MASTER'S THESIS | LUND UNIVERSITY 2015

Cross-Platform Video Management Solutions

Thomas Mattsson, Andreas Olsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-24

Cross-Platform Video Management Solutions

Thomas Mattsson
thomas.mattsson.2@gmail.com

Andreas Olsson
olsson.andreas@gmail.com

June 10, 2015

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

Supervisors: Fredrik Brozén, fredrik.brozen@axis.com
Per Ganestam, per.ganestam@cs.lth.se

Examiner: Mathias Haage, mathias.haage@cs.lth.se

Abstract

With a multitude of platform and operating system combinations available today, ranging from laptops and workstations to tablets and smartphones, users want to use their favorite applications regardless of device. Cross-platform development has thus become more important in recent years. When developing a new application the developers must decide what platforms to support and what strategy to use to reach out to them. By developing both native and cross-platform prototypes we try to find advantages and disadvantages of using a cross-platform strategy for video management applications. We show that it indeed is possible to develop cross-platform video management applications for both Windows and OS X and find both advantages and disadvantages of this strategy. The result of this thesis state that the choice of cross-platform or not depends much on the situation and the preferences of the developers.

Keywords: cross-platform, native, software development, comparison, video management

Acknowledgements

We would like to thank our supervisor Fredrik Brozén at Axis for his support during the project. We would also like to thank our supervisor Per Ganestam and examinator Mathias Haage at Lund University for their feedback. Finally we thank the SA team at Axis for their support and openness.

Contents

Acronyms

ACC AXIS Camera Companion. 12, 29

ACS AXIS Camera Station. 12, 15, 20, 27, 29, 41, 46, 57

API Application Programming Interface. 14, 19, 31, 55

CSS Cascading Style Sheets. 22

FPS Frames Per Second. 24, 43, 46

GCC GNU Compiler Collection. 36

GUI Graphical User Interface. 21, 22, 28–31, 33–35, 37, 38, 45, 46, 50, 53–58

HTML HyperText Markup Language. 14, 22

IDE Integrated Development Environment. 22, 32, 33, 36

IP Internet Protocol. 19

LOC Lines Of Code. 41, 42, 44, 45, 52, 53

MSVC Microsoft Visual C++. 36, 42

NAS Network-Attached-Storage. 12

TCP Transmission Control Protocol. 16

UDP User Datagram Protocol. 15, 16

UI User Interface. 23, 31, 32, 35, 37, 38, 41

VMS Video Management Software. 11, 12, 53, 58

WPF Windows Presentation Foundation. 29, 55

Chapter 1

Introduction

Today people use many different devices both at home and at work, for example different kinds of computers, mobile phones and tablets. The demand from users that applications should work on several different operating systems have increased in recent years since we today use more different devices than before. This is because today we have powerful smartphones and because laptop performance and battery life have been greatly improved. When developing software it is therefore desirable to support a wide range of platforms.

An application that works across different platforms or operating systems is often said to be *cross-platform*. Other commonly used names for this property are multiplatform, platform-independent and portable, see [1] for more information. In this report we will use cross-platform. An important development decision when developing software for several different platforms is what strategy to use. Ideally there are two different strategies, either you develop separate *"native"* applications for each platform, or you develop one *cross-platform* application that can be deployed on all platforms. Cross-platform applications often compile into different platform-specific binaries though but shares the underlying codebase between the platforms. A native application is an application that is written in the programming language supported natively by the corresponding platform and that is targeted just to that specific platform. In reality it can be hard to achieve a fully cross-platform application, often some platform-specific code is needed.

The demand for cross-platform applications applies to more or less all types of applications, for example web browsers, music players, word processors and video applications. In this master thesis project we will investigate the possibilities for a cross-platform Video Management Software (VMS) for Axis Communications AB and compare cross-platform and native development.

A VMS is used for managing surveillance camera installations, supporting monitoring of live and recorded video. A VMS can also include features such as event and alarm handling and logging.

1.1 Axis Communications AB

This master thesis project was carried out at Axis Communications AB in Lund, Sweden. Axis is an IT company offering network video solutions for professional installations. It is a Swedish-based company that was founded in 1984. Axis offers a wide portfolio of products, for example network cameras, video encoders, network video recorders and VMS.

For many years Axis has developed VMS solutions to meet the need for efficient surveillance of small to midsize installations. For small installations Axis offers the system *AXIS Camera Companion (ACC)* for Windows, Android and iOS. In ACC the video is stored on either an SD-card in the camera or remotely on a Network-Attached-Storage (NAS) and the client application connects directly to the camera to fetch the video. For midsize installations with up to 100 cameras Axis offers a system called *AXIS Camera Station (ACS)* which currently only is available for Windows. ACS is a client-server solution where also the server runs on Windows. For larger installations third party applications are used.

Historically Axis solutions targeted Windows as the platform since that is what customers have expected. However lately they have seen an increasing demand from customers to support a range of different platforms, mainly Windows and OS X but also mobile platforms such as Android and iOS. Axis therefore wants to investigate the possibilities to support the *ACS client* on different platforms.

1.2 Problem Definition

The main goal of this master thesis project is to compare cross-platform development with native development for VMSs. This means to perform an investigation with the purpose of finding advantages and disadvantages regarding development and maintenance using the two strategies.

The goal is also to support a simple version of the ACS client, providing a small subset of the ACS functionality, on different platforms and to implement a prototype of it. This means that the prototype should support streaming video from Axis cameras on Windows and at least one more operating system. Which other operating system the application will be developed for will be decided based on the initial study in the area. More operating systems can be added if it is considered that there is time for that. This goal also means that the clients should share the same codebase to as great extent as possible to avoid maintaining several different codebases. The application should also support displaying at least two parallel video streams simultaneously, possibly with lower quality depending on platform. It is not a requirement to use the existing code and programming language of the ACS client.

1.3 Previous and Related Work

To achieve cross-platform applications have been desirable for software companies for a long time. If and how to develop cross-platform applications has been an important strategy decision for as long as end users have been using different platforms and companies

have seen a market outside their currently supported platform.

In 1994 Netscape Communications Corp was established and the same year they released the cross-platform web browser Navigator. Netscape became distinguished for its ability to develop Internet software for all major personal computer platforms. Cross-platform development was an important and central part in the company's business strategy. Developing cross-platform was not always easy though and to compete with the competition from other companies, Netscape wrote more platform-specific code over time. Much time was required for development and testing and the performance could be weaker compared to platform-specific solutions. To ease cross-platform development Netscape developed the Netscape Portable Runtime (NSPR) as a low-level abstraction layer that worked on all supported platforms. NSPR included operating system abstractions for e.g. file access, threading and socket I/O. To make it work on all platforms resulted in some disadvantages in performance though. The NSPR also required a separate development team and it was hard to keep the layer in pace with the development of the different platforms and techniques. What can be learned from Netscape's history is the design techniques they used, to share components and use platform-specific code when needed. [2]

Netscape's browser later developed into Mozilla Firefox, which is one of the major web browsers today, which still uses NSPR. Very little is left from the original source code of NSPR and Mozilla writes that "Many of the concepts have been reformed, expanded, and matured". Today NSPR is considered functionally complete and the basic API is stable. The layer has entered a mode of sustaining engineering and will be moved forward when new operating systems are released. [3, 4]

In 2000 the United States Navy Research Laboratory released the report "Cross-Platform Development: A Difficult Necessity" [5] where they gave an overview of cross-platform development using the C++ programming language and investigated existing cross-platform applications for different categories of applications. The motivation behind this report was to reach a larger audience and to help make development decisions when creating new software. In this report it was found that three different cross-platform development models were widely used:

- **Double source tree:** Means developing separate copies of the application for each platform. Maintaining and debugging a project of this model can be hard, especially if the project is large.
- **Single source tree emulation:** Means developing the application for one platform and use some emulation tool to make it run on other platforms. This means that only one application is developed but there may be problems with compatibility and performance.
- **Single source tree translation:** Means developing one application using an abstraction layer to the underlying architecture (like the NSPR described above). The abstraction layer translates function calls so that they work on the current platform.

The latter one is generally the most preferred model since only one application is developed and performance is usually better than using the emulation model. Using this model usually means relying on some sort of cross-platform framework though and there may be problems with support lacking for some functions on some platforms, depending on the framework used.

Two related master thesis projects have been done, both carried out at Axis, namely ”Real-time video streaming with HTML5” [6] and ”Software Portable VoIP Client” [7].

In ”Real-time video streaming with HTML5” the feasibility was investigated for showing a real-time video stream from a network camera in a web browser using HyperText Markup Language (HTML) version 5. This means the solution should be plugin-free by instead utilizing the new features and Application Programming Interfaces (APIs) of HTML5. The main focus of the project was on the H.264 video format and the Media Source Extensions API. The project showed that it was possible to provide real time video streaming without plugins with low latency and high bandwidth using the Media Source Extensions API together with client-side MP4 fragmentation. The solution was not ideal though since the Media Source Extension was very new and not supported by all major browsers. [6]

In ”Software Portable VoIP Client” different methods to achieve software portability (support cross-platform) for a Voice over Internet Protocol (VoIP) client was investigated and examined. The purpose of the project was also to find practical evaluation methods for the VoIP client. A prototype was developed using the open-source project WebRTC which enables real-time communication in web browsers using JavaScript. The conclusion was that it performed well under the right circumstances but that WebRTC was not stable enough to be recommended as a serious solution. WebRTC was still in draft though at the time when the master thesis was carried out. [7] What is most interesting for our project is that in this master thesis project *web technologies* was chosen as solution to make the application cross-platform.

1.4 Contributions

In this master thesis the work was divided evenly between the authors. However the Windows-specific development was done by Thomas and the OS X-specific development was done by Andreas. We have discussed different solutions to all problems along the way, and thus we have reached a collaborative result.

Chapter 2
Technologies

This chapter will describe the different technologies that have been used. We start by describing the underlying techniques including network video streaming, protocols, video and pixel formats and network cameras. We continue by describing ACS and the different platforms and cross-platform frameworks we have considered. Finally the main tools used will be described.

2.1 Network Video Streaming

A stream is a way to transfer data in a continuous flow. This means that the receiver can interpret the data as it arrives, instead of waiting to complete the whole transfer before it interprets the data. Completing the whole transfer is not always possible either, it might be a stream that has no end.

In this master thesis project streaming is used in the context of video streaming. Thus a stream is a stream of video data that the camera is sending to the receiver which interprets the video data and shows the video on the screen.

2.2 Protocols

A protocol is a set of rules that describe how communication between two computer systems should be carried out. [8] We continue by describing the network protocols that we have used.

2.2.1 Real-time Transport Protocol (RTP)

The Real-time Transport Protocol (RTP) is a protocol that provides end-to-end delivery of real-time data, for example audio and video. RTP is usually used on top of the User Data-

gram Protocol (UDP), although other transport protocols may be used as well. When RTP is used on top of a transport protocol that provides the possibility to use multicasting, as when used on top of UDP, it supports the possibility to send data to multiple destinations.

RTP on its own does not make sure that the packets are delivered on time or that the packets are delivered at all. It relies on the transport protocol to handle those quality-of-service guarantees. Each packet that is sent is however numbered so the client can re-order them if they are received out of order. [9]

2.2.2 Real-time Streaming Protocol (RTSP)

The Real-time Streaming Protocol (RTSP) is a protocol which makes it possible to establish and control media streams. RTSP does not take care of the actual transport of one or several streams, it does however take care of the possibilities to for example play, pause and terminate the streams. So RTSP can be seen as a kind of remote control for the multimedia streams.

Since RTSP does not take care of the transportation of the packets, several different protocols can be used to handle the transport. The protocols that can be used for the transportation is for example Transmission Control Protocol (TCP), UDP and RTP. [10]

2.2.3 Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) is a protocol where one party acts as a server and the other as a client. The client sends requests to the server and the server sends a response back. Typically the client is a web-browser and the server is a web-server. There are different kinds of request methods that the client can use, for example GET and POST, where GET requests to get information from the server and POST is similar to GET but can also be used to send information to the server.

The protocol commonly uses a TCP connection for communication between a server and a client, although the only requirement HTTP has on the transport protocol is that the transport is reliable so any protocol that can guarantee the reliability can be used. [11]

2.2.4 RTP over RTSP over HTTP

RTP and RTSP can be used together to stream data. Unfortunately this traffic can be blocked by HTTP proxies and firewalls and therefore Apple has created a protocol called *RTP over RTSP over HTTP* where the traffic is tunneled via HTTP. The protocol uses the fact that the HTTP GET and POST methods can contain an indefinite amount of data. Basically the communication is setup by the client first makes a HTTP GET request and then a HTTP POST request to the server. The GET request opens the server-to-client channel and the POST request opens the client-to-server channel. [12]

2.3 Video Formats

A video format is a combination of a media container and the related codecs that are needed. The media container describes the file structure, for example where the various

data is stored within the file and which codecs that are being used for each part of the data. The data stored within the container has been encoded by the codecs, and to play the video the codec has to decode the video data. It is the codecs that provide the video compression mentioned in the next section.

2.3.1 Video Compression

Data compression is the process of reducing the amount of data needed for transmission or storage of data. This is often achieved using encoding techniques. Using data compression is important in a digital context to reduce the bandwidth and storage space needed for data. It is often possible to reduce the data size while still maintaining good quality.

Data compression can be *lossless* or *lossy*. Lossless compression means that the compression can be reversed to yield the original data which is not possible with lossy compression. When reversing lossy compression, details are lost or small errors are introduced. These types of compression are suited for different situations. For example, lossless compression could be needed for text where it is important to restore the original data, while for images, video and audio lossy compression may be acceptable. [13]

For video applications lossy compression is often used. In the lossy video encoding techniques that are used, data that have a small impact on the video quality is removed while data that is significant for the quality is preserved. For video applications it is often possible to remove a large part of the data while still keeping a good video quality.

There exist standards for video compression, specifying how the compression should be performed. This for example makes it possible for users to choose which vendor they want to use instead of having to choose the one vendor who provides the technology for their needs.

2.3.2 H.264

H.264 is a modern video encoding standard that has become popular because it can reduce the video size by more than its predecessors while not compromising the video quality. This is a positive thing for video streaming since a smaller file size means that you need less bandwidth and less storage space. It can also be seen as one can achieve higher video quality with the same size. H.264 is typically used for lossy compression but it is also possible to create lossless encodings with it.

