retrieve a RTSP-stream from the camera using an ONVIF-command (GetStreamURI). This functionality is mandatory in devices compliant with profile S or T. [4]

3.1.3. OPTIMAL CAMERA CHOICE

The search for an ONVIF-compliant WiFi IP camera led to the result of acquiring the *Digoo DG-M1Q*. The specs describe a 1.3 MegaPixel 960P Onvif-Compliant WiFi IP camera for a low price, under €20, which is the best price/quality ratio one can acquire in this price range currently. Furthermore it fulfills other requirements of the customer, by supporting IR Night Vision up to 5m, with an IR distance of 10m, and it is a small camera, with a diameter smaller than 8 cm. Some other plusses listed are H.264 compression support, the operating temperature of -10 to 60 °C and the ability to store video on a SD card, in case of a network failure.

Thus this camera is the recommendation for this project, but not unexpectedly, for such a low price some compromises have been made. The expectation is that these compromises can be solved by buying a more expensive ONVIF-compliant WiFi IP camera or by buying a more advanced but still cost-efficient camera in the future. The first compromise is that the camera is black instead of white. The second one is that the camera does not describe humidity level boundaries or a waterproof criterion. This may not be a big deal however, as it is an indoor IP camera and it should thus operate satisfactorily at regular indoor conditions. The third compromise is that the camera does not appear to offer built-in encryption. As a mitigating factor, neither does the competition. Built-in encryption does not appear to be a priority for manufacturers. The final compromise is that the lack of a specific ONVIF version number in the specs, could indicate that the camera manufacturer may not have adhered properly to the ONVIF specification. However tests indicated that one can retrieve a RTSP videostream over UDP from the camera via an ONVIF-command, so fortunately it appears there is ONVIF-compliance with Profile S, as this functionality is supposed to be mandatory in that profile [3].

Taking into account all of the above, there is no guarantee the camera will perform to *all* the expectations in practice, especially considering the very low price. However, until new information comes in from real-life empirical data, expectations will be set to the theoretical specifications supplied by the manufacturer. So for now this camera appears to be the most optimal/cost-efficient choice.

3.2. STREAMING

Multiple protocols related to streaming exist. This chapter discusses different streaming technologies to answer the corresponding research question: What streaming technologies should be used?. Section 3.2.1 discusses the existing transport protocols for transferring data over the internet. The used transport protocol has an impact on the bandwidth usage in the hospital and on possible blocks by firewalls in routers, two important aspects of the system. Section 3.2.2 lists the different streaming methods. The choice of a particular streaming method is important for being able to play a live stream from the camera. Section 3.2.3 compares the two most important implementations of adaptive bitrate streaming. The main concern here is to use an implementation which works in browsers across different platforms. Section 3.2.4 compares RTSP and HTTP Live Streaming, two different streaming protocols. Finally, in Section 3.2.5 a recommendation is made for the streaming protocol which suits this project's needs the best.

3.2.1. TRANSPORT PROTOCOLS

To compare different options for streaming video, a basic understanding is needed of the two transport protocols that are used for transporting data over the internet: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).

TCP is connection-oriented and establishes a connection between sender and receiver, and data can be sent bidirectional. UDP is connectionless, which means no connection is established between server and client. TCP will ensure that the packets sent are received in the right order and that no packets are lost. With UDP these guarantees are not given, which makes TCP a more reliable transport protocol. Because UDP doesn't establish a connection, there is less overhead. This makes UDP faster and more lightweight. UDP can have troubles traversing the net, because firewalls in routers might block UDP. This is usually not a problem with TCP. Finally, with UDP it is possible to multicast, which means a stream can be sent out once for multiple people to watch. With TCP multicast is not possible because TCP requires a connection between client and server; this means that the stream has to be sent individually to every client that is connected and therefore costs more bandwidth and CPU power [5, 6]. However, multicast requires forwarding of the data in the router which is not done automatically. This means that settings in the router need to be changed manually, because often routers block multicast [7–9]. The aim is to build a system that works out of the box and is not dependent on external factors, which is why multicast cannot be used.

Protocol	**Connection**	**Reliability**	**Latency**	**Overhead**	**Firewalls**	**Multicast**
UDP	Connectionless	Low	Low	Low	Possibly blocked	Yes
TCP	Connection-oriented	High	High	High	Not blocked	No

Table 3.1: Comparison of the two transport protocols: UDP and TCP

3.2.2. STREAMING METHODS

There are three different options to deliver multimedia over the internet:

- Progressive download
- Streaming
- Adaptive streaming

Progressive download uses HTTP (which runs over TCP). The stream is downloaded progressively into a local buffer, and as soon as enough necessary data is retrieved and buffered the playout can start. Progressive download is not bitrate adaptive. This means the user must choose the most appropriate version (i.e. the quality he wants to watch the video in) before the download starts and this is fixed. If there isn't enough bandwidth for the selected quality, the user might experience frequent freezes and rebuffering. Progressive download doesn't support live streaming, which makes it unsuitable for this purpose [10, 11].

Traditional streaming sends the content as a continuous stream of packets over UDP or TCP transport. It requires a stateful protocol which establishes a session between the service provider and client. A widely used

implementation of this is the Real-Time Streaming Protocol (RTSP), which uses RTP (Real-Time Transport Protocol) as transport protocol [11]. RTP usually runs over UDP, and therefore supports multicast and has low overhead [10].

With adaptive streaming, the streaming client sends HTTP request messages to retrieve particular segments of the content from an HTTP server and then renders the media while the content is being transferred. These segments are short and are available at multiple bitrates (corresponding to different resolutions and quality levels), which enables the client to switch between different bitrates at every request. The client player aims to always retrieve the next best segment, given the constraints of the user (available bandwidth and CPU capacity). This means that the quality of the video is supposed to be always optimal for every user [10]. Since adaptive streaming is transported over TCP, it works based on the unicast delivery scheme, meaning the server has to send a packet as many times as the number of clients requesting that packet [10]. Solutions like content caching and Content Delivery Networks (CDN) can prevent the server from being overloaded and ensure efficient delivery to the client. CDNs are discussed in Section 3.4.1.

3.2.3. IMPLEMENTATIONS OF ADAPTIVE STREAMING

Multiple implementations exist for adaptive bitrate streaming, including MPEG Dynamic Adaptive Streaming over HTTP (DASH), Apple HTTP Live Streaming (HLS), Adobe HTTP Dynamic Streaming and Microsoft Smooth Streaming. HLS is the most used solution and DASH is the only solution which has been accepted as an international standard, which is why this section only focuses on these two options.

First, DASH aims to solve certain problems that exist with other adaptive streaming technologies (like HLS), and therefore has a number of improvements compared to HLS. It is beyond the scope of this report to discuss these improvements here, but a nice comparison of features for different implementations can be found here [12].

The most important criterion for the system is that the stream can be played in all major browsers. Both DASH and HLS can be played in almost all major browsers, with the exception being iOS for DASH. HLS is widely used, opposed to DASH, which means a lot more support exists for HLS. This makes HLS the best option at this moment.

It is important to note that DASH is the only implementation that has been accepted as an international standard and therefore it can be expected to become the most used solution in the future. With an eye on the future, it would be good if it is easy to transition from HLS to DASH. Recent developments might enable this. To discuss how, some more explanation is needed of how the adaptive streaming technologies work.

The basics of every adaptive streaming technology are the same. A compressed video stream is split up in segments of a few seconds and these segments are wrapped in a container. Each segment is saved multiple times with different qualities, which allows the client to quickly switch between different video qualities based on the available bandwidth. Each batch of segments has a manifest file associated with it, specifying characteristics of the segments [13, 14].

HLS normally uses MPEG-2 transport streams (MPEG-TS) as container for the segments, and DASH uses fragmented MP4 (fMP4) [13, 14]. This means that if you want to be able to serve both DASH and HLS (depending on the client) the stream would need to be transcoded (re-encoded) twice: once to MPEG-TS and once to fMP4. Recently HLS added support for fMP4. This means that the only difference between DASH and HLS is the content of the manifest file (which is a simple text file), and all that is needed to be able to stream either DASH or HLS are two manifest files, which does not require any significant additional storage space or transcoding.

Because DASH seems to be moving to become the industry standard, it is a good idea to already prepare for that by enabling the use of both DASH and HLS from the server, without significant additional processing or storage capacity needed. However, the Apple developer guide [15] states that live streams require MPEG-TS, but it is uncertain how up-to-date this guide is and whether this might change.

3.2.4. RTSP VERSUS HLS

Two different streaming protocols for adaptive bitrate streaming have been discussed in the previous sections. This section compares RTSP (widely used for live streaming) and HLS, and discusses their respective benefits and downfalls.

