# Summary

# Declarations/Statements

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by giving explicit references. A bibliography is appended.

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed: ...................................................... Date: ......................................................

# Acknowledgements

I would like to express thanks to Dr Neil Harman, whose advice and steady guidance throughout the development of this project was very much appreciated. His patience and talent for constructive criticism was very helpful. I would also like to note his willingness to put aside time for meetings on very short notice, as well as his rapid response to communication, was also appreciated.

I would also like to thank the staff of the Welsh Video Network organisation for their consistent feedback and aid provided during the design, development, and testing stages of the project, and for their help in providing access to their equipment and services. Their feedback often provided a guiding light to give the project direction and purpose.

# Table of Contents

[Summary 1](#_Toc429419948)

[Declarations/Statements 1](#_Toc429419949)

[Acknowledgements 1](#_Toc429419950)

[Table of Contents 2](#_Toc429419951)

[Lists of Figures and Tables 4](#_Toc429419952)

[Abbreviations Used 4](#_Toc429419953)

[Introduction 4](#_Toc429419954)

[Proposal 4](#_Toc429419955)

[Aims and Objectives 4](#_Toc429419956)

[Existing Products 5](#_Toc429419957)

[Specification and Design 7](#_Toc429419958)

[Base Specification 7](#_Toc429419959)

[Standard Specification 7](#_Toc429419960)

[Ideal Specification 8](#_Toc429419961)

[Overall Design 8](#_Toc429419962)

[Device 8](#_Toc429419963)

[Server 9](#_Toc429419964)

[Client 9](#_Toc429419965)

[Implementation 10](#_Toc429419966)

[Hardware Implementation 10](#_Toc429419967)

[Raspberry Pi Software Setup 11](#_Toc429419968)

[Raspberry Pi Implementation 12](#_Toc429419969)

[Server Implementation 14](#_Toc429419970)

[Client Implementation 17](#_Toc429419971)

[Voice Chat Implementation 18](#_Toc429419972)

[Testing 22](#_Toc429419973)

[Methodology 22](#_Toc429419974)

[Results 22](#_Toc429419975)

[Analysis 22](#_Toc429419976)

[Achievements 22](#_Toc429419977)

[Future Recommendations 22](#_Toc429419978)

[Retrospective Analysis 22](#_Toc429419979)

[Conclusion 22](#_Toc429419980)

[List of References 22](#_Toc429419981)

[Bibliography 22](#_Toc429419982)

[Appendices 22](#_Toc429419983)

# Lists of Figures and Tables

# Abbreviations Used

# Introduction

## Proposal

The original proposal for this project was submitted by the Welsh Video Network organisation, which provides video conferencing services via ~100 public sector sites across Wales. Not only do they provide and manage this service, but they also provide technical support to those using the service, usually remotely. The support offered is available for troubleshooting issues with integrated video systems, involving both physical pieces of equipment and wiring issues. The person requesting support is usually not technically skilled themselves, which can cause some communication issues when discussing the problem over the phone with WVN staff. The proposal itself was to develop a piece of equipment that would allow the technical staff to see a live view of what the end-user can see, so that they might more easily understand and diagnose issues. This equipment would ideally be wearable and also allow easy verbal communication, and preferably be relatively low cost to produce.

## Aims and Objectives

After further discussion with the Technical Support team at WVN, clear aims and objectives for the project were set. The solution should:-

* Be able to provide live video feed of user’s location
* Allow two-way easy voice communication
* Require minimal user interaction to activate any on-location device
* Be wearable (and therefore portable) for easy use
* Speed up the troubleshooting process
* Have a desktop client to view and manage remote devices, and also to view information on remote locations
* Be able to remotely control the device to turn it off, or update software etc.
* Be scalable to work across more than 100 locations
* Be relatively low cost to produce and deploy

## Existing Products

There are a variety of existing technologies that can satisfy some of these aims, though in most cases not satisfactorily or not completely.

The most obvious potential solution would be to take advantage of the popularity and power of smart phone devices. Most are easily capable of high quality video recording and transmitting video over the internet, and are inherently capable of verbal communication. All that would be required would be an app that handles the routing of the video call, which could even be an existing app like Skype or Facetime.