In H.264 there are three different types of frames, I-, P- and B-frames. These frames contain different information. The I-frames (intra frames) are independent frames. They do not need information from any other frames to be decoded. The first frame in a video sequence is always an I-frame. They can also be used as starting points for new viewers and can be used as re-synchronization points if the stream has become corrupt as well. The I-frames can be used for rewinding and fast-forwarding among other access functions. The I-frames have the drawback that they are large in size, so just using I-frames would create a need for a lot more bandwidth in comparison the combining the different types of frames.

P-frames (predictive inter frames) are used to reduce the bandwidth that is needed by having references to earlier I- or P-frames, and thus not having to contain all the information that is needed to decode it. The drawback is that any transmission error could make the frame corrupt, and thus not working as intended.

The B-frames (bi-predictive inter frames) has references to both previous and future frames. These have the drawback that they increase the latency of the video stream since it requires the decoder to look at both previous and future frames, which increases the delay. The advantage of a B-frame is that since they can reference data in previous and future frames, they can be kept smaller and thus reduce the bandwidth that is needed.

To decode a video stream the decoder first starts with an I-frame and what happens next depends on what kind of frames that follows. If it is a P- or B-frame they are decoded together with their referenced frames and if it is an I-frame it is decoded on its own. [14, 15]

2.4 Pixel Formats

When encoding an image the format of the pixels needs to be determined. For this there exist several different pixel formats, the format describes the memory layout for each pixel in the image. We continue by describing the *RGB* and *YUV* formats.

2.4.1 RGB

Digital images are often encoded in RGB format. RGB stands for Red, Green and Blue which are the primary colors for the color model. In the RBG format a color is encoded using these three values which corresponds directly to portions of the visible spectrum. With this format a broad array of colors can be reproduced. The RGB color model is an additive color model, meaning that the three values are added together to get the resulting color.

2.4.2 YUV

Digital video is often encoded in YUV format. YUV is a family of color spaces where color is encoded using three values Y, U and V. Y represents the brightness value of the color, it is also called *luma*. The U and V components are called *chroma* or *color difference* values and represents the color information.

Historically video was encoded in YUV format so that the signal transmission for color television also was compatible with black-and-white television. YUV was used because black-and-white televisions could just use the Y component and color televisions could also use the UV components. Since the black-and-white signal already existed in the current television infrastructure only the UV components were added.

Another advantage of YUV that is more relevant today is that it takes human perception into account which makes it possible to reduce the bandwidth used. The human eye is more sensitive to changes in brightness than in hue, therefore an image can contain more luma (Y) information than chroma (UV) information without affecting the perceived quality of the image. Hence it is common to downsample the chroma information. A YUV image is not necessarily smaller than if it was encoded in RGB though. If the chroma information is not downsampled, a YUV pixel has the same size as an RGB pixel. [16]

2.5 Network Cameras

A *network camera* or *IP camera* is a camera that is connected to an Internet Protocol (IP) network and primarily sends video and audio over the network. The network can for example be a local area network (LAN) or the Internet.

A related type of camera is *web camera*. What differs a network camera from a web camera is that a web camera only can operate when it is connected to a computer via for example a USB port, running specific software for the camera. A network camera is connected directly to a wired or wireless network and can be placed wherever there is a network connection and thus can run on its own. Network and web cameras typically have different fields of application, where network cameras often are used for surveillance and web cameras may be used for video conferences and Internet chats. Therefore they are designed in different ways, targeting different groups of users.

Axis describes a network camera as "a camera and computer combined in one unit" [17]. The camera part is responsible for capturing the video using a lens and image sensor. The computer part is responsible for e.g. image processing, compression and network functionalities. Axis cameras enable viewing of live or recorded video. Recording can be done continuously, when triggered by events or at scheduled times. The cameras also include a web server which means that they can be accessed by typing the camera's IP address in the location field in a web browser. In this web interface users can e.g. configure camera settings, define user access and add action rules for events.

To receive and configure an Axis camera video stream, Axis has implemented a video streaming API called *VAPIX*. The VAPIX specification can be found in [18] and specifies which protocols are available and how to connect to the camera. In version 3 VAPIX supports video streaming over HTTP and RTSP.

2.6 AXIS Camera Station (ACS)

AXIS Camera Station (ACS) is one of Axis' applications for displaying live and recorded video. It is a client-server system where cameras are connected to a server and then a client connects to the server to access the cameras, Figure 2.1 shows an overview of the system communication. To communicate with the cameras the protocol RTP over RTSP over HTTP is used, and the communication between the client and the server is done using a custom protocol also using HTTP. One of the reasons why these protocols are used is to make it easier for the communication to get through firewalls and proxy servers since other protocol are often blocked.

2.7 Platforms

Today there exists many different software platforms, for example Windows, OS X, Linux, Android, iOS and Windows Phone. It is common today that people use several different platforms every day, when for example using smartphones or computers at home or work. In this section the platforms used in this master thesis project will be described briefly.

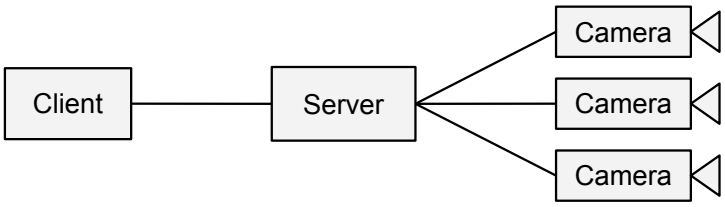

Figure 2.1: An overview of the communication between the ACS client, ACS server and cameras.

2.7.1 Windows

Windows is a family of operating systems developed, marketed and sold by Microsoft that e.g. includes operating systems for desktops, servers and phones. We will look at the desktop version of Windows. The first release of Windows was in 1985. Since then the operating system has been further developed and released in several versions, the currently latest released version is Windows 8.1. The next version of Windows, Windows 10, is scheduled to be released in 2015.

Windows 8 applications can be written in C#, C++ and JavaScript using the *Windows Runtime*. A Windows Runtime app can be run on a Windows device such as a desktop computer, tablet or phone. In this way the same Windows app can run on different types of Windows devices. [19] Traditionally native Windows applications have been developed in C# or C++.

2.7.2 OS X

OS X, previously known as Mac OS X, is a series of operating systems developed and marketed by Apple Inc. OS X is an Unix-based operating system and is designed to run on Mac computers and has been pre-installed on all Macs since 2002. OS X is the successor to Mac OS which Apple released in 1984. The current latest released version of OS X is 10.10 “Yosemite”.

Traditionally native OS X applications have been developed in the programming language *Objective-C*. Recently Apple announced a new programming language called *Swift* to be supported on both its mobile iOS devices and also on Mac. Swift and Objective-C code can co-exist in the same project. [20]

2.8 Cross-Platform Development Models

As the United States Navy Research Laboratory found in their report [5], described in section 1.3, there are different ways or *models* for how to develop cross-platform applications. In this master thesis project we will focus on *single source tree translation* and also on using a single source tree but with techniques and frameworks that work on several platforms without translation. We chose these models because we found them most interesting since they help to prevent having to maintain several codebases and should have better performance than using emulation.

There are variations of these models though and there can be different degrees of how much source code that is shared between the different platforms. To completely use a *single* source tree, the entire codebase would have to be shared, including the Graphical User Interface (GUI). One way to achieve a truly native GUI though would be to share as much code as possible between the platforms, but to develop the GUIs separately.

Cross-platform frameworks can make it possible to develop cross-platform applications without the need of a translation layer. Cross-compilation can be used instead to compile different binaries for different platforms or by using a programming language that runs in a virtual machine like Java or C#, and thus no extra translation layer is needed.

2.9 Cross-Platform Techniques and Frameworks

In this section the different cross-platform techniques and frameworks we have found and considered will be described. Several potential techniques and frameworks were found. Both programming languages and frameworks were searched for.

One does not necessarily use only *one* framework or programming language when developing a cross-platform application. One approach could e.g. instead be to use one framework for desktop applications and another for mobile applications. This can be the case if there exist no one well functioning framework to cover all platforms. Using different frameworks can come at a cost though, compared to using a single framework, but can be a good alternative if the single framework does not function well or does not support all platforms.

2.9.1 Programming Languages

Several programming languages can be used to develop applications that run on multiple platforms. In this section some of the languages that were considered for the cross-platform prototypes are listed and described.

C#

C# is developed by Microsoft as a part of its .NET initiative. Using platforms such as Mono/Xamarin it is possible to run C# applications also on OS X, Linux, iOS and Android.

C++

There exist many C++ compilers and it is possible to compile C++ applications to target Windows, OS X, Linux, iOS and Android (using the Android NDK toolset).

Java

Java applications run in a virtual machine called Java Virtual Machine (JVM) and thus applications written in Java works on platforms that have a virtual machine implemented.

Java is supported on Windows, OS X and Linux. Android applications are also written in Java.

Web (HTML, CSS and JavaScript)

Web applications is an area that has been on the raise for several years and is used in several applications today. Web applications use HTML for the structure of page, Cascading Style Sheets (CSS) for styling and JavaScript for the application logic. The drawbacks of using web is that it often makes the application less responsive than it would have been if it was developed natively for the platform.

Python

The design philosophy of Python focuses on code readability. Python supports many programming paradigms, including object-oriented, imperative, functional and procedural styles. There exist Python interpreters for many operating systems, making it a cross-platform language.

2.9.2 Frameworks

There exists many frameworks and tools for creating cross-platform applications supporting different approaches and for different programming languages. In this section we list and describe some of the frameworks and tools considered for the cross-platform prototypes. We list platforms for both desktop and mobile platforms. We will only develop prototypes for Windows and OS X though, but for further work these frameworks can be interesting.

Mono/Xamarin

Mono [21] is an open source implementation of the .NET framework that is designed to allow easy creation of cross-platform applications using the C# programming language. Xamarin [22] is a company that uses and sponsors Mono and provides a development platform, also called Xamarin, including various extensions, an Integrated Development Environment (IDE) called Xamarin Studio and business support. Xamarin can be used to develop both desktop and mobile applications.

Qt

Qt [23] is a C++ cross-platform application framework that allows targeting both desktop and mobile platforms with little to no changes to the underlying code. There exists modules for creating GUIs that are shared for the different platforms, with different styling which makes them look and feel native. Qt also includes an IDE called Qt Creator.

Apache Cordova/PhoneGap

Apache Cordova [24] is a platform for building *mobile* applications using web technologies, i.e. HTML, CSS and JavaScript. Apache Cordova makes it possible to use JavaScript

to access native device functions such as the accelerometer and camera. It is possible to use User Interface (UI) frameworks to provide a touch interface. Basically the application displays web pages stored locally on the device. The *PhoneGap* framework [25] is an open source distribution of Cordova. In PhoneGap's FAQ they give the following comparison "Think about Cordova's relationship to PhoneGap like WebKit's relationship to Safari or Chrome." [26].

TideKit

TideKit [27] is a framework for creating *desktop, mobile and web* applications using web technologies. It is a further development of TideSDK [28], which only supported desktop applications. TideKit uses a "Develop Once, Deploy Everywhere" approach meaning the same shared source code can be used on all platforms. TideKit is currently in development and no official release exists yet.

Electron

Electron [29] is an open source framework maintained by GitHub for creating cross-platform *desktop* applications using web technologies. It was initially developed for GitHub's text editor Atom, but has later been used by for example Facebook. It is based on io.js and the Chromium browser.

React Native

React Native [30] is a framework maintained by Facebook for developing native applications using JavaScript based on *React*. React is also maintained by Facebook and is a JavaScript library for building UIs. Facebook currently uses React Native in several production applications. The framework is currently only available for iOS but Android support is under development.

J2ObjC

J2ObjC [31] is an open-source tool maintained by Google allowing translation of Java code to Objective-C for iOS. There is no support for UI and the goal with the tool is to write the non-UI application code in Java and share it with web, Android and iOS applications. Google believes writing UI code needs to be done separately for the different platforms, i.e. iOS UI code needs to be written in Objective-C or Objective-C++.

JUniversal

JUniversal [32] is a project aiming to allow development of primarily *mobile* applications in *Java* by providing source code translation to C# for Windows Phone and C++/Objective-C++ for iOS. The C++/Objective-C++ translation is currently under development though. To produce Objective-C code, JUniversal recommends using *J2ObjC*. There is no support for UI, instead developers are supposed to write this natively to provide a good user experience. JUniversal also provides a set of Java libraries called *JSimple* supporting functionality commonly needed in mobile application development.

Haxe

Haxe [33] is a cross-platform toolkit providing a programming language, called Haxe as well, together with a cross-compiler that translates the code to native source code or binary for the target platform. Currently Haxe supports development of *desktop, mobile and web* applications.

2.10 Tools

To support and ease the work with the prototypes some different tools were used. The tools were used for testing, video handling and evaluation. The tools are described briefly with references linking to more information.

2.10.1 Wireshark

Wireshark [34] is a free and open-source cross-platform network protocol analyzer. Wireshark can be used to analyze stored and real-time network traffic which can be helpful when developing network applications. In Wireshark the sent and received packets can be analyzed and the traffic can be filtered.

2.10.2 FFmpeg

FFmpeg [35] is a free software project for handling of multimedia data including both command line programs such as `ffmpeg` and `ffplay` and a set of libraries such as `libavcodec` and `libavformat`. FFmpeg can e.g. be used to get video information (such as resolution and Frames Per Second (FPS)), transcode video into different formats, play video files and network streams and to decode video data. FFmpeg contains more than 100 codecs and it is used by several well known applications such as VLC media player, HandBrake and Blender.

FFmpeg is licensed under the LGPL license. [36, 37] There are some optional modules and optimizations of FFmpeg that are released under the GPL license, if these are used FFmpeg is considered to be used under the GPL license.

2.10.3 Process Explorer

Process Explorer [38] is a freeware system monitor and task manager application for Windows. Its basic functions are the same as of the default task manager on Windows but Process Explorer offers more functionality such as displaying detailed resource utilization per process, GPU activity and finding open file handles. This tool has been used to measure the performance of the prototypes on Windows.

2.10.4 Activity Monitor

Activity Monitor [39] is a program that comes pre-installed on OS X. It shows all the running processes on the computer and for example how much CPU-power and memory the

processes are using. This tool has been used to measure the performance of the prototypes on OS X.