A big advantage of RTSP over HLS is that RTSP supports multicast. When a large number of people want to watch a stream, unicast places a much higher burden on the server and on the network, because each packet has to be duplicated for every client connected to the stream. In this use case there will not be many people watching one stream (a maximum of 10), so unicast connections will not be a very high burden for the

server. Multicast also requires us to control the network end-to-end, which is not the case in this project, so it is impossible for us to make use of. In the case that many viewers would need to watch one stream, solutions like content caching and CDNs ensure efficient delivery to the client without overloading the server.

With RTSP the latency is much lower and there is less overhead (meaning less bandwidth is used) than with HLS. In this project's use case latency is not an issue, so the stream can be delayed by 30-60 seconds. Bandwidth usage is an important consideration, so the fact that RTSP uses less bandwidth is a significant advantage.

There are two big problems with RTSP, which are solved by HLS. First, RTSP streams sent over UDP might be blocked by firewalls in routers. Therefore the network traversal is less reliable and router parameters possibly have to be changed. The system should work anywhere and should not require specialized knowledge, so it is undesirable to have to change settings in the router for the system to work, for instance. HLS is delivered with HTTP, which is never blocked by firewalls. So it offers a more reliable network traversal and the system works out of the box.

Second, there is no support for RTSP in browsers, meaning a RTSP stream cannot be played in a browser without installing a plugin. HLS streams on the other hand, can be played in all the major modern browsers, including mobile browsers (both Android and iOs) without requiring a plugin. This means that to display a stream in the browser it must be a HLS stream.

3.2.5. CONCLUSION

Initially, the streaming choice is dependent on the camera and the protocol it uses. For example, the IP Camera from Digoo described in Section 3.1.3 uses RTSP over UDP. This stream can be sent to an Edison on the same network for processing. From there it can be sent to a remote server, which stores the video and serves it to the client (see Section 3.5 about the architecture of the system). Based on the conducted research the best option is to send the stream to the remote server using RTSP over UDP, because it has the lowest latency and has the lowest overhead, therefore posing the least amount of load on the network to which the camera is connected. Issues might arise if the outgoing stream is blocked by the firewall of the router (which is controlled by the hospital). In that case, the stream can be sent with RTSP over TCP which is less likely to be blocked by firewalls. To serve the stream from the server to the client the best option is HTTP Live Streaming (HLS) because it can be played in all browsers and ensures proper network traversal. To prepare for the future, in which DASH can become the most supported solution, it is a good idea to setup the system in such a way that an easy switch to serving the stream over DASH can be made.

3.3. ENCRYPTION

When the video stream leaves the camera and is sent out to the internet, only the people for which the stream is intended should be able to watch it. It is important unauthorized people cannot intercept the stream and watch it, a so-called man-in-the-middle attack. To prevent this, the transport needs to be secured.

In Section 3.2 different streaming protocols are discussed. Because it concludes that the two different protocols RTSP and HLS are used throughout the system, two different methods for secure transport are needed. The stream is first sent to the server using RTSP. Before it is sent to the server, it can pass through a device (for example the Intel Edison) where the stream can be secured. RTSP uses RTP as transport protocol, and RTP has a secure variant as well: Secure RTP (SRTP). SRTP ensures the confidentiality of the RTP payloads and the integrity of the RTP packets [16]. Open source software exists to convert an RTP packet to SRTP. Therefore SRTP can be used to ensure safe and secure transport of the stream from the camera to the server.

The stream is served from the server to the client using HTTP Live Streaming (HLS) which uses HTTP as a means of transport. Transport Layer Security (TLS) provides communications security over the internet, and allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. It can be layered on top of a reliable transport protocol like TCP [17]. HTTP is sent over TCP. HTTPS is the secure variant of HTTP, which sends data over TCP which is secured with TLS [18]. So to secure transport from the server to the client, a HTTPS connection is established.

For additional security it is possible with HLS to encrypt the media segments [13]. This means that even if an attacker would intercept the video of a stream (which is unlikely), he would not be able to watch it. Open source software exists to apply this encryption.

3.4. STORING VIDEO

This section discusses the best way to store videostreams by dividing the answer to the research question 'How should the videostream be stored?' into three parts: *where* to store in Section 3.4.1, the method *by which* to store in Section 3.4.2 and *how* to store the video in Section 3.4.3. It concludes with some ideas about scalability in Section 3.4.4.

3.4.1. STORAGE TYPE SOLUTIONS COMPARED

When deciding where to store the video an important distinction is made between personally managed storage media and the cloud, where you rent space from an entity (e.g. Amazon CloudFront) to store the video at a certain cost. For privacy reasons the customer wants to maintain control over the data as much as possible, so using a personally managed server seems like the better option to store the video.

A server is a multipurpose device, always connected to the internet, that can provide different services to multiple clients over the web, often at the same time. A server can also run other services, like authentication, so that the content can be securely distributed to multiple clients. The customer already set up different applications to work on a server running Ubuntu 14.04, so the preference is to use this same server for storage of video data.

A custom server also has high customizability, so it lends itself to easily introducing new features like Motion Detection on the video data. Therefore, the video data will be stored on a server running Ubuntu 14.04.

3.4.2. WHAT METHOD CAN BE USED BY WHICH TO STORE THE VIDEO DATA?

The method by which to store the video is a Network Video Recorder (NVR). A Network Video Recorder is a software program, often sold in a combined package with a dedicated storage device, that is specialized in storing video footage from IP cameras over a local network (LAN). Sometimes the NVR is on a dedicated storage device and this combined package is sometimes also referred to as a NVR. In that case some additional work must be done to be able to serve the content securely to multiple clients over the web, for example connecting a webserver to the dedicated device.

In this case a custom server is used, so webserver and NVR functionality can both be present on a single multipurpose server. There is a possibility of using commercial proprietary software like Wowza [19] to do this, which turns the server into a rudimentary form of a "streaming server", but the preference is to configure open-source software, like FFmpeg, to create custom NVR functionality. This reduces costs, allows for better control over confidential data and eases introduction of new features like Motion Detection on the video.

3.4.3. HOW TO STORE THE VIDEO DATA?

There are generally two different ways of storing the video data on a storage device:

- In a database (DB)
- In the filesystem

Searching around on the internet it appears video is virtually always stored in the filesystem for efficiency reasons. That makes logical sense as to store a large file like video in a DB, a videofile must first be converted into a BLOB (Binary Large Object), or vice versa, to retrieve the video for watching, unnecessarily wasting processing power. An open-source DB that can be used is MariaDB, that can store large files in sizes up to 4 GB as a LONGBLOB [20]. Unfortunately MariaDB appears to lack a specialised datatype for efficiently storing video in the DB, that a proprietary solution like SQL Server does have. The SQL Server FILESTREAM type combines the performance of the filesystem with the transactional consistency and easier administration of the database to create an optimal solution to manage video in a database [21]. To eliminate the conversion step to/from BLOBs associated with storing videos in MariaDB one can store the videos itself in the filesystem of the storage and if there is associated metadata, that itself can be still be stored in a lightweight DB. The filesystem will be optimized for storing and retrieving files efficiently and the DB can then contain pointers to the actual paths of the file locations in the filesystem. The choice for storing in the filesystem is supported by [22] which concludes after empirical experiments that for storing objects under 256 KB, a DB has a clear performance advantage, for objects larger than 1 MB on average, the filesystem has the performance advantage. Therefore this project will store video files in the filesystem and associated patient data on these videos in a MariaDB instance, as this will make the video storage most efficient.

Furthermore an approach must be devised as to how exactly the video is stored in the filesystem. The requirements list the ability to watch saved video footage recorded recently at any moment in time. An example would be video up to 1-hour ago. However, the camera will be continuously streaming, so some method must be devised to throw away old data that is not of the past hour, so that storage limits are not exceeded.

One approach, is to store the videostream in segments, for example 15 minute segments, and always serve the 4 most recent segments to the viewer by merging and serving the merged file or by serving a playlist of the different segments. All older segments can be discarded. Segmenting however, may introduce some loss of data when the instance of ffmpeg that is receiving data is temporarily not able to receive data because it has to switch over to a new file to record data. Furthermore this approach may be harder to program and more error prone due to being forced to work with multiple files. Also, if using a playlist is not possible, processing power must be expended on merging the files for serving.

A second approach which possibly eliminates these disadvantages is storing the video in one file, of for example a maximum of 12 hours, and when content is requested, split off the last hour and serve this content. This approach introduces a different disadvantage though, after a certain point the file contains a too large amount of data, a lot of which is older than 1-hour and must somehow be discarded. So if cutting away that old data is not easily possible, some kind of segmentation must be introduced, which allows for recording of the next 12 hours. In practice that means that both 12-hour segments must somehow have a 1 hour overlap, so that the customer requirements stay fulfilled.