One of the main problems of this solution is that it requires any users wanting technical support to own a smart phone compatible with the application, which not only means creating an application that supports a wide range of devices on a variety of operating systems. An application that could range across these devices could be prohibitively expensive due to the licensing fees involved with becoming a developer able to release and distribute applications on each major operating system’s app store. A developer license on Apple’s app store alone costs over £65 a year. It would also mean that the technician would have to walk the user through downloading the application before benefitting from its use, slowing down the troubleshooting process. Actually slowing down the troubleshooting process completely defeats the purpose of the project as a while, so makes it difficult to justify pursuing a course of action like this.

The way around these problems would be to have already available a smart phone stored at each location with the app pre-loaded. This way the application would only need to support a single type of device and a single operating system. This would be potentially expensive, however, and discussions with WVN staff concluded with a preference against the idea of purchasing a smart phone for every location.

Another pre-existing and well established technology is that of the IP Camera. Most come with software already installed that allow remote control of certain functions of the device, with many products even allowing panning and zooming of the camera for a better view, and some even offering voice communication. This offers the prospect of an ideal amount of interaction from the individual requesting assistance (i.e. none). However it would not be possible for it to properly show their viewpoint, nor would it be portable, as these devices are designed as static security cameras that are usually bolted onto a surface and plugged into mains power. Most also don’t come with software or methods to manage more than one of these devices, and due to their finished-product nature it is impossible to modify them to work in such a way that would allow a desktop client to monitor their activity or provide details about their location, unlike a smart phone where an application could have been written to serve this function.

Another solution, while instantly discountable as viable due to the fact development of it ended before it was released as a consumer product, would be Google Glass. As a device it could satisfy nearly every requirement set by the project proposal. It has an embedded camera, in-built speakers and microphone, is inherently wearable, can connect to the internet, and would likely have been able to run apps just like a smart phone could. This could, ultimately, have been the perfect solution that could provide all of the requested functionality. However, disregarding the fact the product was discontinued, the cost of purchasing a Google Glass device for over 100 locations is well beyond the cost limitations set in the proposal.

The solution ultimately decided upon was to use a Raspberry Pi. The Raspberry Pi is effectively a small, lightweight, low-cost computer board that has a variety of accessories available for it, including a specialised camera accessory, and USB and other ports to add generic accessories. A WiFi adapter would allow it to connect to the internet and a headset would allow voice communication. Due to the fact that the Raspberry Pi 2 is powered via a Mini-USB port, it can even be powered using a smart phone power bank, meaning it can be portable and even wearable, as it is a small device. It has low power requirements and so can be powered for a significant length of time by a relatively cheap power pack. Also, the fact that it is effectively a small computer means it is inherently capable of running applications with no extra development license costs, allowing for modification for management by a remote desktop client. If nothing else, a solution built upon a Raspberry Pi would serve as an excellent proof of concept to justify increasing the budget to expand upon this project to develop a potentially better system.

Of course, none of the hardware solutions listed above actually satisfy the criteria regarding a desktop based management client. There are a variety of existing pieces of software designed for similar purposes, such as security camera monitoring software, but again, it is very expensive to purchase or rent software like this for a device that may well be very rarely activated. It is a relatively simple matter to develop a custom desktop application in a cross-platform compatible language (such as Java) to serve as a management client of all the remote devices registered to it. This also allows the creation of a custom database to store location details, or the integration of an existing one.

# Specification and Design

The specifications of the project were broken down into 3 tiers of features; base, standard, and ideal. In this way each feature could be categorised and prioritised, where base was an absolute requirement of the finished product, standard was an intended requirement, and ideal features are useful extras and would only be added if time allowed for their addition. Some features listed in higher tiered specifications (base being the ‘lowest’ and ideal being the ‘highest’) are iterative versions of a more basic version of that feature; for example ‘high frame rate video feed’ is iterative on ‘high quality viewable live feed of user’s viewpoint’.

## Base Specification

* High quality and remotely viewable live video feed of user’s viewpoint
* Desktop-based method of viewing camera feed
* Method of voice communication
* Operable with minimal end user interaction
* Portable
* Be relatively low cost

## Standard Specification

* Scalable to at least 100 separate devices
* High frame rate video feed
* Low latency two way voice communication
* Stand-alone client that identifies location and lists site details
* Wearable device
* Remote control of the device
* Secure

## Ideal Specification

* Self/Remotely updatable software
* Client that can access existing ticketing system database to display location-specific frequent issues and solutions

## Overall Design

To satisfy these specifications an overall sub-system structure was designed:-

SUB SYSTEM DIAGRAM

As can be seen, the whole system was divided into 3 separate sub-systems.