Chapter 3
Methodology

To fulfill the goals described in section 1.2 a number of prototypes were developed. First native applications were developed and then some cross-platform tools were used to develop cross-platform applications.

3.1 Prototype Requirements

The following minimum requirements were set for the prototypes to support:

- Receive video data over HTTP.
- Play H.264 encoded video.
- Receive two video streams simultaneously.
- Play two video streams simultaneously.

3.2 Server

Because the current ACS server was developed for Windows clients there were some problems connecting non-Windows clients to it. Axis also had plans to update the protocol used between the server and the client. Therefore, in discussions with Axis, we decided to implement our own minimal server allowing platform-independent clients to connect to cameras over HTTP and then receive frames using our own custom application protocol. The server was implemented in Java.

When we started to develop client applications we realized that it was preferable to use RTP over RTSP over HTTP, which was the protocol used for connecting the server to the cameras, as protocol also for the communication between the client and server. This was because this protocol is well tested and supported by various software frameworks

and libraries. This allowed us to develop the clients faster, but also meant that the server became unnecessary from the client's point of view and the scope of this master thesis project. This was because that if this protocol was used, the server basically would resend the packets received from the cameras, with just minor modifications. For the client there would basically not be any difference in connecting directly to the cameras and connecting to the server. For the system as a whole there is of course a difference as the server acts as an access point for the clients and can offer features like listing of available cameras and recording. The focus of this master thesis project though is on the clients and therefore we decided to not use the server and instead connect directly to the cameras.

3.3 Prototype Design

When developing the prototypes we used a main design across all prototypes. This design was created with object-oriented programming in mind, as well as using threads and monitors, since this is the way we have experience of programming and it seemed like a good design for the prototypes. There were only minor changes and deviations needed when using the different frameworks and programming languages. The design is illustrated in Figure 3.1 and consists of six classes:

Client The class containing the main function, it starts the client and creates instances of the other classes.

Camera For each physical camera a `Camera` class is created responsible for starting the threads for fetching video and updating the GUI.

VideoFetcher The thread that utilize FFmpeg to fetch and decode the video stream. The decoded video frame is converted from YUV-format to RGB-format so that it can be displayed in the GUI.

FrameBuffer The monitor used to store the decoded video frames.

ImageUpdater The thread that pulled images from the `FrameBuffer` object and then updated the GUI.

GUI The GUI of the client.

When using a development model where the GUI is not shared between the platforms the classes most suitable for sharing is `VideoFetcher` and `FrameBuffer`. These classes contain the main application logic by being responsible for fetching and storing the video respectively. Of course some classes like `Camera` and `Client` will also need to be present in some form but the classes regarding the GUI need to be customized for each platform.

3.4 Native Prototypes

Native applications were developed for Windows and OS X. In this section how these prototypes were developed and what approaches and techniques that were used will be described.

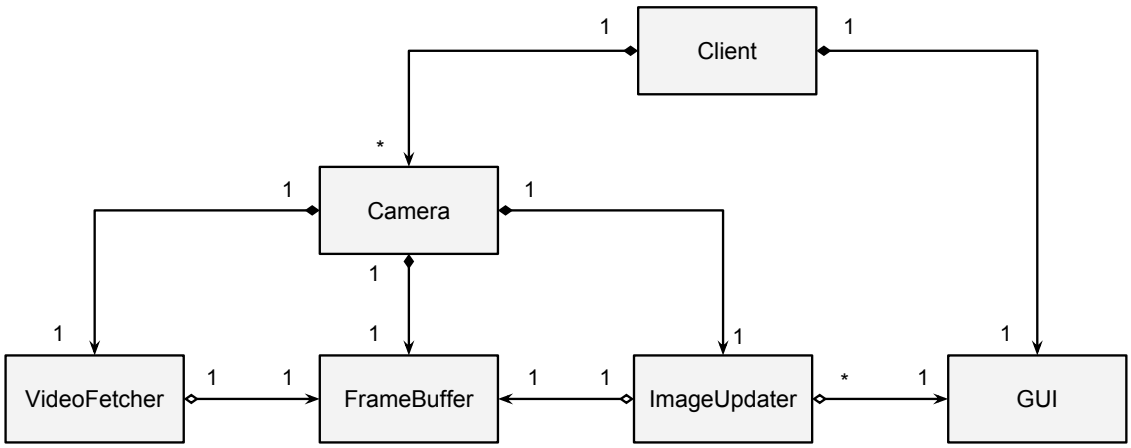

Figure 3.1: The main software design for the prototypes.

3.4.1 Windows

When developing the native Windows client many different techniques and frameworks were considered. The main question was what frameworks to use for decoding and displaying the video.

In ACS and ACC Axis uses Microsoft's filter-based *DirectShow* framework for decoding and displaying video. The applications' GUIs are built using Windows Presentation Foundation (WPF).

To resemble the ACS client, WPF was chosen for displaying the video and for building the GUI. Because of this it was also decided to implement the application in C# to as great extent as possible. For displaying video WPF offers an element called *MediaElement* but this element only support playing files and not streams. There exists some third party implementations such as WPF MediaKit [40] or Axis own solution to get around this although we did not get this to work and focused on other techniques instead. The solution we finally went with was using a WPF *Image* element and repeatably update the image source. This solution required us the convert the image from YUV-format to RGB-format since this is the format that the element supported.

For decoding the video stream we instead used *FFmpeg*. FFmpeg has support for reading streams and for reading the protocol RTP over RTSP over HTTP. To use FFmpeg in C# we used the C#/.NET wrapper *FFmpeg.AutoGen* [41]. This made it easy to use FFmpeg in C# to decode the video and then update the image element with the decoded and converted image. The prototype can be seen in Figure 3.2.

3.4.2 OS X

When developing the native OS X client we had to do it from scratch with no reference to an already existing system, although some inspiration was taken from the existing ACS client when considering how it should work.

The application is written in Objective-C and C. *FFmpeg* is used for reading the stream and also for decoding the video. Since *FFmpeg* is written in C and you can write C code

Figure 3.2: The native Windows client.

within Objective-C code, all the interactions with *FFmpeg* are written in C, while everything else is in Objective-C.

For the GUI and displaying the video the *Cocoa* framework was used. *Cocoa* is the native framework for OS X applications.

The GUI consists of an `NSViewController` and two `NSImageViews`. On the image views we repeatedly update the image when we get new frames from the decoded stream. The frames are in YUV-format when we receive them and we convert them to RGB to be able to show them on the image views. The prototype can be seen in Figure 3.3.

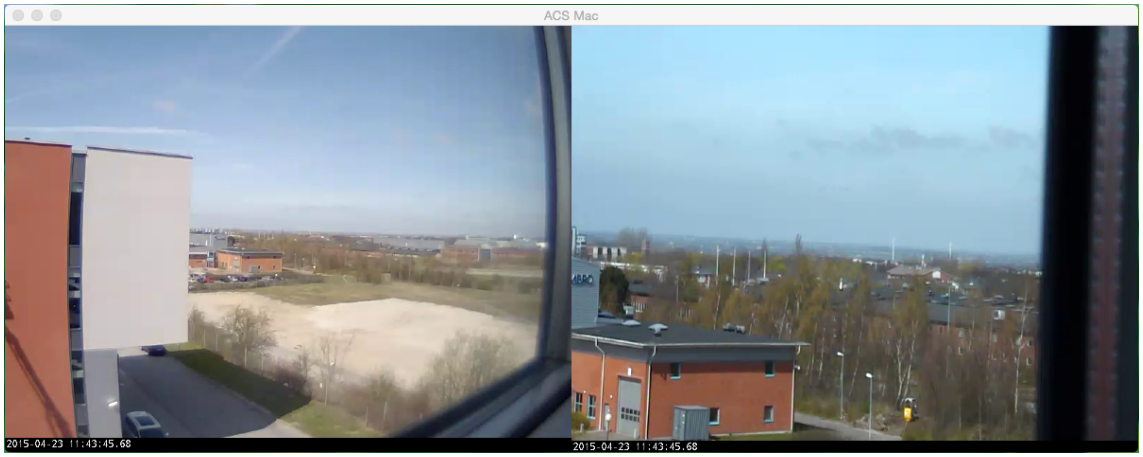

Figure 3.3: The native OS X client.

3.5 Cross-Platform Prototypes

Cross-platform applications that can be run on both Windows and OS X were developed. Here we describe the different prototypes that were developed and why we chose to use the techniques and frameworks that we used. The theory of each framework will first be

described and then the implementations. The different techniques and frameworks considered are described in section 2.9. The prototypes have only been developed and tested on Windows and OS X but when choosing techniques we chose so that the techniques should also work on mobile platforms.

First we decided to try to use the *Xamarin* framework since we already had a working native Windows prototype written in C#. When deciding what other approach to test we chose between Java, Qt and using web technologies. Java since this is a programming language we know well and there existed several video decoding and playback libraries for it. Qt because it is a C++ library meaning it will run on most platforms and also it integrates well with FFmpeg, which we had good experience from when developing the native clients. Web technologies since there exist several frameworks for it and it is an area where there is a lot of development going on. Finally we decided to test *Qt*, mainly because of the ability to create a GUI shared across platforms and the good integration with FFmpeg without bindings.

3.5.1 Xamarin

In this section the Xamarin framework will be described in more detail and also how the prototype was implemented.

Theory

Xamarin is a platform for developing applications for Windows, OS X, iOS, Android and Windows Phone. [42] The company with the same name was created in 2011, although the Mono platform that is used was created already in 2001. [43] Xamarin's main approach for cross-platform desktop application development is to have a single source-tree except for the GUI. This approach is chosen so that the applications look and feel native for each platform. Since the native GUIs are different for each platform they have to be implemented for each platform separately. According to Xamarin this is the best approach because you only have to write the functionality once while still providing a native user experience. By having a single shared codebase that contains the functionality makes implementing new functionality and bug fixes easier since you only have one codebase to maintain, in comparison to one codebase for each platform if you develop native applications for each platform. [44] Figure 3.4 shows an illustration of the code sharing in Xamarin.

Xamarin also includes an API called *Xamarin.Forms* [45] to build native *mobile* applications completely in C# sharing the UI code across platforms. Unfortunately this is not available for desktop application development. Figure 3.5 shows an illustration of the code sharing in Xamarin.Forms.
Xamarin applications are written in C#, which is a native language for the Windows platform. The C# code can be run on multiple platforms in Xamarin thanks to *Mono*, which is an open source implementation of the .NET framework. [44]

Xamarin is available in different packages, all are subscription-based, except the free starter edition, and is paid either monthly or annually. The packages contains different kind of features and support, where the starter edition contains limited functionality and the full fledged enterprise edition contains all functionality and includes support.

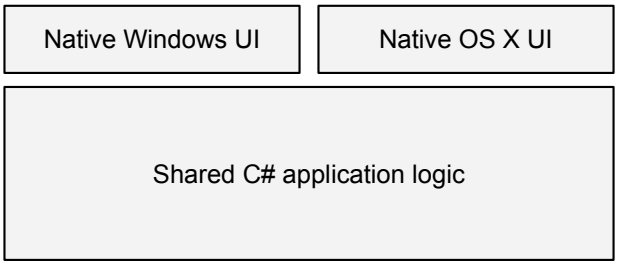

Figure 3.4: Illustration of the code sharing for desktop applications in Xamarin using shared application logic and separate native UIs.

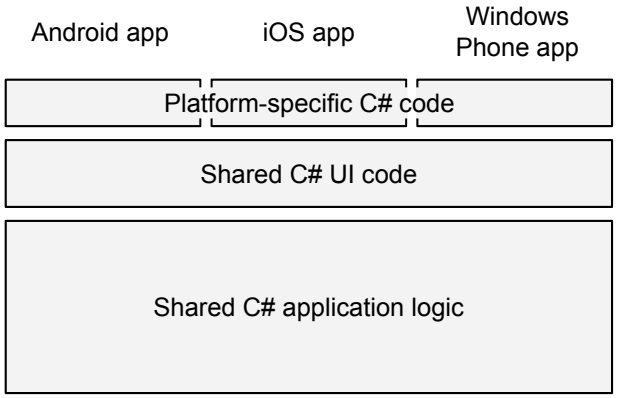

Figure 3.5: Illustration of the code sharing in Xamarin.Forms. [46] At the bottom all applications share the underlying application logic. At the top the applications also share UI code and possibly some platform-specific code. Everything is written in C#. The code finally compiles into separate applications for each platform.

Xamarin applications can be released under the LGPL license without any additional cost, although if you need to release it under a different license there is the possiblitity of buying a commercial license. [47, 37]

Xamarin also provides a cross-platform IDE, named *Xamarin Studio*, to develop the applications with, which can be seen in Figure 3.6. Xamarin Studio is not required on Windows, Visual Studio with a Xamarin-plugin can be used as well.

Implementation

When we started with the Xamarin client we decided to use the Windows client as a starting point, because it was written in the same programming language it ought to be quite similar. We started with extracting the classes that could be shared between the two platforms into a new separate project, so that this project would contain the shared code and then import this into the two platform specific projects. The code that was supposed to be used by

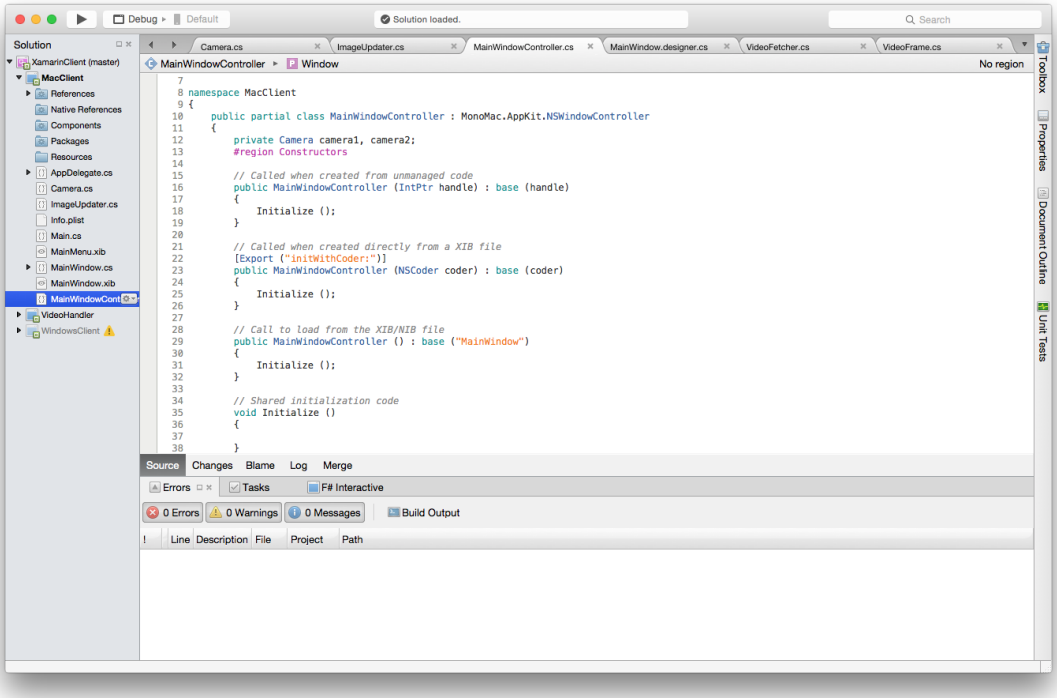

Figure 3.6: The Xamarin Studio IDE included in Xamarin

both platforms was the `VideoFetcher`-class that handles all the communication with the cameras and saves the frames to a buffer.