The first approach is preferred, because it uses smaller video segments. When a video file is then corrupted, only a small video segment is lost as opposed to when using a 12 hour video.

3.4.4. HOW TO SCALE THE SYSTEM?

The project should result in a scalable system, that can scale with the number of cameras or be applicable to use cases in other countries. By storing the video on a NAS server, upgrading the server with more storage capacity and processing power should be sufficient when scaling the number of cameras. However, there may come a time, when upgrading the server won't do anymore. In that case some other solutons must be devised.

One first solution is to copy the server setup to other servers. This should take care of most scalability issues pertaining the server. However, there are edge cases where a different solution may be desired, such as in the case of a client wanting to access video data from a different country. No matter how good the server is, user experience may be poor due to having too much travel time for the data.

A different second solution to fix this is to scale the system by offloading server tasks to a Content Delivery Network (CDN), although unfortunately, this does rely on making some compromises. A CDN is a large distributed network of servers that delivers content to clients efficiently, by utilizing a variety of techniques, and gets paid for that by the content provider [23]. An example is caching static content and delivering this content from the nearest server, where a CDN URL replaces the original URL that served content from the original server. It is suitable for efficiently delivering content to thousands of users, especially when an efficient global reach is desired, by having servers in many countries. Using a CDN, however, may require us to hand a limited set of confidential videodata to the CDN for efficient redistribution. Theoretically the least handing over of confidential data can be done by having a DB with confidential metadata point to CDN URL's instead of to filesystem paths, while also sending over video footage to the CDN URL. Footage from unidentified sleeping babies that is served via the CDN servers should be quite uninteresting without the associated metadata in the DB indicating identity. The upside though is that serving videodata to more clients will be easier and serving clients in other countries will be much more reliable.

When considering scaling via CDNs it is important to make a rough distinction between the type of CDNs that is relevant to the project, so that upscaling is done via the right type of CDN. The first type, which is general purpose CDN, which is commonly known as just CDN, will be the least interesting to the project. The second type, which is On-Demand Video CDN, is more interesting. Streaming servers deliver only the bits requested that will actually be watched due to adaptive bitrate (ABR), so you only pay for actual consumption of data, in contrast to the situation of downloading video. [23]. The third type is Live Video CDN, which is also interesting for the project. This is arguably the least mature of all CDN models, as the inability to cache live video increases costs and basic CDN infrastructure must be modified and invested in months ahead of time to be able to meet peak demand for live video. [23].

Furthermore, when picking which type of CDN to upscale with, it is also important to keep in mind the distinction in services that the project provides, which is a mix of VOD and livestreaming. Because of this mix, just one CDN may not satisfy all the upscaling needs. Depending on which of these services satisfies

clients the most for the least amounts of costs, one can decide to upscale only the most desired service based on market research and empirical experience with the product. Another option is to use multiple CDNs at the same time. All-in-all, scalability appears to be quite a complex topic as there are many ways to mix the possible options and it looks like quite an extensive area of future research.

3.5. SYSTEM ARCHITECTURE

This chapter answers the research question concerning the system architecture: What should the architecture of the entire system look like?. It gives a general overview of the entire system that is involved in streaming video from a camera to a browser, giving special consideration to the constraints mentioned in the research question. Figure 3.1 shows a schematic overview of the complete system architecture, which is elaborated on in Section 3.5.1. Section 3.5.2 discusses why this architecture is most efficient given the constraints.

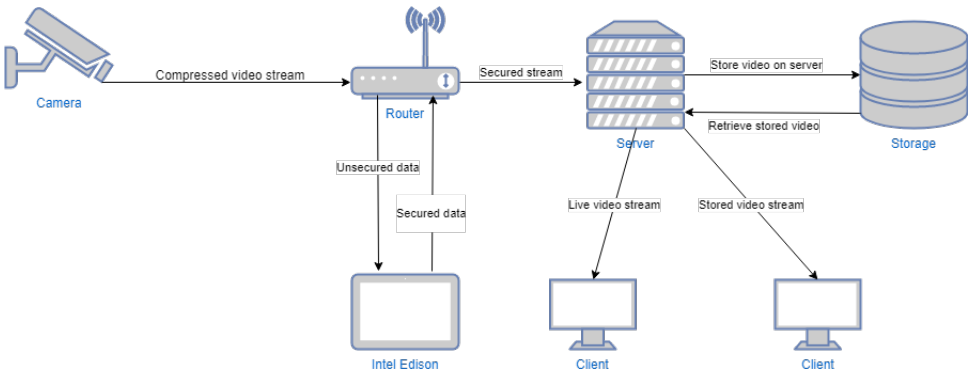

Figure 3.1: Schematic diagram of system architecture

3.5.1. SYSTEM ARCHITECTURE IN DETAIL

Because WiFi IP cameras are considered to be the best option, as is discussed in Section 3.1, the stream from the camera is being broadcast on the local network through the router.

Because the customer already uses an Edison and an Ubuntu server for other applications, it makes sense to use these for streaming as well, since that prevents the customer from having to spend money on additional hardware. So in the system uses one Edison per camera and an Ubuntu server.

One of the constraints given by the client is that as little processing as possible should be done on the Edison. If possible, the Edison is only used to secure the stream that it receives, after which it sends it to a remote server. As is discussed in Section 3.2, the stream of the camera might be blocked by firewalls on the router and might not reach its intended destination. In this case the Edison is needed to transform the stream to a format that is not blocked by the router's firewall.

From the hospital the secured stream is sent to the remote Ubuntu server. The server decrypts the stream and then stores a copy of it in its file system (see 3.4). When a client requests the livestream of a camera from the server, the server transforms the incoming stream to an appropriate format so it can be played in the browser (refer to Section 3.2.3). It also encrypts the stream to ensure secure transport (see 3.3). The livestream is then sent to the client.

The client can also request a recorded stream. In this case, the server retrieves the right stream from the storage, after which the same process as is described for the livestream is set in motion.

The entire sequence of events is depiced in Figure 3.1.

3.5.2. REASONS FOR THIS ARCHITECTURE

The system architecture described above is an efficient solution for displaying the stream of a camera in a browser, given the constraints and the available resources.

As little processing as possible is done on the Edison. The Edison ideally only performs encryption of the video stream. The key factor in keeping the bandwidth usage as low as possible lies in keeping the size of the stream that is sent as low as possible.

The architecture is secure: The Edison provides encryption of the livestream. And before the server sends the stream to the client it encrypts it again. Therefore, the described architecture ensures that the stream is always secured when sent over the internet.

The architecture should be easily scalable: On the server scaling is easy, when the server cannot handle the load anymore, more processing power and/or storage can be added. In case of large upscaling, the system can be modified to use CDNs (see Section 3.4.1). In the hospital the customer provides one Edison for every camera, so this automatically scales. An issue can arise with the bandwidth availability in the hospital. If there are a lot of cameras in one hospital, the router might get overloaded. The solution in this system would be to tell the hospital to increase their available bandwidth.

The system could be expanded to decrease the bandwidth usage in the hospital. Because monitoring the babies will often be done by nurses who work in the hospital (and are connected to the hospital network), the router will also need bandwidth to fetch the streams from the server and serve it to the nurses. Because the nurses are on the same LAN as the cameras and Edisons, they can also receive the stream directly. So an extra interface could be made which retrieves the stream from a camera that is on the same network. Processing of the stream needs to be done before it can be displayed in a browser. To make this possible, either the Edison needs to do more processing of a stream, or a server needs to be placed in the hospital which does the processing.

4

SOFTWARE

To implement the system described in section 3.5 external software is needed. Because the aim was to make the system without any costs, only FOSS is used. Section 4.1 discusses the streaming software that is used. Section 4.2 describes the software used for encryption. Section 4.3 elaborates on the software used to make a streaming server. And finally, section 4.4 discusses the open source libraries used to create a multi-platform video player.

4.1. STREAMING SOFTWARE

To work with video streams, streaming software is needed both on the Edison and on the server. This software needs to be able to transcode the stream, change the format of the stream (e.g. from RTSP to HLS), and change the transport protocol of the stream.

To do this, FFmpeg is used as a streaming library. FFmpeg is an open source library that supports multiple transmission protocols, media container formats, and video and audio coding standards [24]. It also contains efficient codec algorithms that meet the requirements of real-time video / audio streaming [25]. The wide variety of options provides a lot of flexibility, and the efficient algorithms ensure that the framework is efficient enough to handle live streaming.