Device

The remote device should be responsible for providing:

* Live high frame rate/quality camera feed
* Portable/Wearable device
* Voice recording/sending and receiving/playback
* Self/Remotely updatable software
* Information about its status

To satisfy these requirements using a Raspberry Pi 2 as a base the following hardware was needed:-

* Camera
* Microphone and speakers (or a headset)
* Battery/power pack
* Wireless internet adapter

The software was also needed to:-

* Expose the video feed of the camera so that it was remotely accessible over the internet
* Record voice and send it to the technician’s client as well as receive audio from the same and play it back
* Inform some other remote application of its status and how it can be accessed

Part of satisfying these requirements is connecting the device to the internet, and forwarding the necessary ports to allow it to actually perform its function. However, beyond the proof-of-concept demonstration, configuring the internet on each of these devices for each location they might be at is well beyond the scope of this project.

Server

To keep the project scalable, and to ensure the entire system isn’t accessible via only one machine, a server application running on a machine with a static IP address is needed. Without this, either every remote device would need a static IP address, or the machine running the client would need a static IP address. This way, remote devices can have IP addresses that change regularly (like is common practice with dynamic IP addresses) and the client can run on any PC. The remote devices can simply ping the server when they’re active, and the server can register their IP address and make it available to the client so that the client can communicate directly with the device.

The server should ultimately be responsible for:-

* Serving as a registrar of all devices
* Providing information about registered and online devices
* Storing IP addresses of online devices
* Routing voice calls
* Block unauthorised access to information

To provide these services a server application would have to be developed, most likely as a Java JAX-WS web service running on a GlassFish server along with a database to store data. It is important to note that deploying to an actual server machine is beyond the scope of this project, as WVN will need to set up their own services to run the system.

Client

The client should serve as the interface through which the technician would view and interact with any devices. It should be responsible for:-

* Overall device management interface
* Displaying device information
* Displaying location-specific information
* Serving as call interface
* Voice recording/sending and receiving/playback
* Providing view of live camera feed
* Limited remote control of devices
* Block unauthorised access to management tools

To provide access to these functions a client would have to be developed, probably as a Java Swing Forms application with a database to store location-specific data. It will also need to integrate the server’s WSDL information so that it can access the methods located on the server.

# Implementation

## Hardware Implementation

A Raspberry Pi 2 Model B along with the official Raspberry Pi Camera Module and Raspberry Pi 2 Model B Case were purchased to serve as the basis of the portable portion of the project that would reside at the site that required technical support. With the Camera Module being more than capable of capturing high resolution and high frame rate video, the option of powering the Raspberry Pi 2 Model B via mini-USB, the low power requirements of the Raspberry Pi as shown by the article referenced above in the literature search, as well as the relatively cheap cost, it seemed the perfect device to build the system upon, even if only for a proof-of-concept if not for a fully realised solution.

Along with this device the following parts were acquired to serve all the required functions:-

* An EasyAcc Ultra 16000mAh power pack was procured to serve as a portable power source
* A Microsoft Lifechat LX-3000 was borrowed from WVN to provide audio support for the voice chat portion of the project, a Dynamode Nano USB WL-700N-RXS Wifi Adapter was bought to allow the Raspberry Pi to connect to Wireless Networks
* A Kingston 16GB Micro SD Card was purchased to act as the Hard Drive of the Raspberry Pi

Raspberry Pi Software Setup

The first step to set up the Raspberry Pi was to select and install an Operating System (OS). An OS called Raspbian (based on the Debian Wheezy Linux distribution) was chosen due to its status as the officially supported OS for Raspberry Pi, and its widely adopted use by Raspberry Pi users. Once the image file for this OS was downloaded, it was written to the SD Card using a tool called Win32DiskImager. Once written, the SD Card simply needed to be inserted into the Raspberry Pi and it could be powered and used.

While a Raspberry Pi is simple to configure for operation via SSH, first-time set up of the Raspberry Pi requires a screen and keyboard to provide the interface to configure SSH, wireless connections, and a handful of other settings. A small portable screen and keyboard were provided by the developer for the duration of the development process to serve this purpose, though these accessories are not required for use in the finished product.