Some small changes had to be done to the `VideoFetcher`, since the way it was used in the Windows client did not work on OS X. So instead of saving `BitmapSources` in the buffer we use a data-class named `VideoFrame` that contains the video data needed.

Each of the platform-specific projects then uses these two classes to get the video frames from each camera and converts them to the format that the specific platform wants so they can be shown in the GUIs.

The platform-specific code is kept separate from each other, so that the differences in approach can be taken care of without affecting the other platform. This is also where the GUI code is kept, since it is implemented natively for both platforms. Figure 3.7 and Figure 3.8 shows the prototype on Windows and OS X.

There are some problems with the OS X client: the videos it shows are distorted and in black and white, and also the client uses a lot of processing power. The image problems can be seen in Figure 3.8. The problem could have multiple sources, from the immaturity of the Xamarin.Mac framework, to problems with 32-bit compilation of FFmpeg. Another problem is that we have not successfully been able to make the OS X client run as a 64-bit application, which could possibly solve the issue.

Figure 3.7: The Xamarin client running on Windows.

Figure 3.8: The Xamarin client running on OS X.

3.5.2 Qt

This section describes the Qt framework in more detail and how the prototypes that uses this framework are implemented.

Theory

Qt is a C++ cross-platform application framework that was first developed in 1994. [48] The framework can be used to develop applications for a wide range of operating systems such as Windows, OS X, Android and iOS [49]. The framework allows developers to develop applications that work on all these platforms with little to no change to the application codebase. Qt covers many different areas including GUIs, threading, networking and multimedia. The framework is modular and the areas are divided into modules containing cross-platform C++ Qt libraries for handling the different areas.

Qt introduces many extensions to C++, e.g. an object model including a signals and slots mechanism and a so called meta-object compiler (moc), an event system, a property

system and timers. Most of the fundamental extensions are found in the *Qt Core* module. The *QObject* class is located in this module and is the parent class for many Qt classes and forms the foundation for the Qt object model. This class enables many of the Qt extensions listed above. [50]

The *signals and slots* mechanism is a central part of the Qt framework and is used for communication between objects. The mechanism is described as an alternative to using function *callbacks*. Callbacks are often used in GUI programming when writing code that react to events. When using callbacks a pointer to a function is registered at the processing function and when this function is done, the registered function gets notified by calling the registered function. The developers of Qt have seen problems with this approach and describe that it can be unintuitive and that there may be problems with type-correctness of callback arguments. Therefore they have developed the signals and slots mechanism. When a particular event occurs using this mechanism, a signal is sent. Signals can be connected to slots, which basically are functions that get called in response to particular signals. Several signals can be connected to the same slot and a signal can be connected to several slots. In Qt there are many predefined signals and slots, but it is common practice to implement new signals and slots specific to the application. [51]

Signals are declared in the header file of C++ classes under `signals:` and emitted in the class code using the `emit` keyword. Slots are also declared in the header file but under `public slots:` and are defined like ordinary functions. Signals are connected to slots by using one of the `connect` functions of QOjbect. [52] Figure 3.9 shows how a signal is connected to a slot.

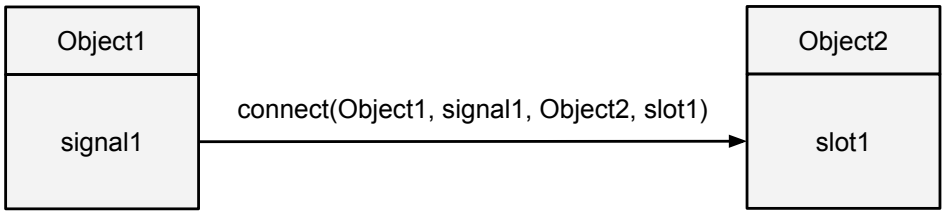

Figure 3.9: Illustration of how a signal is connected to a slot. `signal1` of `Object1` is connected to `slot1` of `Object2` with the `connect` function.

The main reason Qt introduces a meta-object system is to provide the signals and slots mechanism, but the system is also responsible for run-time type information and the property system. In short the meta-object compiler supplies each QObject subclass with code that is necessary for the different mechanisms. [53]

Qt supports developing GUIs in C++ using the *Qt Widgets* module but there is also support for the *QML* language using *Qt Quick* module. QML is a UI specification and declarative programming language. It allows developers to arrange and configure the GUI in `.qml` files using a JSON-like syntax. There is also support for imperative JavaScript expressions for e.g. event logic. [54, 55] By supplying different widget styles, Qt Widgets support giving the application a native look and feel for Windows, OS X and Linux. [56] Qt Quick together with the module *Qt Quick Controls* also, at least, include a style for Android. [57]

Qt supports several compilers such as GNU Compiler Collection (GCC) C++ compiler and the Microsoft Visual C++ (MSVC) compiler. A tool called *qmake* is included in the framework to ease the build process by generating Makefiles from project files (`.pro`). The framework also includes a cross-platform IDE called *Qt Creator*, see Figure 3.10.

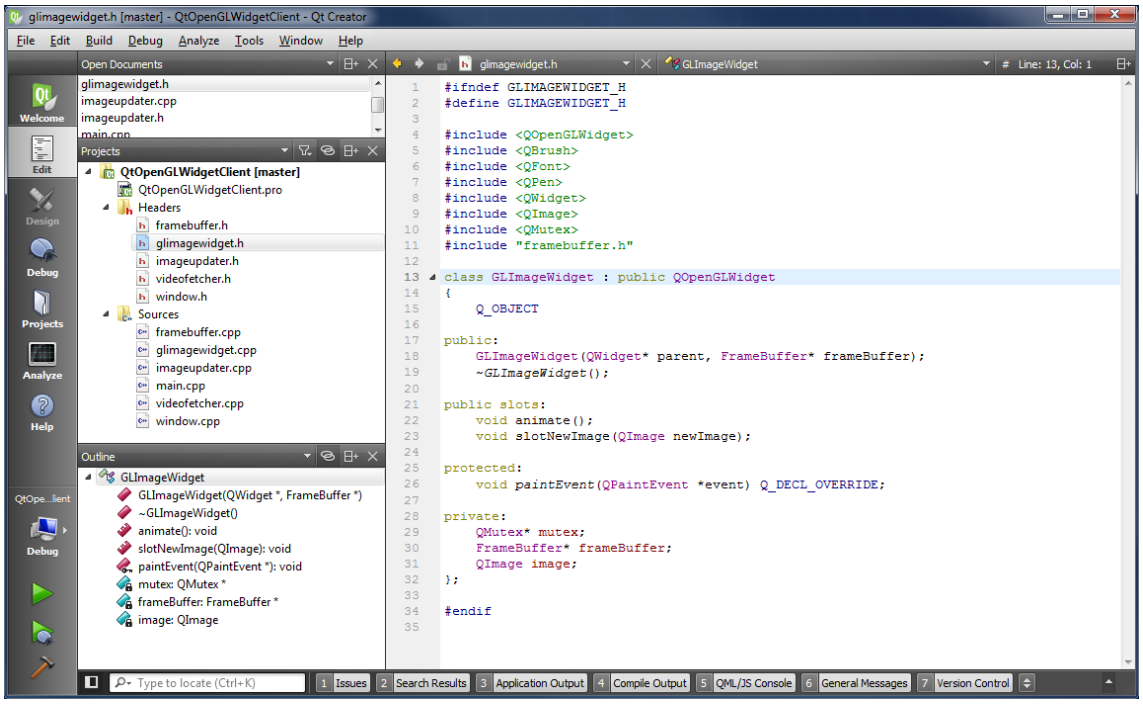

Figure 3.10: The Qt Creator IDE included in the Qt framework.

Qt was first released in 1995 and the latest released version is currently 5.4.1. Qt is available in different editions, ranging from a free limited edition to tailored enterprise editions with commercial license containing more features and support. [58]

Applications using Qt can be released under the GPL or under the LGPL license for free and if the project cannot comply to these licenses a commercial license is included in all the paid versions of Qt.

Qt is used by organizations such as ABB and BlackBerry and is used in applications such as VLC media player and the LaTeX editor Texmaker.

Implementation

Before developing the Qt prototypes existing libraries and solutions were searched for to prevent ”reinventing the wheel”. First we looked at the video and multimedia classes that comes included in Qt. Our investigation and tests showed that these only work for local and remote *files* and not on network streaming protocols. Also they are platform-dependent meaning that they use the multimedia framework of the underlying system such as DirectShow or Media Foundation on Windows and AV Foundation on OS X [59]. This means that different implementations are used on different platforms and thus the video playback may behave differently depending on platform. This is not necessarily something negative since these multimedia frameworks already exist on the platforms, so there is no

need to include a framework for this with the application. Also these frameworks have been developed specifically for the different platforms and may therefore provide good performance and playback experience. Qt also includes classes for accessing cameras but they only support web cameras and mobile device cameras, not network cameras.

Because of the lack of support for our specific situation in the default Qt libraries we searched for third party libraries. We found the libraries *QtAV* [60] which is a multimedia framework based on FFmpeg and *VLC-Qt* [61] which is a library to connect Qt and libvlc libraries. Unfortunately we had no time to test VLC-Qt.

We tried to use QtAV because we had success with using FFmpeg in the native prototypes, but we had problems when using this library. We did manage to get it to work on OS X but not on Windows. There were instructions and examples on how to use the library but the documentation was a bit poor and not always clear. We tried both using precompiled binaries and compiling the library from source as well but we had problems using the library in our projects on Windows. After many attempts to use QtAV we finally had to move on and test an another solution. We decided to use the FFmpeg libraries directly instead, without a Qt library in between.

Two Qt prototypes were developed using the same underlying FFmpeg implementation but with different UI implementations, one using Qt Quick and QML and one using Qt Widget with OpenGL. We developed two Qt prototypes because we wanted to see if there was a difference between the different Qt GUI modules. The FFmpeg implementation consists of a `VideoFetcher` class that is run in a separate thread and fetches video frames and stores them as `QImage` objects in i class called `FrameBuffer`.

Qt Quick Client

In the prototype using Qt Quick we created a GUI using QML and used Image elements to display the video. To update the image elements we used an `QQuickImageProvider` class. Using this class it is possible to set the image element source property to `image://imageProviderName` and dynamically load images. To make the image element update the image though the source address has to change, therefore we had to change the address for each new video frame. Figure 3.11 and Figure 3.12 shows the prototype on Windows and OS X.

Qt Widget OpenGL Client

With this prototype we tried using the desktop style Qt Widget module and OpenGL to implement the GUI. This meant implementing our own custom OpenGL widget, which we call `GLImageWidget`, that inherits from `QOpenGLWidget`. Implementing a custom widget gave us more control over when and how the element is updated. There was no need for an image provider like in the Qt Quick prototype. In this prototype the `FrameBuffer` had blocking functions and a the class `ImageUpdater` was run in a separate thread to fetch new video frames and send them to the widget using signals and slots. Figure 3.13 and Figure 3.14 shows the prototype on Windows and OS X.

Figure 3.11: The Qt Quick client running on Windows.

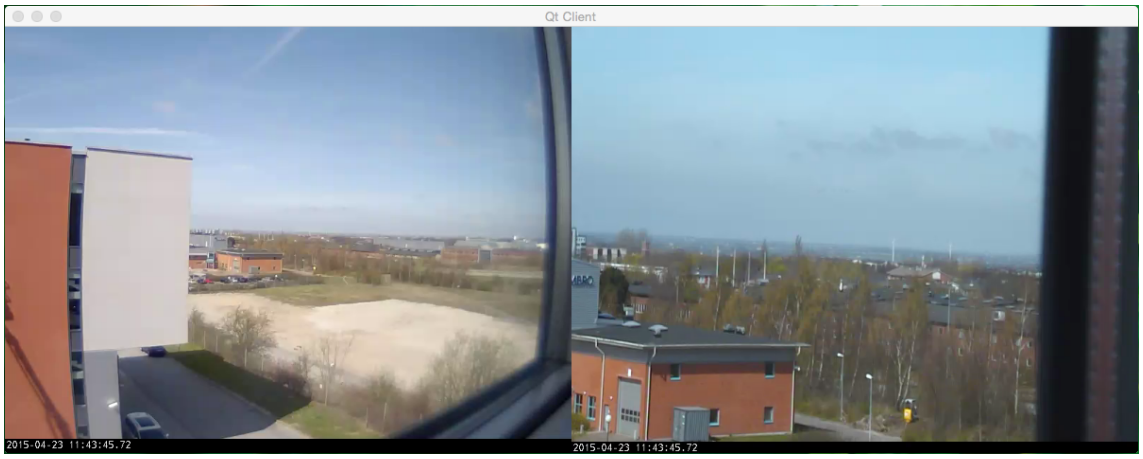

Figure 3.12: The Qt Quick client running on OS X.

User Experience Test

Our prototypes do not feature many UI elements, therefore we tested building some Qt applications featuring e.g. buttons, tabs and text fields to test the user experience of the framework. Qt includes many example applications demonstrating the functionality of the framework, for example the GUI capabilities. For the user experience tests we therefore used two example applications called “Qt Quick Controls - Gallery” and “Basic Layouts Example” to test Qt Quick and Qt Widgets respectively. Figure 3.15 and Figure 3.16 show the applications running on Windows and OS X. The applications looked and felt native on both platforms.