It is highly customizable, and well documented. It is widely used so there is a lot of support for it. Finally, the server software used depends on FFmpeg (see section 5.3.1), so for uniformity it is the best option.

4.2. ENCRYPTION SOFTWARE

To prevent man-in-the-middle attacks, the stream that is sent over the internet needs to be secured. The stream is served to the client (i.e. the browser) using HLS, which uses HTTP as a means of transport. As was discussed in section 3.3, this can be secured by sending the stream over HTTPS. The software used on the server to serve the stream over HLS is a web server at its core (Nginx, discussed in next section). This means it has built-in support for setting up HTTPS connections and this can be used to securely send the stream to the browser.

The stream that is sent from the Edison to the server also needs to be secured, and this stream is not sent over HTTP. Therefore, a different protocol for encryption is needed. The server software used does not have support for accepting and processing encrypted streams. It is also not clear from the documentation of FFmpeg whether it has support for encrypting streams. So intermediary software is needed on the Edison to encrypt the stream before sending it to the server and software on the server to decrypt the incoming stream before passing it through to the software that handles the stream.

The software used to handle encryption and decryption of the stream is Stunnel. "The Stunnel program is designed to work as TLS encryption wrapper between remote clients and local (inetd-startable) or remote servers" [26]. This is exactly what is needed for this use case. The way it works is that both the Edison and the server have an instance of Stunnel running. The server creates a self-signed certificate and a private key. This certificate is distributed to trusted clients (i.e. all Edisons). Stunnel on the Edison uses this certificate to encrypt the data. Stunnel on the server receives the encrypted data and decrypts it using the private key. Because the server is the only entity that knows the private key, only the server can decrypt the data that is encrypted with the certificate.

4.3. SERVER SOFTWARE

Software is needed to turn the server into a streaming server. The requirements are that it can accept incoming streams, record them, and serve them to the client using HLS. Because HLS is server over HTTP, a web server with some additional functionality is sufficient. The advantage of using a web server to handle streaming is that it can also be used for serving the web content (i.e. the website/app itself), which means everything can be handled with the same software. Also, a web server has built-in support for authentication and secure connections.

Widely used free web servers are Apache and Nginx. Apache appears to lack a streaming module and is therefore not a viable option. Nginx has seen a steady rise in popularity in recent times and is the third most popular web server in usage [27]. Because of this popularity, open source modules have been built to turn Nginx into a streaming server (more information about the module in section 5.3.1). These modules combined with a FastCGI server and a Node.js server written ourselves, offer all the functionality needed.

Different options exist as well, that offer out-of-the-box solutions which do not require any additional code to be written. These include Wowza Streaming Engine and Nginx Plus (a paid version of Nginx). However, these and other fully implemented streaming software solutions are paid, opposed to the open source version of Nginx. Because the customer wants a free solution, these are not viable options for us. Besides, a disadvantage of these out-of-the-box solutions is that less customization is possible. Therefore, Nginx is used as software for the streaming server.

4.4. VIDEO PLAYER

To play the video served over HLS in the browser, a video player is needed. A widely used open source video player for HLS video is hls.js [28]. This is a basic video player which lets the developer customize everything. This offers a lot of freedom to the developer, but it also means more work needs to be done to create a reliable video player. Because this project is a proof-of-concept and because it is easy to switch to a different video player at a later stage, this option is not used because of the additional work it required to make the player work reliably.

Instead, video.js HLS source handler [29] is used. This video player is built on top of hls.js, which means that it took a lot of work out of our hands by making a more reliable video player which can be easily implemented. It is actively developed and has a lot of contributors, so it seems like a good choice.

5

IMPLEMENTATION

The project consists of many separate parts working together. As a consequence a lot of different programming languages and open source software is used, such as C++, JavaScript, Bash, Node.js, Nginx, FastCGI, and FFmpeg. The project is set up in a modular way. Each module is described in this chapter, together with giving a complete overview of the system. Section 5.1 discusses the development environment that was used. Section 5.2 explains how the stream from the camera is sent to the server. Section 5.3 describes how the server is set up to act as a streaming server. Section 5.4 contains a description of the video player used to display the video stream. And finally, section 5.5 gives an overview of the entire system.

5.1. DEVELOPMENT ENVIRONMENT

Because the project consisted of multiple parts, multiple development environments were used as well. The server that will be used by the customer will be running Ubuntu 14.04, so this is the development environment that was used to develop software for the server. A Virtual Machine running this Ubuntu version was provided by the customer. However, developing in a Virtual Machine is not very convenient, which is why a fresh install of Ubuntu was used for development.

Streaming video to the server will be handled by an Intel Edison, which is an embedded device. Because this embedded device also runs Linux, development for it was done on Ubuntu as well. The developed applications were then ported to the Edison.

Finally, an implementation of a video player in the browser was needed. Development of this video player can be done on any platform. For convenience it was done on Ubuntu as well.

5.2. STREAMING TO THE SERVER

Based on research and requirements of the client, a system is implemented where an IP camera is used in combination with an Intel Edison to stream video to the server. To be able to do this, the Edison needs to find the camera it wants to stream (5.2.1). When it has found the camera, a streaming library can be used to transcode the video and transmit a stream to the server (5.2.2). To ensure a secure connection the stream is passed through a proxy which encrypts the data (5.2.3). To combine the discovery application and the streaming library, a bash script is used (5.2.4).

5.2.1. DISCOVERY

One requirement for the IP cameras is that they are ONVIF compliant. This ensures that they can be found on the local network by any client application that implements the ONVIF protocol. An application is written in C++ to find all the ONVIF compliant IP cameras on a local network.

ONVIF uses a SOAP message exchange protocol [30]. gSoap [31] is used to make SOAP calls from within the application. gSoap is a library which attempts to make it easier to make SOAP calls from C/C++. It creates a number of files, which need to be included in the project, based on the API of ONVIF. ONVIF uses Web Services Dynamic Discovery (WS-discovery) to find devices on a local network. WS-discovery defines a discovery protocol to locate services, and was accepted as a standard by Oasis in 2009 [32].

The application uses WS-discovery to find all ONVIF compliant IP cameras on the local network. It registers the Universally Unique Identifier (UUID) of each camera it finds to the customers database on the server.

The existing server has an API for this which is called with a HTTP request, passing the UUID as a parameter. If the camera is already present in the database, it will not be registered again. The registration ensures that any camera that is used for streaming is registered in the database. When a camera is registered, a hash of the UUID is also created and stored with it. This hash is used instead of the UUID itself in the name of the stream that is transmitted. This is for security reasons.

The application is run with settings containing the UUID of the camera. This is to indicate that the camera with this UUID should be used for streaming. This is necessary because in a hospital environment multiple cameras will be on the same network, so an Edison needs to know which camera to stream. This is done by specifiying the UUID of a camera on each Edison and passing these settings to the discovery application. When the application has found the camera with the matching UUID, it retrieves the camera hash for this camera from the server. This hash is written to a file, together with the RTSP URI(s) of the camera. These RTSP URI(s) are used as input by the streaming library to retrieve the live stream from the camera.

5.2.2. FFMPEG

To stream the video from the camera to the server, FFmpeg is used as a streaming library. To use FFmpeg on the Edison, it needs to be compiled for the operating system running on the Edison. When FFmpeg is installed, a single command is used for streaming. This command takes the RTSP URI of the IP camera as input and outputs the stream in RTMP format.

The reason for outputting the stream in RTMP format is twofold. The first reason is that RTMP is sent over TCP by default, as opposed to RTSP which is often sent over UDP. As is discussed in Section 3.2.1, TCP is preferred over UDP. The second reason is that the streaming server that is used for receiving incoming streams and transmitting them as HLS, only accepts streams in RTMP format. The server is discussed in more detail in section 5.3.

Because the Edison is an embedded device, its processing power is limited. Because streaming is a CPU-intensive task, a concern was whether the Edison would be able to process the stream fast enough for live streaming to work smoothly. To accomplish this, the resolution and quality of the stream are downgraded and optimizations have been added to ensure that the Edison can handle the stream. With these settings, the Edison uses up to a maximum of 50% of its total CPU capacity for streaming. The resulting resolution of the stream is 320x240, instead of the default 1280x960.

Finally, it is important that the stream is sent over a secure connection to prevent man-in-the-middle attacks. RTMP has a number of secure variations on its protocol, but these are not supported by the streaming server used, so they cannot be used. Instead, the stream is sent through an SSL/TLS proxy, which encrypts the stream and then sends it to the server. See the next section (5.2.3) for more details.