Once powered, the Raspiconfig utility was loaded and settings to enable the Camera Module, to enable SSH, to expand the size of the partition accessible to Raspbian, and to enable WiFi were set using the utility. Once these settings were enabled, the Raspberry Pi was rebooted and connected to a Router (again supplied by the developer) via Ethernet cable to allow for access via SSH. Once powered up again, an attempt was made to configure WiFi settings, which is performed by editing the WPA\_Supplicant.conf file located in /etc/WPA\_Supplicant. The following snippet was appended to the file and saved:-

network={  
ssid="NETWORKNAME"  
psk="PASSWORD"  
}

Adding this snippet should have allowed the Raspberry Pi to connect to the router wirelessly, but this didn’t work. After a brief investigation it was found that the Nano USB WiFi Adapter was faulty, and would quickly overheat and malfunction. After a cursory search, it was found this was a common problem with this adapter. To remedy this, an Edimax EW-7811UN Wireless Nano USB Adapter was purchased as a replacement, chosen largely due to its purported reliability and widely adopted use in other Raspberry Pi projects.

Once the new WiFi adapter was installed, the Raspberry Pi quickly connected to the router, and SSH was possible over the Wireless Local Area Network (WLAN) provided by the router. SSH control was established using Putty, a common tool used for this purpose. A similarly common tool called WinSCP was used to transfer files to and from the Raspberry Pi.

Raspberry Pi Implementation

The first part of the device portion of this project tackled was providing a live video feed over the internet. It was decided due to time constraints that it would be far better to find and use existing software to provide this functionality rather than to design and build programs that likely already existed. After reading various documents (blogs, articles, and forum posts) regarding other Raspberry Pi projects used for IP Camera, WiFi Camera, and Security Camera purposes (similar to those specified by this project), it was found that the most commonly used package was Motion. Motion provided a basic video feed accessible over the internet, and had a large variety of configuration options, including password protection and motion detection, among other features.

Installing packages to the Raspberry Pi is usually as simple as giving a single command to the Raspberry Pi, in this case: sudo apt-get install motion. Upon receiving this instruction, the APT (Advanced Packaging Tool) provided by Raspbian looks up the package in its stored Debian Sources file found in /etc/apt/sources.list (which is updated using apt-get update), and then calculates all dependencies of the package that weren’t already installed, and downloads and installs the package and all its dependencies. However, this requires an internet connection to work, and it was proving impossible to connect the Raspberry Pi to the internet provided by the university, Eduroam. Unfortunately, the developer lived in university accommodation and had no other access to the internet. Instead, an attempt was made to install Motion and its dependencies manually by finding them in the Raspbian Repository Archive, writing them to the SD card, and de-packaging them on the Raspberry Pi. This did not work, and progress was halted until internet access could be established. A full reinstall of Raspbian was then performed to remove the mismatched software and dependencies and to reset any configuration options that were potentially incorrectly changed in efforts to connect the Raspberry Pi to Eduroam.

Internet access was eventually achieved by registering the Raspberry Pi to another internet service provided by the university, Swis-lite, that is usually reserved for game consoles. Once connected, running the command sudo apt-get install motion correctly installed the program to the Raspberry Pi. It was at this time that it was discovered that since the program was designed to be used as Security Camera software, it deliberately delivered only low frame-rate video to remote viewers. Tests showed that this frame rate was unacceptably low (2-3 FPS) and could not provide higher FPS due to the limitations of the software, since it originally designed to only provide an overview of the area live, and to record video for retrieval later only when motion was detected.

A deeper search of software available for the Raspberry Pi revealed a package called RPi Cam Web Interface. Though it utilised Motion to provide some of its functionality, it was designed not as a Security Camera-like application, but was intended to provide a full HD high frame rate camera feed, as well as a comprehensive interface that allowed the user to change the desired frame rate, capture resolution, brightness and contrast settings, and a host of other configuration options, as well as access to some system tools, including buttons to shut down or reboot the Raspberry Pi (this interface would serve well to allow limited remote operation of the Raspberry Pi without full SSH connection). All of this functionality was provided as a web page viewable in any browser and accessible by the IP address of the Raspberry Pi through port 80. This software seemed like it would well satisfy the video requirements of the project.

As it was not an archived Raspbian package, installation of RPi Cam Web Interface was not as simple as that of Motion, and required the source code to be cloned from GitHub and installed using the following set of commands:-

git clone https://github.com/silvanmelchior/RPi\_Cam\_Web\_Interface.git

cd RPi\_Cam\_Web\_Interface

chmod u+x RPi\_Cam\_Web\_Interface\_Installer.sh

./RPi\_Cam\_Web\_Interface\_Installer.sh install

At this point an installer menu provided access to installing the program itself as well as a pre-configured Apache client to provide the web capabilities.