Figure 3.13: The Qt Widget OpenGL client running on Windows.

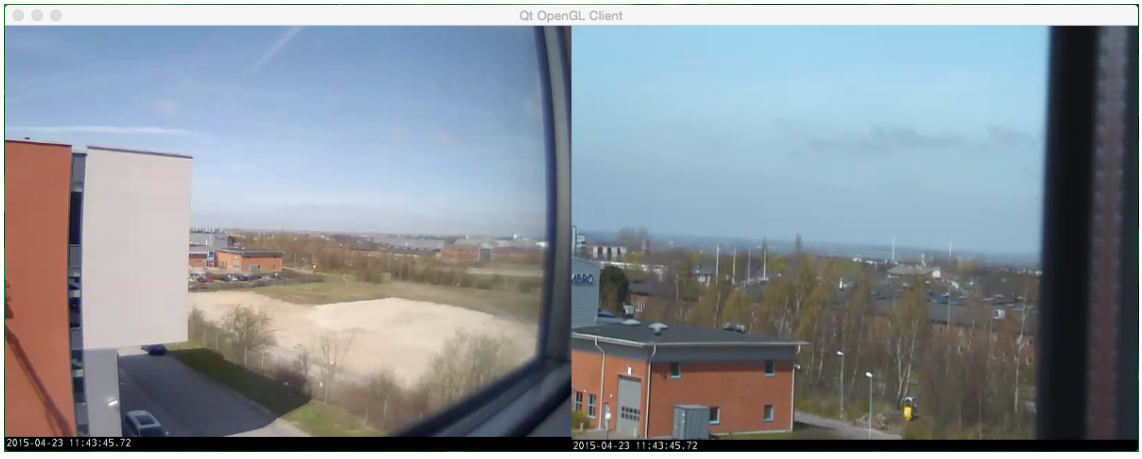

Figure 3.14: The Qt Widget OpenGL client running on OS X.

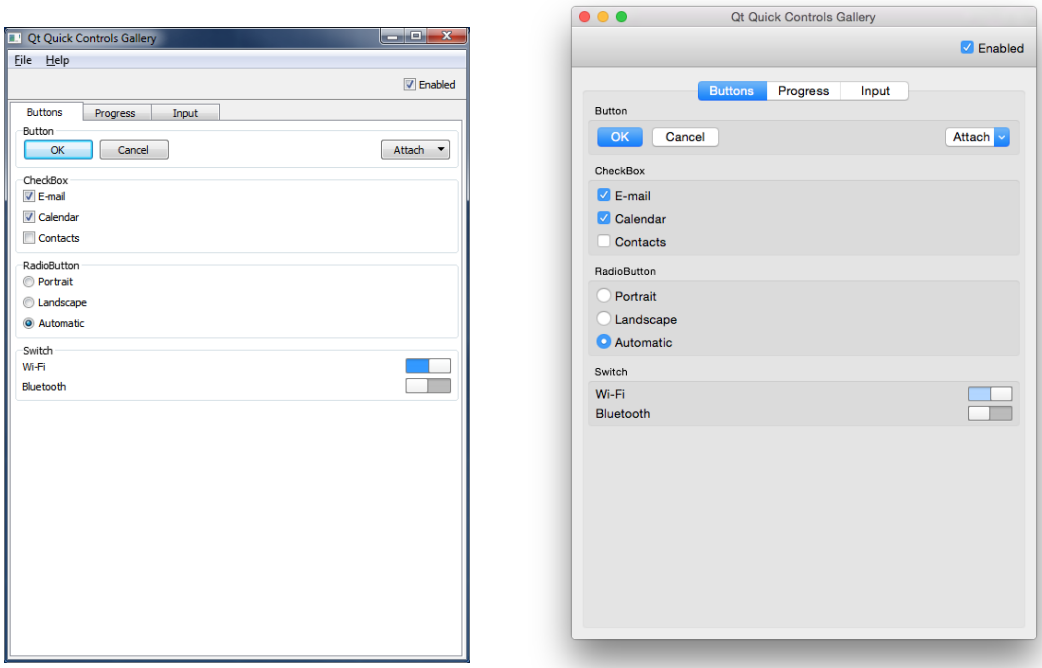

Figure 3.15: The Qt Quick test application running on Windows (left) and OS X (right).

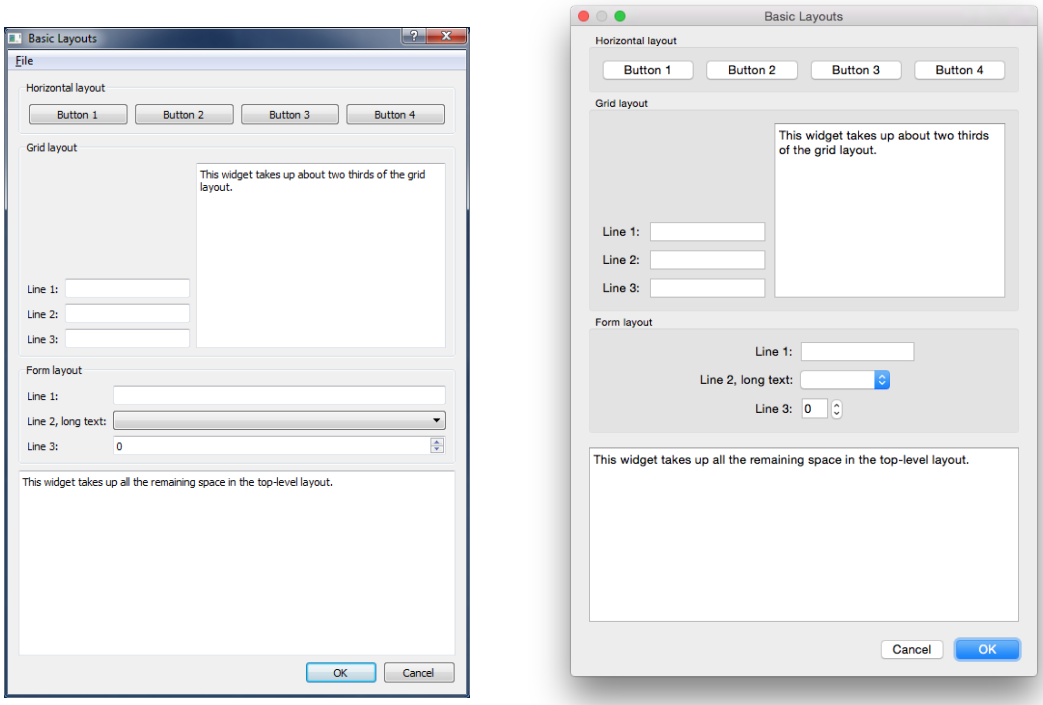

Figure 3.16: The Qt Widgets test application running on Windows (left) and OS X (right).

Chapter 4
Evaluation and Results

The solutions we have developed will be evaluated to specific evaluation criteria. We start by describing the evaluation criteria and experimental setup, then in the following sections we describe the evaluation performed.

4.1 Evaluation Criteria

To evaluate the prototypes the following evaluation criteria were used:

- **Code sharing**: How many Lines Of Code (LOC) that are shared between the platforms for the cross-platform prototypes.
- **User experience**: The look and feel of the cross-platform prototypes will be investigated to see if they fit into the different platforms' UI guidelines.
- **Size of codebase**: How many LOC the codebase consists of.
- **Performance**: How the application utilizes CPU and RAM. The ACS application will be used as reference.
- **Scaling**: How the prototypes performance is affected by adding more cameras.
- **Development time**: How long time we estimate we have spent on developing the prototypes.

When comparing the cross-platform prototypes the focus will be on code sharing in relation to performance and user experience. These criteria are focused on since we have found them to be important when developing a cross-platform video streaming application. One goal with cross-platform development is to share as much code as possible between the platforms, but it must be compared with performance so that the application runs well on all platforms and the user experience is also important so that the application looks and

behaves like a native application would have. The ideal application is considered to be the one where 100% of the code is shared between the platforms, the CPU and RAM usage is low and the application looks and behaves like a native application on all supported platforms.

4.1.1 Lines Of Code (LOC)

Lines Of Code (LOC) or Source Lines Of Code (SLOC) is a software metric for measuring the size of an application. This is done by counting the number of lines in the application source code.

LOC can be measured in different ways, the main ways for measuring is to either measure *logical* or *physical* LOC. Logical LOC means only counting *statements* in the source code while physical LOC means counting *all* lines of the source code files including e.g. comments, empty lines and lines only containing brackets. Thus an application may have different LOC depending on the way the measurement was performed.

In our evaluation we will use *physical* LOC but without counting comments and newlines. We will also make sure the code is written in a similar way for the different prototypes, following the same coding conventions. To get a fair comparison we also make sure that as little imports/usings/includes as possible are used. In this way we show how much code is needed to be written by the developers to produce these kinds of applications.

4.2 Experimental Setup

Here the experimental setup is described including prototype compilation details and computer and camera specifications.

4.2.1 Windows

The prototypes running on Windows were compiled as 64-bit executables and used the 64-bit FFmpeg libraries. The native Windows and Xamarin prototype was compiled using Visual Studio 2013 with .NET 4.5 as target framework. The Qt prototypes were compiled using Qt 5.4.1 and the MSVC 2013 compiler.

The Windows computer had the following specification:

Operating system: Windows 7 Enterprise Service Pack 1 64-bit

CPU: Intel Core i7-4770 3.40 GHz

RAM: 16 GB

GPU: Nvidia GeForce GT 620 2048 MB DDR3, with driver version 350.12

Network: 1 Gbit/s Ethernet

4.2.2 Mac

All prototypes, except the Xamarin OS X prototype, were compiled as 64-bit executables and used the 64-bit FFmpeg libraries. The Xamarin OS X prototype was compiled as a 32-bit executable using Xamarin Studio and was thus also using the 32-bit FFmpeg libraries as we did not manage to get a 64-bit version to work. The Qt prototypes were compiled using Qt 5.4.1 and the clang 6.1.0 compiler.

The Mac computer had the following specification:

Operating system: OS X Yosemite 10.10.3

CPU: Intel Core i5 3.2 GHz

RAM: 2 x 8 GB 1600 MHz DDR3

GPU: Nvidia GeForce GT 755M 1024 MB

Network: 1 Gbit/s Ethernet

4.2.3 Cameras

Two cameras were used, connected to the local network:

AXIS M1143-L Network Camera [62]

Resolution: 800 x 600

Frame rate: 25 FPS

Video compression: H.264 or Motion JPEG

AXIS P5534-E PTZ Dome Network Camera [63]

Resolution: 1280 x 720

Frame rate: 25 FPS

Video compression: H.264 or Motion JPEG

4.3 Limitations

We have not tested the prototypes with cameras that stream video in a higher resolution than 1280 x 720. There is also no handling of time stamps since we show the frames as soon as they arrive, so the video feeds might not be synced with each other.

4.4 Code Sharing

All prototypes use FFmpeg for communication with the cameras and video decoding. Therefore all prototypes share the same FFmpeg libraries but separate FFmpeg binaries are needed for the different platforms. When measuring how large part of the codebase that are shared between the prototypes we will consider FFmpeg to be 100% shared between the platforms since the same calls are made to the library, independent of platform and the library is used by both platforms. Table 4.1 shows the result of this evaluation criteria. A pie chart illustrating the code sharing of the Xamarin Client can be seen in Figure 4.1.

Prototype	Total (LOC)	Shared (LOC)	Code sharing (%)
Xamarin	370 (Windows: 97, OS X: 122)	151	41
Qt Quick	269	269	100
Qt Widget OpenGL	320	320	100

Table 4.1: The amount of code that is shared between the platforms for the different prototypes when not including the FFmpeg codebase.

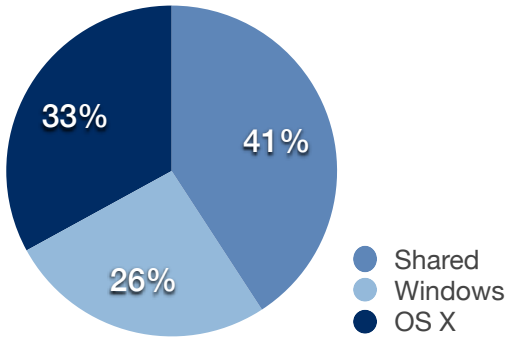

Figure 4.1: Pie chart of the code sharing in our Xamarin Client when not including the FFmpeg codebase.

FFmpeg is a big software project. At the time of writing it consists of 590910 LOC when not including comments and blank lines [64]. If this codebase is added to the calculation of code sharing, the shared code size increases drastically. The code sharing percentage for the Qt clients does not change since also the total code size increases. For the Xamarin client on the other hand there is a significant increase, where the non-shared code more or less becomes negligible. The Xamarin code sharing percentages becomes $(590910 + 151)/(590910 + 370) = 591061/591280 \approx 99.96\%$.

When including FFmpeg in this calculation one must note that our prototypes do not use all functionality of FFmpeg. It is hard to extract the LOC for only the files used by our prototypes since there are very many dependencies between the files of FFmpeg. Even if we extracted the exact LOC that we use though, the code sharing percentage would not

change much since the FFmpeg codebase would still be several times larger than the one of our prototypes.

4.5 User Experience

Since the prototypes we have developed only display video streams and not use other GUI elements such as buttons, text fields or lists it is hard to say something about the user experience of the frameworks that have been used by only looking at the prototypes. The user experience is more or less the same across all the prototypes and the video playback functions well, except for the Xamarin OS X client.

To evaluate the user experience we will instead look at the possibilities of the frameworks. For Qt this will be based on the example applications we have tested. We will also look at the two different approaches used when developing the Xamarin and Qt prototypes respectively. In Xamarin we developed the GUIs separately using native components for the specific platforms and in Qt we used the Qt libraries and built a GUI that worked on both platforms.

Using the approach used for the Xamarin prototype the user experience is good and the application can look and behave like a native application, since the GUI is written using the same components as for a native application. Using Xamarin.Forms it is possible though to have a platform-specific GUI using shared code, but this is currently only supported for mobile platforms and thus nothing we have verified practically.