5.2.3. STUNNEL ON EDISON

Stunnel is used to encrypt the stream before sending it to the server. In this case, Stunnel listens on port 2121 on localhost on the Edison. Stunnel encrypts any data that comes in on this port using a certificate that is provided by the server and then sends it to the server. The server also runs an instance of Stunnel, which decrypts the incoming data using a private key and then passes the decrypted data through to the actual streaming server. This setup ensures that the data that is sent over the internet is encrypted.

5.2.4. PUTTING IT ALL TOGETHER

In the previous sections, the different components used for securely streaming to the server are described. To make these components work together and create a working system, a bash script is used which runs as a service on the Edison. Because the system is dependent on Stunnel for encryption, Stunnel is also running as a service on the Edison.

This bash script first runs the discovery application to find the camera on the network. If it cannot find the camera, it will wait a few seconds and then try discovery again. It will do this over and over until the camera is found or the service is stopped manually.

When the cameras are found with discovery, the bash script calls the FFmpeg library to transcode the video and send it to Stunnel. It uses the RTSP URI(s) found by the discovery application as input, and it uses the camera hash found by discovery in the output name.

An IP camera usually has multiple RTSP URI(s), for example with different quality levels or one with sound and one without sound. When the first URI fails, the bash script tries to use the second URI as input for FFmpeg. If this also fails, the script starts over again from the start (i.e., with discovery). The same happens when streaming fails after a while for whatever reason.

5.3. SERVER

To store the video from the IP camera and to broadcast it live to clients, software is needed on the server that can accept streams and manipulate them (i.e. converting it to HLS). This section describes how the streaming server is implemented. Software is needed on the server to turn it into a streaming server (5.3.1). One task of the server is to accept and process incoming streams (5.3.2). It also needs to serve VOD to the client dynamically (5.3.3). To prevent anyone from being able to watch the stream, the possibility for authentication is needed (5.3.4).

5.3.1. NGINX

Nginx is used to build a streaming server. According to its own website, "Nginx is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more" [33]. To turn Nginx into a streaming server, additional open source modules are needed. The following modules are used:

- nginx-rtmp-module [34]: this module is necessary for live streaming with Nginx. It allows the server to accept incoming streams in RTMP format (this is the only format that is accepted) and offers the possibility to store the incoming stream and to publish a live stream in HLS format.
- nginx-vod-module [35]: this module is necessary for offering Video on Demand (VOD) dynamically. With this module it is possible to create a HLS video of one or more stored videos (the stored videos must be in MP4 format) and serve this to the client.
- nginx-secure-token-module [36]: this module is necessary in combination with nginx-akamai-token-validate-module to secure the stream that is viewed by the client.
- nginx-akamai-token-validate-module [37]: this module is used to validate client requests for videos.

5.3.2. INCOMING STREAM

The server needs to have the ability to accept and process incoming streams, create a livestream in HLS format, and store the incoming stream. To do all this, the nginx-rtmp module is used. This module allows us to send the stream from the Edison to the server in RTMP format (this is the only format the module accepts). But before the server can work with the incoming video data, it needs to be decrypted.

DECRYPTION

As was discussed in section 5.2.3, the stream that is sent to the server is encrypted using Stunnel. To be able process the stream, the server first needs to decrypt the stream. This is done by running an instance of Stunnel on the server as well. The stream is sent from the Edison to this instance of Stunnel, which decrypts the data using the private key which only the server knows. After decryption of the data, it is passed through to the nginx-rtmp-module.

NGINX-RTMP-MODULE

To prevent anyone who knows the server IP to be able to send its stream to the server, the incoming stream needs to have a private key in its URL which is only known by the server and by trusted entities (i.e. the Edisons). The server checks the key to see if it is correct. If it is not correct, the stream is not accepted. If it is correct, the stream is accepted and processed.

Using the nginx-rtmp-module, the incoming stream is converted to HLS continuously. This converted stream is stored in the file system of the server and is continuously updated with the last video data. This means that old video gets replaced by newer video. This video can be accessed by the client to watch a live stream of the IP camera in HLS format.

The nginx-rtmp-module also records the video in intervals of 10 minutes and stores these videos in the file system. The module only has the possibility to store video in FLV format, and for the nginx-vod-module the stored video must be in MP4 format. The workaround for this is to run a bash script after the recording of a 10 minute video is done (the nginx-rtmp-module offers the possibility to do this). This script converts the recorded FLV file to MP4 using FFmpeg.

VIEWING LIVE STREAM

The live stream in HLS format that is created by the nginx-rtmp-module can be accessed by a simple HTTP request, because of the nature of HLS. Nginx has a section which points to the directory where the live stream is created and then serves the video with the correct name to the client.

5.3.3. VIDEO ON DEMAND

To offer the possibility to watch Video on Demand in HLS format, the stored videos need to be converted to HLS and served to the client on request. To do this the nginx-vod-module is used.

NGINX-VOD-MODULE

The nginx-vod-module creates an HLS video from MP4 videos on-the-fly. It uses a JSON file as input to know which MP4 videos to combine in one HLS video. Additionally it is possible to specify exactly what part of the video to use (e.g. create an HLS video from *example.mp4* from minute 2 until minute 8) with this JSON file. So with this module it is possible to request videos per minute (e.g. watch video from yesterday 18:07 until 18:33). To do this, the JSON file that is used by the nginx-vod-module needs to be generated dynamically based on the HTTP request of the client. This is done by putting a UNIX timestamp in the request URL and passing this request through to a FastCGI server.

FCGI SERVER

The FastCGI server that is used to create a dynamic JSON file is an application written in C++. Every request for VOD is passed through to this application, and based on the parameters in the request URL (i.e. the UNIX timestamp and a string specifying the camera from which to select videos) it creates a JSON file which is passed back to the nginx-vod-module. The module then uses this JSON file to generate an HLS video with the right videos at the right time. The created HLS video is returned to the user.

5.3.4. AUTHENTICATION

To add security to both the live stream and the VOD stream, an Akamai token is added to the request URL as a parameter. Without a correct token in the request URL the server will not accept a request. The nginx-akamai-token-validate-module and nginx-secure-token-module are used together to handle the validation of the token.

This means that the client needs to create an Akamai token and add it to the request URL. To create such a token a Node.js service is written, which uses the akamai-auth-token npm package to generate an Akamai token. To retrieve the token from the client (i.e. the browser), the client can make a request to Nginx (which acts as a reverse proxy here) which passes the request through to the Node.js service. This service then generates a token and returns it to the client, which can append it to its request URL.

A limitation of this system is that currently no authentication is needed to create the token. This means that anyone can call the server to generate a token, which makes it useless. However, since this project will be integrated into an existing web application which already handles authentication, it should be easy to add authentication somewhere along the chain. The authentication can be handled either with Nginx (before passing the request to the Node.js service) or in the Node.js service itself (i.e. without proper authentication it will not create a token).

5.4. VIDEO PLAYER

The customer did not require full integration of a video player in his multi-platform frontend, however the Live and VOD video should be playable from inside a HTML5 videotag (i.e. <video></video>), so that integration with his existing set-up would be easy to do. For the proof of concept a simple video player (EyaVideo-Player) is built using video.js HLS source handler [29].

It has functionality to watch both the live stream or VOD. To watch the live stream it retrieves a token (see section 5.3.4 and then retrieves the live stream from the server. Because this player is not integrated into the existing frontend, a camera identifier is hardcoded in the JavaScript code.

To watch VOD a datetimepicker is built in to the frontend. This picker can be used to select video per minute from the last 10 days. When a time and date has been selected, a HTTP request is made to the server with the selected date and time as UNIX timestamp. The server creates a video of maximum one hour from the selected time.

5.5. SUMMARY

Figure 5.1 contains a schematic overview of the whole system. First, the IP camera that needs to be streamed by an Edison is searched for on the network through discovery. When the camera is found, its RTSP URL is used as input for FFmpeg, which transcodes the streams and transmits the stream over RTMP. The stream is sent to Stunnel, where it is encrypted before it is sent to the server.

The stream is sent to a Stunnel proxy on the server, which decrypts the stream. After decryption, Stunnel passes the RTMP stream through to the nginx-rtmp-module. This module stores the files in MP4 format and it creates an HLS playlist (which is continuously updated) which can be accessed for livestreaming.

When the client wants to watch a stream, he first needs to acquire a valid token. This is done by calling the getToken location on the Nginx server. Here, Nginx acts as a reverse proxy, passing the request through to a Node application which creates a valid token and sends the response back to the client. After having acquired a valid token, the client is able to watch VOD or a livestream. When watching a livestream, the client connects with port 8080 on the server. The server then grabs the video in */mnt/hls* and serves this to the client. When watching VOD, the client connects to port 8081 on the server. The server passes the request through to a FastCGI server, which passes a JSON file back to the server. Based on this JSON file, the server grabs the correct videos from */video_recordings/* and serves these to the client.