It was found that the application would automatically record video whenever it detected any motion. This was undesirable as it could quickly and inadvertently fill up all of the available memory space on the SD card. Since there was no option to directly deactivate this, the motion capture settings were tweaked so that motion had to be detected for over 10000 video frames before triggering recording, and ‘motion’ was changed to be defined as a difference of over 10000 (arbitrary units) between frames. This effectively made it impossible for ‘motion’ to be detected and impossible for it to last long enough to trigger recording if it was.

From this point on, RPi Cam Web Interface required no further configuration. It provided a sufficiently high quality video stream that was expandable to see more detail, as well as all the interaction utilities advertised.

Next, a lightweight application called RemotePiManager that was developed and written alongside the Camera Registrar server was installed to the Raspberry Pi stored in /RemotePiManager. This program would access a file called details.conf stored in the same directory and register itself to the server with the location specified on the first line of details.conf, and receive and store a unique ID. Once registered, the application then pings the Registrar every second to inform it that the Raspberry Pi is online and to allowed the Registrar to register the IP address by which the device could be accessed.

Once RemotePiManager was tested and was working as intended, focus shifted to the voice chat requirements. Since this function is common to both the remote device and the local client, the implementation of Voice Chat has been given its own section on page XX

## Server Implementation

The server was supposed to function as a Registrar, to allow the remote devices whose IP address may well change over time a static address to ping to. Upon receiving the pings, the server should store the IP addresses of the devices and serve them to the Camera Viewer client.

The server was implemented in Java as a Web Service using JWS libraries. The Netbeans IDE was used for development, along with a GlassFish server for testing deployment. The server also had a Database designed to store a list of registered devices.

The server originally offered methods to register and retire devices, and to return lists of the devices currently online, and lists of all registered devices. To do this it utilised three classes.

The main RPITechSuppRegistrar class was the port of call for devices and clients, and contained the methods for registering and retiring devices, and for returning lists of devices.

RPICam served as an Object to store details of individual devices, and would be used in both the server and the client. RPICam stores the location and ID number of a given device, as well as its IP address and a Date variable containing the precise time the last ping for the device was received. It also offered timeout methods to check whether the time since the last received ping was greater than the timeout parameter passed in its constructor. The RPITechSuppRegistrar class would store instances of RPICam in an ArrayList to keep track of the online devices. RPICam also offered methods for converting its contents into Strings for easy transfer, and contained constructors that would accept those strings.

DBManager was the class that managed the database, and offered methods to add and remove entries, to search for specific entries based either on an ID or a location String, and for returning the entire database in String format that was parse-able for the RPICam class.

As the managing class, RPITechSuppRegistrar was where most of the processes would occur.

When registering, a device would send its location to the server, and the server would then check to see if a device was already registered at that location. Only one device was allowed to be at each location, and if another was already registered at the location then the server sends back an error code. If the location was open, the server would assign the device a unique ID number (and send this pack to the device) and register it in the database. The retire() method offers the reverse of this, and removes devices from the database.

Once registered, the device can then call the pingAlive() method to tell the server that it’s online. When receiving a ping, the server was designed to use the ID passed to it to grab the device from the database, and extract the IP address of the device from the MessageContext of the HTTP request. Using this information it would then construct a new instance of RPICam with the IP address and add it to the list of online devices (onlinePis) along with updating the lastPing member to the current time. If the device had already pinged and had not timed out and been pruned from the list, then the entry in the online device list would be updated with the new lastPing time.

A number of methods were also designed to be used by the client to display information to the Technician using it. The getOnlineList() method could be used to retrieve the list of all the devices online. Calling this method causes the server to run through the list of online devices, checking all of them to see if they have timed out and pruning them from the list if they have, before sending the list back to the client as a parse-able String. This method and the pingAlive() method both synchronise on onlinePis to prevent any errors or concurrent accessing of the list.

The method getRegisteredPiList() allows access to the list of all devices registered in the database. Much like getOnlineList(), it returns the list in a parse-able String format.

At this point the server was deployed until the client had been implemented and testing of the systems could be carried out. The server saw very little modification beyond debugging after this point.

One of the additions was added to support passing of IP addresses to devices so that a two-way VOIP call could be established between the Technician’s machine and a remote device, but was deprecated in favour of the device waiting to receive a UDP packet and extracting the IP address to return packets to from it. These methods and their related member variables remain implemented in the final version so that voice communication can be expanded to a more complex program in the future that may require methods like these.