Because of the different platform styles for Qt Widgets and Qt Quick Controls included in the Qt framework it is possible to achieve a native-looking GUI using shared code. The GUI is not using the actual native components but the Qt buttons, tabs and other GUI elements are styled to *look* like the native equivalent. From what we have seen this works good on Windows and OS X.

4.6 Size of Codebase

Table 4.2 shows the size of the codebase for the different prototypes.

Prototype	**Size (LOC)**
Windows	209
OS X	227
Windows + OS X	436
Xamarin	370
Qt Quick	269
Qt Widget OpenGL	320

Table 4.2: The size of the codebase for the different prototypes.

4.7 Performance

Here the result of the performance tests will be shown. Since our test computers have different hardware specifications it is hard to compare Windows and OS X prototypes directly. Comparing different prototypes running on the same operating system on the other hand is of course possible.

When performing the tests we have streamed video with the highest supported resolution of the cameras (800 x 600 and 1280 x 720 respectively) with a frame rate of 25 FPS from both cameras. In the GUIs of the prototypes the videos have been displayed in the resolution 800 x 600 and 800 x 450 respectively. When performing the tests a still image was filmed to avoid variances in the videos to provide a fair comparison and easier find the average resource utilization.

On Windows the performance was measured using the tool *Process Explorer*. Table 4.3 shows the performance of the different prototypes running on Windows.

Prototype	**CPU (%)**	**RAM (MB)**
ACS	4	344
Windows	5	193
Xamarin	5	200
Qt Quick	9	161
Qt Widget OpenGL	8	146

Table 4.3: The average performance of the different prototypes running on Windows.

On OS X the performance was measured using *Activity Monitor*. Since Activity Monitor represents the maximum CPU as 400% (100% x 4 CPU cores) we have normalized the values by dividing by four. Table 4.4 shows the performance of the different prototypes running on OS X.

Prototype	**CPU (%)**	**RAM (MB)**
OS X	4	62
Xamarin	25	105
Qt Quick	5	100
Qt Widget OpenGL	5	110

Table 4.4: The average performance of the different prototypes running on OS X.

4.8 Scaling

The requirement of the prototypes were to play two video streams simultaneously, although how the prototypes scales with more cameras is also interesting since ACS supports more than two streams. We did the same performance tests as we did in the previous section, although each prototype had 1 to 8 streams that were showed at a resultion of 800 x 600.

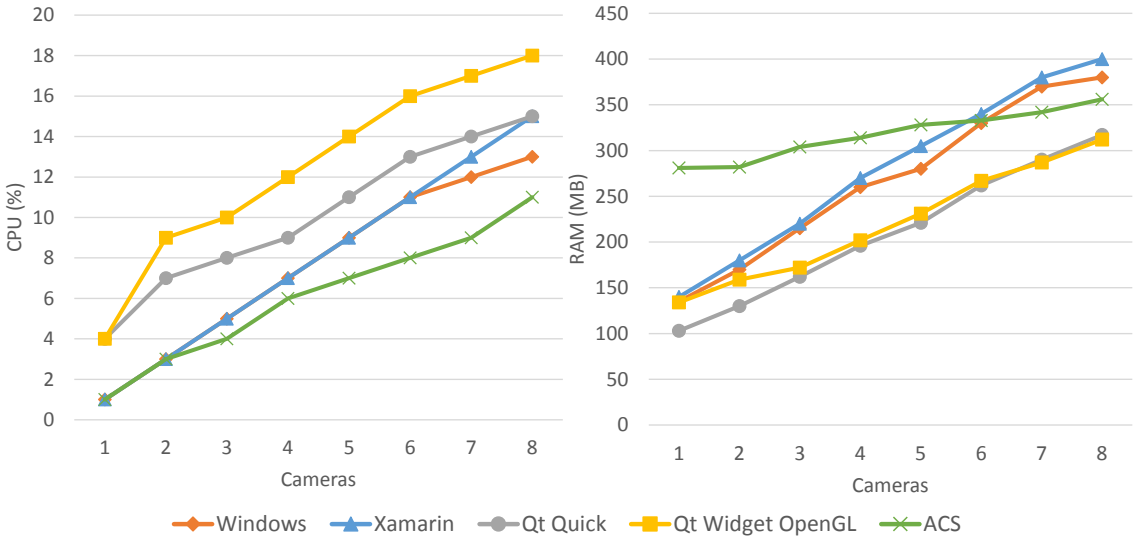

Figure 4.2: Graphs showing the scaling of the Windows clients.

The result for each prototype is that the CPU- and RAM-usage is linearly dependant on the number of video streams.

The only exception to the 800 x 600 resolution is the ACS client, we get 800 x 600 camera feeds from the cameras, although we cannot show the native size of the streams because of layout limitations.

To illustrate the scaling we will show the graphs that shows how each prototype scale. These can be seen in Figure 4.2 for Windows and Figure 4.3 for OS X. These graphs shows the CPU- and RAM-usage, and they are all quite similar and show the same linear scaling for all prototypes. The only exception to this is the Xamarin client on OS X, that has some issues regarding CPU-usage.

The tables containing the data used for the graphs can be seen in Appendix A.

4.9 Development Time

We have tracked our development time for the different prototypes by after each day write down what we have been working on and how long we have worked. In this way we can give a fairly accurate estimation of the development time for each prototype. A source of error for the estimations is that we sometimes have done other work in parallel with the prototype development and not only worked with the prototypes during the day. In the estimation we also include the time for literature study and research done specifically for implementing the prototypes. We estimate the development time in *man-days*. A man-day is an unit equal to the work one person can produce in one day. In our estimation we use a eight hour workday. Table 4.5 shows how much time we estimate we have spent on each prototype.

Prototypes developed later benefit from knowledge and source code from previous prototypes and hence their complete development time is longer. This is for example the case for Xamarin and Qt Widget OpenGL. It is hard to estimate what the complete development time would be for these prototypes if they would have been developed first. We can not

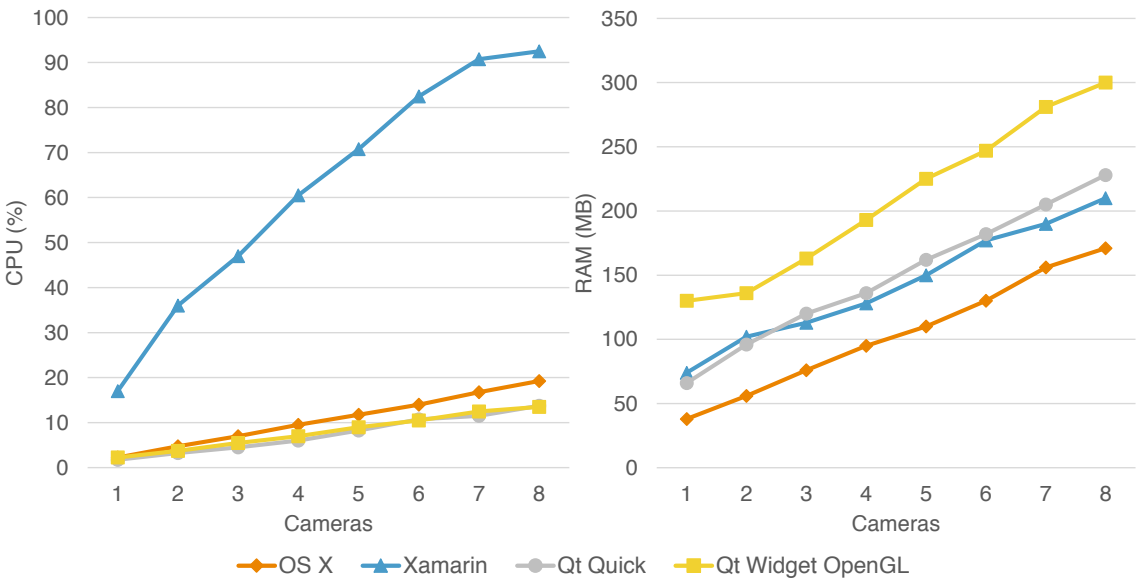

Figure 4.3: Graphs showing the scaling of the OS X clients.

just add the development time of the previous prototypes since we would have made other choices when using other techniques and not all of the work done for these prototypes is relevant.

Prototype	Development time (man-days)
Windows	10
OS X	10
Windows + OS X	20
Xamarin	12
Qt Quick	8
Qt Widget OpenGL	2

Table 4.5: The estimated time spent on developing each prototype.

Chapter 5
Discussion

Our experiences from the work we have carried out investigating cross-platform frameworks and implementing prototypes and the result from the evaluation will be discussed. First we discuss the application development and then the results.

5.1 Application Development

Developing native applications our code became very similar for Windows and OS X, but written in different programming languages. This is because we used FFmpeg on both platforms. We more or less used the same function calls to FFmpeg, since the clients should have the same functionality. This was not something we had expected beforehand when developing native applications, we thought they would be more different. They could also have been much more different if the video for example was decoded using DirectShow on Windows instead, but for simplicity and limit amount of time we chose to use FFmpeg. This shows however that if you use a big enough library, supporting the platforms you are developing for, your code becomes very similar across the platforms. When the code is this similar it is a sign that the code should be shared in some way to prevent having to maintain two versions of the same code.

By developing cross-platform applications we have seen that they in many cases introduce an extra *dependency* to the project, often this dependency can be seen as a extra *layer* applied to the application. The dependency is the framework or tool that is used to make sure that the same code runs on all platforms, for example Xamarin or Qt. Using these techniques enables you to develop cross-platform applications more easily but makes your code dependent on them, meaning that besides following the rules and limitations of the programming language you must also follow the rules and limitations of the framework or tool. This can have a negative impact on the application. An alternative solution could be to write this framework/tool/library by yourself, to have complete control, but then on the other hand you must invest in development and maintenance for this, and it is not guaran-

teed that your solution becomes better than what is already available. In the end one must consider if it is worth introducing this dependency to more easily build cross-platform applications compared to spending time on developing separate native applications. Of course this depends on which platforms the application should support, since there is not always a need for a framework if the programming language includes the necessary features.

Developing both native and cross-platform applications we have seen that there is a difference when introducing changes in the code. On native applications one may have to do the same change on all applications (for each platform) and then test the change on the platform affected. When introducing a change in a cross-platform application it affects all supported platforms and they must all be tested. Often the same change (e.g. adding a feature or correcting a bug) is supposed to be implemented on all platforms and thus the cross-platform approach can save implementation time. Developing cross-platform applications thus implies a different way of working, where you change your code in *one* place and then test on all platforms, while in native development you change in several places and tests them one-and-one when the change is implemented.

From our work with the prototypes we have seen that there is a difference in how much communication that is needed between the developers of the different platforms. When developing the native prototypes there was much communication needed to make sure the clients functioned the same. This for example included how the video should be decoded and how the images were to be displayed in the GUI. In Xamarin it was a comfort to know that the video decoding code was the same for both platforms. With Xamarin we only needed to communicate about the interface to the video-handling part of the application and the GUIs. Finally, with Qt basically no communication was needed between developers for the different platforms, since one developer could implement the whole client by himself. Communication was however needed between the developer and tester of the prototype to make sure that the prototype work correctly across the platforms. The Qt prototypes worked well with no or only minor changes needed, though there were some performance difference between the platforms. When adding support for more than two video streams for performing scaling tests, which mainly was GUI change, we also saw that making changes in the GUI was easier with Qt since there only was one code to change. The change of a Qt client took about half the time of the change of any of the other clients because of this.

Our work has agreed with the literature study in section 2.8. We have seen that the amount of double maintenance for the prototypes has decreased when we have increased the amount of code sharing. We have also seen that the models described work practically, by using a single source tree with Qt and a single source tree but with separate GUIs with Xamarin. With Qt we support different platforms by compiling the application for different platforms. With Xamarin we use the Mono framework to support the platforms. In this way there is no explicit translation needed in neither framework since we instead use programming languages and environments that work across platforms.

5.1.1 Choice of Platforms

We think our choice of platforms to support was good. Windows and OS X were the most important platforms for Axis and they are two very popular desktop operating systems

today. In this way the choice of platforms felt realistic since our work is done for the platforms that a real production application would be developed for, and thus our result is relevant. If we had more time we also would have wanted to support at least one mobile platform to see how well cross-platform development works across desktop and mobile platforms.

Both Xamarin and Qt support mobile platforms such as Android and iOS. Also FFmpeg support these platforms. Therefore it should be possible to extend our cross-platform prototypes to also support mobile platforms. The performance will be worse than for the desktop clients though since mobile phones generally do not have as powerful hardware. Perhaps the video streaming must be done with lower resolution and/or frame rate to compensate for this. It could also be desirable for the mobile applications to have a different set of features than the desktop ones since they are used in another way and context.

5.1.2 Licensing

In terms of licensing and pricing, the two frameworks we chose are quite similar. Both can be used for free, although you will lack features and there are restrictions on how the programs are released compared to the paid version. Xamarin is the most limited free version, which limits the size of the application, while Qt mostly just puts you under the license restriction. They can be released under Lesser General Public License (LGPL), meaning that your own code can be kept private but the frameworks has to be released publicly. If you need to release your application with the framework bundled in the application, you have to buy a commercial license. The rights of the commercial licenses are similar, although the cost is included in the price of Qt, in Xamarin you have to contact Xamarin to get a price so it is an additional cost.

If you are developing an application that can be released under the LGPL license, Qt is the best alternative from a price perspective. Qt is free if your project meets all the requirements of LGPL, which is different from Xamarin where you have to pay without regards to the project, unless it is a small enough project to fit in the starter edition.

FFmpeg that we use for video decoding is released under the LGPL license. Although there are optional modules within FFmpeg that is under the GPLg license, this is something that has to be taken into consideration when using FFmpeg, specially if FFmpeg is used in a commercial application. FFmpeg does not have any commercial license available, which makes any application that uses the library to be released under LGPL or GPL depending on modules.