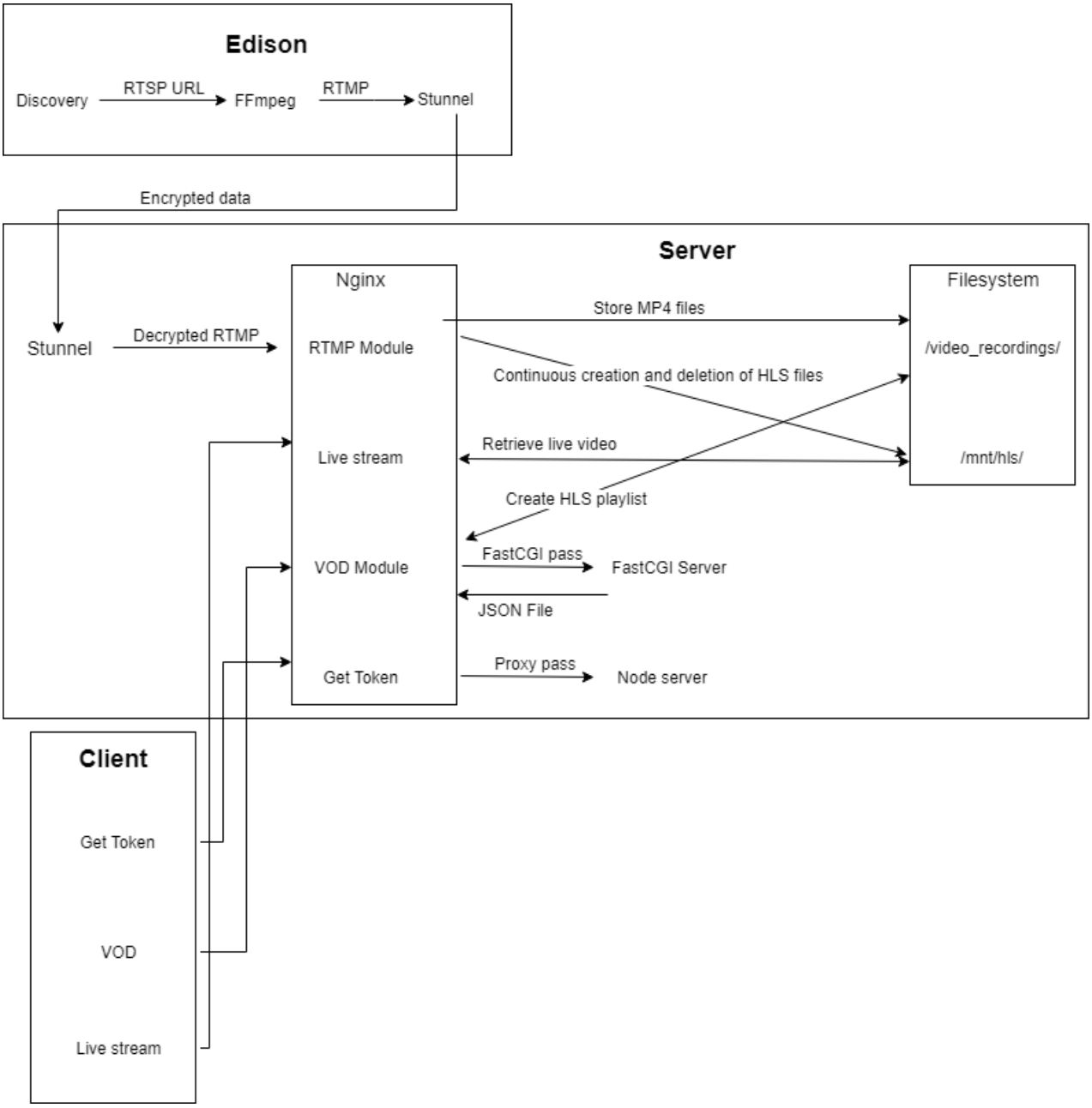

Figure 5.1: Schematic diagram of detailed system architecture

6

SOFTWARE QUALITY AND TESTING

An important aspect of software development is developing software that is easy to maintain (or change), robust, and does what it is supposed to do. To achieve this goal, code needs to be well-written, object-oriented where possible, and well documented. Code should also be tested with unit tests. Unit tests cannot cover everything so manual testing is used to test the functioning of the entire system working together.

This chapter describes the methods used to test the system. Section 6.1 elaborates on the various means used to test the system. Section 6.2 discusses the feedback received from the Software Improvement Group (SIG) regarding the source code.

6.1. TESTING

In the system both unit testing and manual testing are used to see if everything works as intended. Because a lot of external software is used, manual testing is needed because this cannot be automatically tested. Section 6.1.1 describes the unit testing frameworks used for the various applications. Section 6.1.2 discusses how the system is tested.

6.1.1. UNIT TESTING

There are two applications written in C++ in the system: the discovery application running on the Edison and the FastCGI application running on the server. To test the proper functioning of these applications, unit tests are used. the unit testing framework Catch [38] is used to do this. Catch is used because it is easy to set up and easy to use. It proved to be a useful framework, because numerous bugs where found when writing tests.

For the video player developed for the browser is tested using Jasmine [39]. Because the video player implementation is straightforward, testing seemed a bit of an overkill in this case with no added value.

The system also uses bash scripts on several locations. It is possible to unit test these scripts, but because the scripts used are straightforward not tests are written for them because their functioning correctly can easily be verified manually. The Node.js service used to generate a token is also only tested manually, because it only contains a few lines.

6.1.2. MANUAL TESTING

To test if all the external software that is used set up correctly and is working together correctly with the applications that are developed, manual tests are performed:

- To check if the server is accepting RTMP stream, a video is transcoded to RTMP format (using FFmpeg) with a simple command and if FFmpeg can establish a connection it means the server is accepting RTMP.
- To check if the server is publishing the RTMP stream as a live HLS stream, the video is played in a HLS video player. If this goes well, it means the server is publishing the live HLS stream correctly. Some examples of HLS video players are FFplay, Mplayer, VLC media player, Kaltura HLS tester and hlstester.com.

- To test if the Edison is streaming the video from the IP camera to the server, the logs of the bash scripts on the Edison are checked to see if the Edison has established a connection. Also, the livestream is played in a HLS video player.
- To test if Stunnel is encrypting and decrypting properly, data is sent from Stunnel on the Edison directly to the streaming server (instead of sending it to the instance of Stunnel on the server). The streaming server cannot process this data because it is encrypted, meaning Stunnel properly encrypts the stream.
- Video is streamed from the IP camera to the server using the Edison, and the server publishes a livestream in HLS format. The URL corresponding to the livestream is then used in the EyaVideoPlayer, to test if it is properly playing.
- To test the delay of the livestream the camera is pointed at the clock at `https://time.is/` and then the time on the video is compare with the actual time at this website. The difference is the delay in seconds.
- To test if the server is recording videos, the video is streamed to the server and then the file system is checked to see if the videos are recorded.
- To determine if the VOD functionality is working correctly, a few hours of video are recorded with the camera pointing at a clock. Video is then selected at different times to see if it returns the video at the correct time (by comparing the selected time with the time on the video, which should be equal).
- To determine if livestreaming and VOD works over the internet, the Edison and camera are put on a different local network from the server, and the server on a different local network from the client that is using EyaVideoPlayer.

6.2. SOFTWARE IMPROVEMENT GROUP

During the project, the source code from the two C++ applications and the EyaVideoPlayer has been uploaded to the Software Improvement Group (SIG) twice to receive feedback on the quality of the code. SIG provided the following feedback for the first upload:

> [Analysis]
> Your project scores 4 stars in our maintainability model, which means your code's maintainability is above average. The main areas for further improvement are Duplication and Unit Size.
> For Duplication we look at literal duplication within the same file or between files, i.e. copy/pasted blocks of code. In your project, we see some overlap in the parameter lists in DiscoveryEvent.h. This file defines multiple function signatures with long and similar parameter lists (wsdd_event_ ResolveMatches and wsdd_event_Bye), which is considered a code smell. Try to create a class (if C++) or struct (if you're limited to plain C) that encapsulates the parameters that are common to all event types.
> For Unit Size, we look at the length of functions. In general, the longer a function is, the harder it will be to maintain in the long term. In your case we recommend to look at Camera::getRtspUris, which consists of several different steps in a process. You can separate these steps out to new functions, which you can then call from the main function. This also makes it easier to unit test the steps separately.
> Speaking of which, well done for writing unit tests for C/C++ code. We are used to seeing lots of unit tests for high level languages, but we didn't expect it considering your technology choice.

To improve the code the given feedback was implemented where possible, and the same principles were applied to code that was added after the first upload. The feedback regarding the function signatures with long and similar parameter lists could not be implemented however, due to the fact that these functions were required to be declared in this way by a library that was used.

For the second upload, containing the final codebase, SIG provided the following feedback:

> [Feedback second upload]
> In the second upload, we can see a small increase in the maintainability of the code. You have addressed the findings for Unit Size, and we also noticed that the new code added since the first

> upload generally contains smaller units than before. For Duplication, you have not addressed the specific example from the last email, but because the new code contains almost no duplication you have still managed to improve the rating for this metric. Finally, we also see an increase in the amount of test code.
>
> We therefore conclude that you have mostly followed the advise from the feedback on the first upload.

The second feedback is positive, just like the first feedback, but it is not exactly clear whether the rating that SIG provided changed. However, upon a request for clarification SIG provided the following response, indicating the rating had increased:

> We mainly look at changes compared to the first upload and how you addressed the feedback (both in terms if refactoring old code and new code). Since you're interested, you're still at 4 stars, but you went from 3.8 to 4.0. That seems small, but it's a good improvement considering the limited time, so well done.

7

DISCUSSION

This chapter evaluates the final product in 7.1 and makes recommendations for improvements of the product in the future by other teams in 7.2. Finally, ethical implications of this product are discussed in Section 7.3.

7.1. PRODUCT EVALUATION

The project is considered a success since most of the requirements have been fulfilled. All the must haves, should haves and could haves have been implemented to a certain degree. An overview is listed below of all the initial requirements and how they have been fulfilled, discussing the must haves (7.1.1), should haves (7.1.2) and could haves (7.1.2). The requirements are printed in bold.

7.1.1. MUST HAVES

The camera is **cheap, good quality** and **has IR up to 10m**. Other hardware used like the Edisons and Ubuntu server were already present in the existing architecture, so the only additional costs of the system are those of the camera (which was cheap), therefore the system is **cheap overall**. It also has a **small physical footprint**, because Edisons are very small and can be wirelessly connected to the cameras which are also small. The system will most probably not hinder the nurses.

The final solution provides **both livestream and VOD** and the barebones EyaVideoPlayer shows the possibility of **viewing the video in any browser**.

Encryption is partially implemented. RTMP-streams sent from the Edisons to the server are encrypted with Stunnel, so this connection is secure. Encrypting streams from the server to the client is not implemented, because the customer has already implemented HTTPS for its website and this can be reused for the streams, therefore it would be redundant to do this ourselves.

Finally, **the product is well-documented** with this final report and the written code is also well commented.

7.1.2. SHOULD HAVES

Cameras are **easily replaceable** because the system is implemented using the ONVIF protocol, which means any camera that is ONVIF compliant can be used. The livestream has **low latency** with a delay under 15 seconds. This is an acceptable delay for the customer, and getting close to realtime is not possible with HLS (which is needed to show the stream in all browsers). This is a trade-off between compatibility and delay.

The bandwidth usage of the system has not been tested, it is not certain that **bandwidth usage in the hospital is low**. But in the way the system is setup there is currently no room for improvement in terms of bandwidth usage, so it is as low as possible. A different solution which would lower bandwidth usage would be to use an USB camera which is connected with a cable to an Edison. But as was discussed in the Section 3.1.3 an IP camera seemed like the best choice, even though bandwidth usage suffers from this choice.

The system is **scalable** as more cameras and Edisons can be easily added and the server can easily be upgraded. It also has **a low resource usage per camera**. Transcoding and streaming the video from the Edison takes 20-50% of its CPU capacity, so there is still space for other processes to run on the Edison as well. The resource usage on the server has not been tested, because it is less relevant because server capacity can easily

be upgraded. The current setup of the streaming server is not the most efficient in terms of resource usage, because multiple conversions of the stream are performed.

7.1.3. COULD HAVES

Authentication of the streams is partially implemented. The RTSP streams broadcasted by the IP cameras are password protected, and the Edison contains the credentials to access this stream. The streaming server only accepts RTMP streams with a specific key in its URL, so a client (i.e. an Edison) needs to know this key to be able to stream to the server. Each Edison contains this key.

Authentication to watch a livestream or VOD is not implemented, but the foundation to add this has been laid out. Only tokenized stream URLs can be accessed, so without a token a stream cannot be watched. This token is generated on the server, and authentication can easily be added to this facility when integrating the system into the existing frontend application (since this already has authentication).

The system has not been tested in the hospital, it is not certain that **few router settings changes** are needed for the system to work. But the expectation is that the hospital needs to at least adapt the firewall to allow the outgoing stream to go through to a certain port. Issues might arise with discovery of the camera because of the firewall, so this might mean that more rules need to be adapted in the firewall for the system to work.

7.2. RECOMMENDATIONS

This section discusses recommendations for the system. First, general recommendations are given (7.2.1). Second, recommendations specific for the Edison are mentioned (7.2.2). Third, server recommendations are discussed (7.2.3). And finally, recommendations for the frontend are presented (7.2.4).

7.2.1. GENERAL RECOMMENDATIONS

More scalability research is needed. The resource usage on the server has not been checked and the system has only been tested with 1 or 2 concurrent streams. Research that answers the question how many concurrent cameras Nginx and/or the server can handle is important to know how scalable the system is.

Generally, the system needs to be tested in its production environment (i.e. the hospital). The bandwidth usage in the hospital is not known, and could pose limitations on the system (for example a maximum number of cameras that can be used at the same time in the hospital). Also it is important to see if the babies are not bothered by the cameras and if the cameras do not interfere with the work of the nurses.

Issues might arise with the firewall in the hospital, blocking outgoing streams and possibly also blocking the streams that are being broadcast by the IP cameras. This needs to be tested, and if it turns out that streams are being blocked, the firewall of the router needs to changed appropriately.

It is recommended to buy another Digoo camera and test if it is working properly, because from the two Digoo cameras used during implementation, one did not have properly functioning audio. It is important to determine if more Digoo cameras have this problem, because that means only streaming without audio can reliably be done. It is also recommended to test the system with different IP cameras to see if different ONVIF compliant IP cameras are indeed interchangeable. Finally, a white-colored camera is recommended because it blends in better with the hospital environment, so it is less disturbing for the baby.

7.2.2. EDISON RECOMMENDATIONS

Because an IP camera streams its video on the local network and the Edison receives this video through that network, the consequence is that when the network is down the video is lost. To prevent this from happening, a possible solution is to insert an SD-card into the camera on which the camera records its video. This can be used as a backup. This implies that the camera has the possibility to do this. The Digoo camera that is currently used does have an SD slot and claims that this is possible, but this has not been tested.

Each stream shows a timestamp on its video, which is put there by the camera. This timestamp does not necessarily reflect the correct time in the timezone the camera is located in. The timestamp can be changed using ONVIF, so the discovery application running on the Edison can be updated to automatically set the correct timestamp on a stream.

When the software running on the Edison needs to be updated, there is currently no automatic way to do this. This means that every Edison needs to be manually accessed to update the streaming software. A service which runs on the Edison which can be called to automatically update the streaming software to the latest version is recommended.

The FFmpeg command that is used for streaming on the Edison, stops streaming when an error occurs. This is to prevent it from sending a broken stream to the server. However, a situation was encountered with a different IP camera than the Digoo, where the IP camera produced an error once when it started streaming, but after that it worked fine. But the FFmpeg command caused this stream to stop immediately at that first error. Therefore manual error handling should be added in the bash script on the Edison to handle this specific use case.

The Edison is an embedded device, and is used for different purposes as well by the customer. This means that each Edison only has limited CPU capacity for processing the video stream. This is detrimental to the quality of the video that is being streamed from the Edison to the server. A solution would be to put a server in the hospital dedicated to streaming. This server could then also function as backup recording, instead of using SD-cards.

7.2.3. SERVER RECOMMENDATIONS

The customer currently runs it website from a Synology server, but plans to switch to a server running Ubuntu 14.04. When this is done, it is recommended to use Nginx as webserver for the frontend application, because this server can then easily be combined with the streaming server (which is also based on Nginx). A secure connection (over HTTPS) can then be handled the same way for both the application and the video streams. If the customer decides to use a different webserver, HTTPS still needs to be implemented for the video streams.

Authentication for the frontend application is already implemented, this existing authentication system should be used for authenticating the streams (the foundation for this already exists with tokenizing the streams). Again, when Nginx is also used for the webserver the same authentication can be used for both the frontend application and the video streams.

The password that is currently used as authentication for incoming RTMP stream is a simple string, generating a more secure password is advised.