5.1.3 Introspection

If we look at what worked well and not so well and what we would have done differently if performing the project again we mainly find one area of improvement. We spent time developing our own server in the beginning because we wanted the system to function like ACS do today with a client-server solution. When we decided that it was better to use RTP over RTSP over HTTP as protocol, the server was not important from a client point of view anymore since there basically would be no difference in communicating with a server and directly with the cameras, and the focus of the master thesis project was on client development. The server was thus not used further in the project but from our

experiences of developing it we had gained more knowledge in the relevant protocols and video formats used. If there is one thing we would have done differently if carrying out the project again it would be to not developing a server and instead spend the time on the clients, for example developing more prototypes or features. When looking back we also should have spent less time on the Xamarin prototype since the extra time did not make the client work satisfactory. On the other hand what we think worked well was for example the way we divided the work between us when developing the clients and how we discussed the different solutions. The literature study we performed in the beginning of the project was also good since we had use of the knowledge we learned from it in our later work.

5.2 Prototype Results

Comparing native and cross-platform application development can be hard as it is hard to measure the results of the processes. We decided to use the evaluation criteria code sharing, user experience, size of codebase, performance, scaling and development time. By using these criteria we could evaluate our prototypes to be able to find the advantages and disadvantages of cross-platform development compared to native development.

5.2.1 Lines Of Code (LOC)

The choice of using LOC for measuring the size of the code felt natural but it is a measurement that can be done in several ways. We did our best to count the LOC in the same way for the different prototypes. We used the same software design with only small differences needed for the different frameworks and used the same coding conventions to as great extent as possible. The goal was that the measurement should reflect the amount of code you have to write by hand to develop the applications. This worked well, but of course the result is not definitive but gives a good hint of the size of the prototypes and how much code that is shared.

5.2.2 Problems with Xamarin on OS X

The Xamarin client on OS X has some issues that we have not been able to resolve. The reason why is something we are not sure of, but a part of the high CPU-usage is because the image conversion that has to be done uses a lot of CPU for each image. We have tried to implement this the same way as it has been done in the native OS X application, unfortunately the methods being used in the native application cannot be accessed with Xamarin. It is possible that this will be solved in the future, which then possibly could make the Xamarin client to have similar performance as the other prototypes on OS X.

The issue with the video distortion could be related to either FFmpeg or the same problem with immaturity in the Xamarin.Mac framework. FFmpeg has been compiled in several different ways to try to make it work. The problem could also be because of the image conversion that is performed. We use the same pixel format in the Xamarin client as we do in the other prototypes, so that should not be a problem.

As we have shown there can be problems when developing cross-platform applications, like we have in the Xamarin client. Even though we have taken the same approach and

tried to make it similar to how it is done on the native OS X application, there are problems with the OS X Xamarin client. This shows that even though you are trying to make the applications share as much code as possible between the platforms, some platform specific code could need to be written to fix problems since the platforms do not work the same way. Since it is application development, even if you make it cross-platform or not, problems can and will occur and they will have to be solved for everything to work as expected.

5.2.3 Code Sharing

Regarding the code sharing we could actually share 100% of the code between the platforms when using Qt, while the same number for the Xamarin prototype was 41%. When looking at these numbers one must consider that we had a quite small amount of application logic in our prototype, namely the video fetching functionality. In an application with more features there are often more application logic than GUI code, which means that in a real production application the amount of shared code should be higher. This of course depends on the type of application, and also how many platforms that should be supported.

When considering the size of FFmpeg when calculating the amount of code sharing also the Xamarin client gets near 100% code sharing. This is because the FFmpeg project is much larger than our minimal prototypes. It is interesting to see that the application logic in this case is much larger than the GUI code. It shows that it is possible to develop cross-platform applications with native GUIs while still sharing a major part of the codebase between the platforms.

A realistic production VMS would probably utilize most of FFmpeg's functionality. The application would for example include support for recording, settings, logging, event handling, different camera layouts and a fullscreen mode. We believe this added functionality generally would increase the application logic code size more than the GUI code size. If developing native GUIs for each platform the GUI code of course will become quite large, especially if many platforms are supported, since each platform will have a separate version of the GUI. From our experience we even in this case believe the amount of code sharing will be high for networked video applications since the network and video handling code is large. If we for example estimate the LOC required for just a GUI of a realistic client to 20000, the code sharing for four platforms (Windows, OS X, Android and iOS) with just FFmpeg as back-end would be $590910/(590910 + 4 * 20000) \approx 88\%$. When also adding the application-specific back-end code the code sharing percentage will be even larger.

5.2.4 User Experience

When looking at the user experience evaluation it can be seen that Xamarin has the best experience for the user by providing a truly native GUI. The styling used by Qt also works good but the fact that it is not a truly native GUI makes it a less secure choice since if there are changes in the graphical design of the platform there is a risk that the styling not gets updated or does not include any new features of the update. Using Qt we managed to have 100% code sharing between the platforms though. If one want to use a native GUI while still using C++ or Qt this should be possible as well since there are several ways to make calls between C++ and C# or Objective-C. Using Qt with a native GUI may not be the

optimal way though since Qt mainly is GUI framework, that includes other functionality such as threading. If not using Qt for the GUI another framework or the standard C++ library could be more profitable to use for the application logic instead, depending on the needs of the application being developed.

5.2.5 Size of Codebase

When comparing the size of the codebases of all the prototypes the result was as we had expected. Individually the native clients were the smallest, but for a comparison with the other prototypes to be fair their sizes need to be added together, instead making them the largest prototype. The Xamarin prototype is the second largest since it includes two separate GUIs while still being able to share some code between the platforms. The Qt clients were the smallest, since they have 100% code sharing between the different platforms, even though the Qt Widget OpenGL client was a bit larger than the Qt Quick client because it contains more code for displaying the actual video.

5.2.6 Performance

From the result of the performance tests on Windows it can be seen that ACS has the least CPU usage at 4%, which was expected since the application has been optimized and improved during several years. Our native Windows client and Xamarin client is not far behind though. The Qt clients have a bit higher CPU usage. We have also seen that the CPU usage increases when the window size increases (and thus also the GUI video element size increases) for the Qt clients. This does not happen with the native and Xamarin clients, where instead the CPU usage stays the same. We believe this depends on the underlying implementation for how the GUI is displayed.

Looking at the RAM usage on Windows, ACS uses the most amount but it is a program containing more functionality than the others. The native Windows client and the Xamarin client uses about the same amount and the Qt clients also has similar usage. The difference in usage probably depends on how the different frameworks and programming languages handle memory and minor difference in the prototype implementations.

On OS X the native and Qt prototypes have about the same CPU usage and thus it can be seen as the Qt clients perform better on OS X than on Windows compared to the native prototype. The Xamarin client on the other hand has significantly higher CPU usage than the other prototypes on OS X at 25%. This is because the problems we have had on OS X with Xamarin and FFmpeg and that we could not find a more efficient way of storing the video frames using Xamarin before showing them in the GUI.

The RAM usage of the OS X clients is very similar for all clients except the native client only using 62 MB, which is considerably less than both the other OS X clients and the Windows clients. All OS X clients use less memory than the Windows ones, and this could be because of difference in implementation of the frameworks for the different platforms but also because of more efficient memory management in the OS X operating system.

FFmpeg supports hardware acceleration [65] but in our tests we have used software decoding for all prototypes. The purpose of the hardware acceleration is to offload work from the CPU to the GPU that can perform tasks such as video decoding more efficient. In

this way the application gets better performance and lower CPU usage. To use hardware video decoding in FFmpeg one must compile the library with platform-specific options for the hardware acceleration API that should be used, for example Direct-X Video Acceleration API (DXVA) for Windows and Video Decoding API (VDA) for OS X. On Windows we used software decoding since we used the precompiled FFmpeg binaries from [66] that have been compiled without hardware acceleration support. On OS X we installed FFmpeg via the Homebrew [67] package manager that compiles and installs FFmpeg with VDA-support. Unfortunately we have not been able to get it to work. Instead we used software decoding that worked on all platforms to get a fair comparison.

5.2.7 Scaling

In regards to scaling it scales as we expected, by adding more cameras the performance of the applications changes linearly for each camera that is added. This was an expected result, for each camera that is added the same amount of resources has to be allocated and thus the CPU- and RAM-usage increases by the same amount for each camera.

All our prototypes has little overhead, which is the CPU- and RAM-usage that is not affected by the number of cameras. It is in this regard the ACS client shows that it is not a simple application like our prototypes, since it has a significant overhead. Although the ACS client also scales a bit better than our prototypes, which shows that their video implementation is more effective than the ours.

As can be seen by the OS X graph, the Xamarin client is using significantly more CPU than the other prototypes. It does however scale linearly like the others, it is just that the usage is more extreme than for the others.

We expect that the prototypes keeps scaling linearly for more cameras than the 8 we have tested. This has not been tested since we want all the video streams to be of the same resolution, which makes us run out of screen real estate at 8 video streams.

5.2.8 Development Time

When we study the development time for the prototypes we see that the order of which the prototypes are developed is important. We started by developing the Windows and OS X native prototypes in parallel. These were our first prototypes and we had to create a software design and try to develop a video application without prior experience in the area. For the other prototypes we had a well functioning main design that we could use and had some knowledge in the area, for example we had tested some different approaches for video playback. For the Xamarin and Qt Widget OpenGL clients large parts of the code could be reused as well.

When developing the native Windows client quite much time was spent on testing different solutions for video playback, for example using DirectShow or FFmpeg. Also different ways of displaying video in the GUI was experimented with, for example using the MediaElement of WPF. We also had to learn the C# programming language which we had not used before. The C# concepts and syntax is similar to the ones of Java, which we have good knowledge of, so in this way the process became easier.

The idea was to develop the OS X client using Swift which is Apples new programming language. Although after realizing that this required us to learn another whole new

language when we already knew Objective-C and just had to refresh the knowledge of that, we decided to use Objective-C instead since it is still a supported language to use. After choosing Objective-C and doing some testing with SDL to show the videos, we chose the solution with imageviews and updating the images.

The Xamarin Client started out with the Windows native client as a base, we made some changes to the code base to extract the code that could be shared between the platforms and then created the OS X GUI. The OS X GUI also benefited a lot from the OS X native client, since the GUIs are created the same way. The development time of the Xamarin client was mostly spent on trying to fix the strange behaviour on OS X, the actual changes to the Windows client to make it work on both platforms went fairly easy and without any major problems.

Developing the Qt clients meant refreshing our C++ knowledge and learning the Qt principles. We also spent much testing different ways of displaying the video. Develop the Qt Widget OpenGL client did not take much time since it basically just is a different GUI for the Qt Quick back-end.

The Qt prototypes have the shortest development time, the Xamarin prototype the second shortest and the native clients (which have to be compared together for the comparison to be fair) have the longest. As said before we could use knowledge from the previous prototypes in the later ones, but since the different frameworks and languages used differ quite much there was much studying to do before implementing each prototype. It is interesting though to see that the development time of our Qt cross-platform prototypes is about half that of when we developed the native prototypes.

Chapter 6
Conclusions

By performing both a theoretical and practical analysis we have seen that it is indeed possible to create a cross-platform real-time video streaming application sharing 100% of the codebase. This was done using the Qt C++ framework but we have also seen that there are other development approaches, for example when developing the Xamarin C# client with separate native GUIs.

When comparing the different cross-platform prototypes it can thus be seen that the Qt clients had the highest amount of shared code but their performance was worse on Windows when compared to the other prototypes and ACS. On OS X on the other hand the performance is on par with the native client while there have been problems with the Xamarin client resulting in high CPU usage and a distorted video on the platform.

The Xamarin prototype offers a truly native interface but the styling used by Qt to decorate the application to look like a native application works good as well. In this way both frameworks make it possible to provide a good user experience with applications that look and feel like native.

With the focus of finding the cross-platform application sharing as much code as possible between the platforms while providing a good user experience and performance, in our tests the Qt Quick prototype best meets those criteria.

When comparing cross-platform application development with native application development for video applications we have seen both advantages and disadvantages with developing cross-platform applications. These are the main advantages we have seen:

- It is possible to share source code between the platforms. This means that less maintenance work needs to be carried out since the same functionality does not need to be reimplemented for each platform.

- It is possible to get a smaller codebase. This means that it is easier to learn and navigate the code.

- It is possible to get similar performance as a native application. On OS X our Qt prototypes even had lower CPU-usage than its native counterpart.

- It is possible to develop the application in shorter time. This is because features often only need to be implemented once. For our prototypes the development of the Qt Quick client was about half that of the native clients.

- It can be easier to support new platforms if the cross-platform framework used has support for it. Instead of developing a completely new application, code from the current application can be used.

- It is often only necessary to have knowledge of one programming language. Thus there is no need for specific developer teams per platform. For example we needed to know two different languages for the native clients, while we only needed knowledge of one for the cross-platform ones.

The main disadvantages we have seen with cross-platform development compared to native-development are:

- The user experience may not be as good as for a native application. This does not apply when developing a native GUI.

- The application performance may vary on different platforms.

- There may be platform-specific problems. For example we had problems with our Xamarin client on OS X but not on Windows.

- If a cross-platform framework is used it becomes a dependency for the project. The framework could be limiting and cause bugs and it can be hard to remove this dependency later in the project.

Finally the choice of developing a native or cross-platform application is not trivial. Cross-platform development can be done in many ways and we believe that in the end the choice boils down to personal preference. Developing a cross-platform application it is possible to support many platforms, hopefully fast and easy, but at the same time the framework can be limiting. When choosing a framework it is also important to see if it is a mature project, that is continuously developed and seems to be stable so that the development continues in the future. So our recommendation is to investigate the capabilities of the framework and make sure that it fits ones needs and to weight the advantages and disadvantages listed above for the situation.

If we were to choose a framework for developing a full-scale cross-platform VMS we would choose Qt. We base this choice on our prototypes and results where the Qt clients functioned well. They had the shortest development time, good user experience and performance and 100% code sharing. We generally had a good experience using the framework together with the Qt Creator IDE.

6.1 Future work

Our work can be extended in several ways. We have some different suggestions for future work:

- Develop more prototypes using other frameworks and/or programming languages. For example some kind of web framework, maybe a pure C++ solution or Java solution.
- Develop non-video applications. Video has shown to not be a trivial thing to do, so the native vs cross-platform development comparison might be better if this approach is used.
- Develop larger prototypes with more functionality. This could possibly improve the comparison and might yield a different result.
- Develop prototypes that do not use FFmpeg for video decoding and network communication. All our prototypes use FFmpeg for this and it can be interesting to use some other libraries to see how they behave and if there for example is a change in performance. An alternative could also be to implement this functionality without using libraries.
- Develop prototypes using single source tree emulation (described in section 1.3). It can be interesting to see how this approach differs from the one used in our prototypes.
- Develop a server supporting RTP over RTSP over HTTP. To get a complete client-server system, one could develop this server to make sure that the protocol works as expected.