If usage of the system increases, it might be more viable to switch to a paid streaming server, like Nginx Plus. The way the system is currently implemented, a lot of conversions are done along the way, which is not the most efficient solution. Ideally, the stream is only converted once from RTSP to HLS. When a streaming server with more advanced features is used, this is possible because it has the possibility to accept incoming HLS, record it, and serve it live and VOD. When the system is widely used this is an option to consider.

Currently, the only possibility to check if recorded video exists for a given time is to try and view that video in the EyaVideoPlayer (selecting the appropriate time using the datetime picker). It is recommended to implement a service on the server which checks for each hour whether recorded video is present. This service can then be used in the frontend to directly show for each hour whether video is present or not.

Regarding recorded video a script is needed to check once a day (or more often) if every FLV file has been properly converted to MP4. When a stream is suddenly stopped, the bash script to convert FLV to MP4 is not always executed, which is why this extra check is needed. Also, selecting video which overlaps between two days (for example selecting to watch video from 23:55 to 00:55), does not work properly, because it will only show the video for the current day. This is an issue that needs to be addressed.

It is recommended to run a service on the server which checks if a livestream is being received by the server. This way, it can be easily checked if all cameras are properly streaming to the server.

It is recommended implementing adaptive bitrate streaming, which is a key functionality of HLS and allows for viewing a stream in poor network conditions. This does increase the resource usage of the server, because streams with multiple quality levels need to be created.

Finally, when all aspects of streaming are properly functioning, it is recommended to implement motion detection and push notifications on the server. This could be really beneficial for the nurses and hopefully it will greatly improve the care that can be given to premature babies.

7.2.4. CLIENT RECOMMENDATIONS

The video player that is created should be integrated into the customer's multi-platform app. When doing this, the video player and datetime picker should be adapted to fit well into the existing design and to be fully responsive. For the datetime picker it is recommended to implement the bootstrap-datetimepicker which also has a calendar, instead of the currently used flatpickr.

Finally, one issue which has not been addressed is the issue with Daylight Savings Time (DST). On the server this will not produce any errors, because the streams are saved using Unix Epoch Time, which is not affected by DST. But with the datetime picker in the frontend this might not function properly, because when

the time goes back one hour, it means the same hour contains video for two hours. If the customer thinks it is worth the effort, he should look into this issue.

7.3. ETHICAL IMPLICATIONS

The system that is created does have an ethical aspect as well, which is discussed in this section.

A concern may be that the product adds to a "Big Brother" surveillance state from cradle to grave, as here babies are filmed. Also babies cannot consent to being filmed. These by itself may be a valid concerns about unwanted consequences, but these should be balanced out with the possible benefits of this product. A possible better healthcare for premature babies should tip the scales in favor of expanding and using this solution, as this could mean better conditions for premature babies. Also, since the video is secured, if the recorded footage is properly archived and/or disposed of, it is something that the baby (now an adult) or the parents could request in the future as precious memories that can be rewatched. But the customer would need to make clear agreements with the hospital about this, also taking in account the newly introduced General Data Protection Regulation (GDPR).

8

Conclusion

When starting this project the goal was to develop a system which is able to stream the video from a camera to a video player which can be watched in any browser. This goal has been achieved. The system still needs to be integrated in the existing front end application of the customer, and there is room for improvement in some parts of the system, but the foundation has been well established. Because of the extensive documentation of everything that has been done (both in this document and in the code itself), it is easy to build upon this system.

Looking at the requirements, most requirements have been implemented. The most important steps to take now is to integrate the video player into the existing front end, and to test the system in the hospital. The latter might yield new issues which have to be addressed.

Hopefully this system, when fully functional, can contribute to the well being of premature babies. If that is the case, this project is considered a big success.

GLOSSARY

FOSS	Free and open-source software. Anyone is free to use this software, and its source code is openly shared.
Proxy server	A server acting on behalf of another computer.
Forward proxy	Acting on behalf of a service consumer.
Reverse proxy	Acting on behalf of a service provider.
Transcode	Encoding video in a different format.
Localhost	A hostname that refers to this computer.
Service	Refers to an application running on a server continuously.
API	A set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service.
FastCGI	A protocol for interfacing interactive programs with a web server.
RTSP URI	The location of the RTSP stream on the local network.

BIBLIOGRAPHY

[1] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, *Overview of the h. 264/avc video coding standard,* IEEE Transactions on circuits and systems for video technology **13**, 560 (2003).

[2] C.-F. Lin, H.-T. Chiao, R.-K. Sheu, Y.-S. Chang, and S.-M. Yuan, *A fault-tolerant onvif protocol extension for seamless surveillance video stream recording,* Computer Standards & Interfaces **55**, 55 (2018).

[3] ONVIF, *Onvif profiles,* (2018).

[4] ONVIF, *Onvif profile feature overview v2.0,* (2018).

[5] *Transmission Control Protocol,* RFC 793 (1981).

[6] *User Datagram Protocol,* RFC 768 (1980).

[7] F. Baker, *Requirements for IP Version 4 Routers,* RFC 1812 (1995).

[8] D. S. E. Deering, *Host extensions for IP multicasting,* RFC 1112 (1989).

[9] P. Savola, *Overview of the Internet Multicast Routing Architecture,* RFC 5110 (2008).

[10] A. C. Begen, T. Akgul, and M. Baugher, *Watching video over the web: Part 1: Streaming protocols,* IEEE Internet Computing **15**, 54 (2011).

[11] G. J. Yang, B. W. Choi, and J. H. Kim, *Implementation of http live streaming for an ip camera using an open source multimedia converter,* International Journal of Software Engineering and Its Applications **8**, 39 (2014).

[12] C. Mueller, *Mpeg-dash vs. apple hls vs. microsoft smooth streaming vs. adobe hds,* (2015).

[13] R. Pantos, *HTTP Live Streaming 2nd Edition*, Internet-Draft draft-pantos-hls-rfc8216bis-00 (Internet Engineering Task Force, 2017) work in Progress.

[14] I. Sodagar, *The mpeg-dash standard for multimedia streaming over the internet,* IEEE MultiMedia **18**, 62 (2011).

[15] Apple, *Best practices for creating and deploying http live streaming media for apple devices,* .

[16] K. Norrman, E. Carrara, D. D. A. McGrew, M. Naslund, and M. Baugher, *The Secure Real-time Transport Protocol (SRTP),* RFC 3711 (2004).

[17] E. Rescorla and T. Dierks, *The Transport Layer Security (TLS) Protocol Version 1.2,* RFC 5246 (2008).

[18] E. Rescorla, *HTTP Over TLS,* RFC 2818 (2000).

[19] Wowza, *Wowza streaming engine streaming server software,* (2018).

[20] MariaDB, *Data types in mariadb,* (2018).

[21] J. Sebastian, *An introduction to sql server filestream,* (2018).

[22] R. Sears, C. Van Ingen, and J. Gray, *To blob or not to blob: Large object storage in a database or a filesystem?* arXiv preprint cs/0701168 (2007).

[23] D. Ahmed, M. K. Hasan, *et al.*, *IPTV Video Streaming in Content Distribution Network*, Ph.D. thesis, East West University (2014).

[24] *Ffmpeg documentation,* `http://ffmpeg.org/ffmpeg.html`.

[25] X. Lei, X. Jiang, and C. Wang, *Design and implementation of a real-time video stream analysis system based on ffmpeg*, , 212 (2013).

[26] *Stunnel documentation*, https://www.stunnel.org/static/stunnel.html.

[27] Netcraft, *April 2018 web server survey*, (2018).

[28] video dev, *hls.js*, https://github.com/video-dev/hls.js/ (2010).

[29] VideoJS, *videojs-contrib-hls*, https://github.com/videojs/videojs-contrib-hls (2013).

[30] Onvif, *Onvif core specification*, (2012), version 2.1.1.

[31] *gsoap documentation*, https://www.genivia.com/dev.html.

[32] *Oasis web services dynamic discovery (ws-discovery) version 1.1*, http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html (2009).

[33] *What is nginx?* https://www.nginx.com/resources/glossary/nginx/ (2018).

[34] R. Arutyunyan, *nginx-rtmp-module*, https://github.com/arut/nginx-rtmp-module (2012).

[35] Kaltura, *nginx-vod-module*, https://github.com/kaltura/nginx-vod-module (2014).

[36] Kaltura, *nginx-secure-token-module*, https://github.com/kaltura/nginx-secure-token-module (2014).

[37] Kaltura, *nginx-akamai-token-validate-module*, https://github.com/kaltura/nginx-akamai-token-validate-module (2015).

[38] CatchOrg, *Catch2*, https://github.com/catchorg/Catch2 (2010).

[39] Jasmine, *Jasmine*, https://github.com/jasmine/jasmine (2008).