Bibliography

[1] J. Bishop and N. Horspool. Cross-platform development: Software that lasts. *Computer*, 39(10):26 – 35, 2006.

[2] Michael A. Cusumano and David B. Yoffie. What netscape learned form cross-platform software development. *Communications of the ACM*, 42(10):72 – 78, 1999.

[3] Mozilla Developer Network and individual contributors. NSPR, (Accessed: 2015-05-12). `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR`.

[4] Mozilla Developer Network and individual contributors. About NSPR, (Accessed: 2015-05-12). `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR/About_NSPR`.

[5] Michael Pilone, Gregory Stern, and Brian Solan. Cross-Platform Development: A Difficult Necessity. Technical report, Naval Research Laboratory, Washington, DC 20375-5320, September 2000.

[6] Marcus Lindfeldt and Simon Thörnqvist. Real-time video streaming with HTML5, 2014-08-20.

[7] Tobias Andersson and Erik Jönsson. Software Portable VoIP Client, 2014-12-03.

[8] Encyclopædia Britannica Online. Protocol, 2015 (Accessed: 2015-05-06). `http://academic.eb.com.ludwig.lub.lu.se/EBchecked/topic/410357/protocol`.

[9] H. Schulzrinne and S. Casner and R. Frederick and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Technical Report 3550, July 2003. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7160, 7164.

[10] H. Schulzrinne and A. Rao and R. Lanphier. Real Time Streaming Protocol (RTSP). Technical Report 2326, April 1998.

[11] R. Fielding and J. Gettys and J. Mogul and H. Frystyk and L. Masinter and P. Leach and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Technical Report 2616, June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[12] Apple Computer, Inc. Tunneling QuickTime RTSP and RTP over HTTP, (Accessed: 2015-06-08). http://www.opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreamingServer-412.42/Documentation/RTSP_Over_HTTP.pdf.

[13] Encyclopædia Britannica Online. Data compression, 2015 (Accessed: 2015-05-11). http://academic.eb.com.ludwig.lub.lu.se/EBchecked/topic/152168/data-compression.

[14] Axis Communications AB. Video compression (Electronic), (Accessed: 2015-01-30). http://www.axis.com/products/video/about_networkvideo/compression.htm.

[15] Axis Communications AB. Compression formats (Electronic), (Accessed: 2015-02-03). http://www.axis.com/products/video/about_networkvideo/compression_formats.htm.

[16] Microsoft. About YUV Video (Electronic), (Accessed: 2015-03-24). https://msdn.microsoft.com/en-us/library/windows/desktop/bb530104%28v=vs.85%29.aspx.

[17] Axis Communications AB. What is a network camera? (Electronic), (Accessed: 2015-04-20). http://classic.www.axis.com/products/video/camera/about_cameras/overview.htm.

[18] Axis Communications AB. VAPIX® VERSION 3 (Electronic), (Accessed: 2015-04-20). http://www.axis.com/files/manuals/vapix_video_streaming_52937_en_1307.pdf.

[19] Microsoft. What's a Windows Runtime app? (Electronic), (Accessed: 2015-02-02). https://msdn.microsoft.com/library/windows/apps/dn726767.aspx.

[20] Apple Inc. Introducing Swift (Electronic), (Accessed: 2015-02-02). https://developer.apple.com/swift/.

[21] Mono Project. Mono - Cross platform, open source .NET framework (Electronic), (Accessed: 2015-04-27). http://www.mono-project.com.

[22] Xamarin Inc. Xamarin (Electronic), (Accessed: 2015-04-27). http://xamarin.com.

[23] The Qt Company Ltd. Qt (Electronic), (Accessed: 2015-04-27). http://www.qt.io.

[24] The Apache Software Foundation. Apache Cordova (Electronic), (Accessed: 2015-04-27). https://cordova.apache.org.

[25] Adobe Systems Inc. PhoneGap (Electronic), (Accessed: 2015-04-27). http://phonegap.com.

[26] Adobe Systems Inc. PhoneGap - FAQs (Electronic), (Accessed: 2015-04-27). http://phonegap.com/about/faq.

[27] CoastalForge Inc. TideKit (Electronic), (Accessed: 2015-04-27). https://www.tidekit.com.

[28] TideSDK Team. TideSDK (Electronic), (Accessed: 2015-04-27). http://www.tidesdk.org.

[29] Inc. GitHub. Electron (Electronic), (Accessed: 2015-04-27). http://electron.atom.io.

[30] Facebook Inc. React Native (Electronic), (Accessed: 2015-04-28). https://facebook.github.io/react-native.

[31] Google. J2ObjC (Electronic), (Accessed: 2015-04-28). http://j2objc.org.

[32] JUniversal. JUniversal (Electronic), (Accessed: 2015-04-28). http://juniversal.org.

[33] Haxe Foundation. Haxe - The Cross-platform Toolkit (Electronic), (Accessed: 2015-04-28). http://haxe.org.

[34] Wireshark Foundation. Wireshark (Electronic), (Accessed: 2015-03-23). https://www.wireshark.org.

[35] FFmpeg team. FFmpeg (Electronic), (Accessed: 2015-03-23). https://www.ffmpeg.org.

[36] FFmpeg team. FFmpeg License and Legal Considerations, (Accessed: 2015-05-25). https://www.ffmpeg.org/legal.html.

[37] Free Software Foundation. Gnu lesser general public license, February 1999 (Accessed: 2015-05-25). http://www.gnu.org/licenses/lgpl-2.1.txt.

[38] Microsoft. Process Explorer (Electronic), (Accessed: 2015-05-04). https://technet.microsoft.com/en-us/sysinternals/bb896653.aspx.

[39] Apple. How to use Activity Monitor, 2015-04-17 (Accessed: 2015-05-05). https://support.apple.com/en-gb/HT201464.

[40] Jeremiah Morrill. WPF MediaKit - For webcam, DVD and custom video support in WPF (Electronic), (Accessed: 2015-03-23). https://wpfmediakit.codeplex.com.

[41] Ruslan Balanukhin. FFmpeg.AutoGen (Electronic), (Accessed: 2015-03-23). https://github.com/Ruslan-B/FFmpeg.AutoGen.

[42] Xamarin Inc. Mobile Application Development to Build Apps in C# - Xamarin, (Accessed: 2015-04-27). http://xamarin.com/platform.

[43] Xamarin Inc. We knew there had to be a better way to build mobile apps - Xamarin, (Accessed: 2015-06-09). http://xamarin.com/about.

[44] Xamarin Inc. Key Strategies for Mobile Excellence, (Accessed: 2015-04-27). http://cdn1.xamarin.com/webimages/assets/Xamarin_Whitepaper-Key_Strategies_for_Mobile_Excellence.pdf.

[45] Xamarin Inc. Xamarin.Forms, (Accessed: 2015-05-06). http://xamarin.com/forms.

[46] Xamarin Inc. Mobile Application Development to Build Apps in C# - Xamarin, (Accessed: 2015-04-27). http://xamarin.com/content/images/pages/platform/code-sharing.png.

[47] Xamarin Inc. Commercial Mono Licensing, (Accessed: 2015-05-25). http://xamarin.com/licensing.

[48] The Qt Company Ltd. Qt - About us, (Accessed: 2015-06-09). http://www.qt.io/about-us/.

[49] The Qt Company Ltd. Officially Supported Platforms (Electronic), (Accessed: 2015-04-20). http://doc.qt.io/QtSupportedPlatforms/index.html.

[50] The Qt Company Ltd. QT Documentation - Core Internals (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/topics-core.html.

[51] The Qt Company Ltd. QT Documentation - Signals & Slots (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/signalsandslots.html.

[52] The Qt Company Ltd. QT Documentation - QObject Class (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/qobject.html.

[53] The Qt Company Ltd. QT Documentation - The Meta-Object System (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/metaobjects.html.

[54] The Qt Company Ltd. QT Documentation - Qt Widgets (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/qtwidgets-index.html.

[55] The Qt Company Ltd. QT Documentation - QML Applications (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/qmlapplications.html.

[56] The Qt Company Ltd. QT Documentation - Qt Widget Gallery (Electronic), (Accessed: 2015-04-21). http://doc.qt.io/qt-5/gallery.html.

[57] The Qt Company Ltd. Qt Documentation - Qt Quick Controls - Gallery (Electronic), (Accessed: 2015-04-23). http://doc.qt.io/qt-5/qtquickcontrols-gallery-example.html.

[58] The Qt Company Ltd. Download Qt (Electronic), (Accessed: 2015-04-21). http://www.qt.io/download.

[59] Qt Wiki. Qt Multimedia Backends (Electronic), (Accessed: 2015-04-22). https://wiki.qt.io/Qt_Multimedia_Backends.

[60] Wang Bin. QtAV (Electronic), (Accessed: 2015-04-22). http://www.qtav.org.

[61] Tadej Novak. VLC-Qt Library (Electronic), (Accessed: 2015-04-22). http://vlc-qt.tano.si.

[62] Axis Communications AB. AXIS M1143-L Network Camera (Electronic), (Accessed: 2015-04-29). http://www.axis.com/se/sv/products/axis-m1143-l.

[63] Axis Communications AB. AXIS P5534-E PTZ Dome Network Camera (Electronic), (Accessed: 2015-04-29). http://www.axis.com/se/sv/products/axis-p5534-e.

[64] Black Duck Software, Inc. FFmpeg, (Accessed: 2015-05-20). https://www.openhub.net/p/ffmpeg/analyses/latest/languages_summary.

[65] FFmpeg wiki. HWAccelIntro, (Accessed: 2015-05-25). https://trac.ffmpeg.org/wiki/HWAccelIntro.

[66] Kyle Schwarz. Zeranoe FFmpeg - Builds, (Accessed: 2015-05-25). http://ffmpeg.zeranoe.com/builds/.

[67] Homebrew. Homebrew, (Accessed: 2015-05-25). http://brew.sh/.

Appendices

Appendix A
Scaling

Number of cameras	CPU (%)	RAM (MB)
1	2	38
2	5	56
3	7	76
4	10	95
5	12	110
6	14	130
7	17	156
8	20	171

Table A.1: Scaling table for the native OS X client.

Number of cameras	CPU (%)	RAM (MB)
1	17	74
2	36	102
3	47	113
4	61	128
5	71	150
6	83	177
7	91	190
8	93	210

Table A.2: Scaling table for the Xamarin client on OS X.

Number of cameras	CPU (%)	RAM (MB)
1	2	66
2	3	96
3	5	120
4	6	136
5	8	162
6	11	182
7	12	205
8	14	228

Table A.3: Scaling table for the Qt Quick client on OS X.

Number of cameras	CPU (%)	RAM (MB)
1	2	130
2	4	136
3	6	163
4	7	193
5	9	225
6	11	247
7	13	281
8	14	300

Table A.4: Scaling table for Qt OpenGL client on OS X.

Number of cameras	CPU (%)	RAM (MB)
1	1	135
2	3	170
3	5	215
4	7	260
5	9	280
6	11	330
7	12	370
8	13	380

Table A.5: Scaling table for native Windows client.

Number of cameras	CPU (%)	RAM (MB)
1	1	140
2	3	180
3	5	220
4	7	270
5	9	305
6	11	340
7	13	380
8	15	400

Table A.6: Scaling table for the Xamarin client on Windows.

Number of cameras	CPU (%)	RAM (MB)
1	4	103
2	7	130
3	8	162
4	9	196
5	9	221
6	13	262
7	14	290
8	15	317

Table A.7: Scaling table for the Qt Quick client on Windows.

Number of cameras	CPU (%)	RAM (MB)
1	4	134
2	9	159
3	10	172
4	12	202
5	14	231
6	16	267
7	17	287
8	18	312

Table A.8: Scaling table for the Qt OpenGL client on Windows.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-06-04

EXAMENSARBETE Cross-Platform Video Management Solutions
STUDENTER Thomas Mattsson, Andreas Olsson
HANDLEDARE Per Ganestam (LTH), Fredrik Brozén (AXIS Communications)
EXAMINATOR Mathias Haage (LTH)

Video Management Solutions for Several Platforms

POPULÄRVETENSKAPLIG SAMMANFATTNING **Thomas Mattsson, Andreas Olsson**

Supporting several platforms with an application is becoming more and more important. We have found that it is possible to develop an application sharing 100% of the source code between the platforms and compared different strategies to support several platforms.

Supporting several platforms can be done in many ways when developing an application. The most straightforward way is to develop a native application for each platform separately. A more interesting way is to develop a cross-platform application that can run on all platforms while sharing the source code between the platforms. By developing both native and cross-platform video management applications we have found both pros and cons with the cross-platform approach. We have seen that it is possible to achieve 100% code sharing between Windows and OS X while providing good performance and user experience. We have also seen that there can be platform-specific problems when using cross-platform frameworks that can be hard to track down.

The main benefits of cross-platform development comes from that the source code can be shared between the platforms. Sharing source code means that it is easy to make changes that affect all platforms, for example adding features or fixing bugs. It also means that the developers basically only need to know one programming language. The codebase size gets smaller and there is no need to maintain several platform-specific projects.

The main drawbacks of cross-platform development on the other hand comes from the technique used to make the application cross-platform. This for example is the framework or programming language used. The technique becomes a dependency to the project that can introduce limitations and cause bugs. This dependency can also be costly to remove later. We have seen that the performance can differ between platforms using the same framework as well.

Today we use many different devices every day. For example we use Mac laptops, Windows computers at work and have Androids or iPhones in our pockets. We also expect the applications we use to work on all these platforms. ”How hard can it be?” users and developers may ask themselves, to support several platforms. Developing native applications for each platform often requires a developer team for each platform and much coordination between the teams. It also means maintaining many projects and keeping features and bug fixes in sync. The approach to use cross-platform development and share as much source code as possible between the platforms should mean less maintenance work and synchronization between the platforms.

When developing an application one have to decide what platforms to support and the strategy to use to support them. The choice is often between developing separate native applications and using cross-platform development. We believe that our result can be used as a guide for this decision. Our experiences and result show that there is no one right choice. Instead it depends on the situation and on personal preference. For example developing native applications means having more control while cross-platform development can make the application easier to maintain but often introduces a dependency.