The only other addition was to add password support. For a user to access any functions from the client a password must be entered. The password should be embedded into the HTTP request, and would then be extracted by the server and checked each time a method is called. A login() method was added so that controls in the client could be locked until receiving a confirmation of the password entered from the server. Beyond this, if calls are made to other server methods don’t contain the correct password the server returns PASSWORDNOTFOUND strings to the requester.

## Client Implementation

The GUI was the first part of the client that was actually designed and implemented. The GUI was designed using the built-in Netbeans form designer. The following controls were added:-

* A list view to display the list of devices (online and registered)
* A combobox to allow the user to switch between online devices view and all registered devices view
* Buttons to open camera views, to start a call, to copy the IP address of the selected device, and to refresh the list
* A panel to display lists of the past user issues and equipment at the selected location

The GUI itself is contained in its own class, TechSuppGUI, as MVC principles were born in mind while designing the application. This class not only handles all the required events thrown by the GUI controls, but offered a few methods to allow the controller class to enable and disable parts of the UI, as well as update them with new information as needed. A method to return the index of which device was selected from the device list was also added, as the index of the device list corresponds to the index of the same device stored in an ArrayList in the controller class. Each of these GUIs were also responsible for validating user input before sending it to the controller class.

The controller class, RPITechSuppCamViewer, was designed to offer a variety of functions and acts as the access point for the server and the database that stores location specific issues and equipment. It stored a local list of devices that changed depending on whether the user was viewing online or all registered devices, as well as the web service and GUI references. The methods it offered include:-

* A refresh method to update the device list with devices dependant on the selected view
* A method to open a webpage to view the camera feed of a device
* A method to copy the stored IP address of an online camera

This list was later expanded more features were added. The first of these features was the ability to perform calls to devices (though as was stated previously these calls only functioned one-way). Methods for initiating and ending calls as well as dealing with call timeouts were added to RPITechSuppCamViewer, and related UI objects were added to the GUI.

The next addition was that of the location database. It was originally intended that an existing ticketing database filled by WVN’s own ticketing system would be integrated and the panel for view location details would simply pull data from this database, but unfortunately WVN was unable to provide a copy of the database in time for it to be integrated into the application before development halted. Instead, a custom database was designed and a class called LocationDBManager was implemented to access it, as well as methods added to RPITechSuppCamViewer to serve as an interface between the GUI and LocationDBManager.

Buttons were added to the panel to allow a user to add and remove issues and equipment from the database. The buttons to add entries to the database were designed to open instances of the AddEquipment and AddIssue form classes which actually provide the interface for typing and adding new database entries as well as serving to validate user input before submitting it for entry into or removal from the database. The reference to these instances are stored in the TechGUI class and are simply set visible and invisible when needed instead of new instances being created and destroyed.

Lastly, the requirement of a password was added to access the user interface to add some security. This was implemented as a new form class called LoginForm that appears before the full GUI is loaded. The user has to enter the correct password which is then sent to the server (via the login() method of the RPITechSuppCamViewer class which adds the password to the header of every HTTP request sent to the server in future) for verification. If the server confirms the password, then the full GUI is loaded.

## Voice Chat Implementation

During the course of development, and during the research for original specifications document, it was expected that a VOIP (Voice Over Internet Protocol) program would be installed onto the Raspberry Pi and integrated into the client running on the Technician’s computer. Some solutions offered the ability to complete calls using the IP address of each caller, which was ideal. Later on when it came to actually implement voice chat, further inspection was carried out and found that most VOIP solutions required installing several extra applications, such as a soft-phone application, and an IP PBX, among others, none of which could be properly integrated into the client. Even using Java SIP libraries to create a client that could be integrated properly seemed unnecessarily over complicated, since these libraries were designed to handle the handshaking between devices to determine what media and devices they had in common to communicate with each. This seemed especially so when considering the fact the developer had direct control over both applications and could simply decide in advance exactly how they would communicate. In hindsight, it is also likely that while using Java SIP libraries the same problem that was found when pursuing the decided upon course of action would have occurred.

Ultimately it was decided a very simple class could be designed and added to both the client and device that would allow very simple two-way audio communication by recording small frames of audio and sending them using UDP packets and then playing them back on the receiving end. Each caller would run two threads, one that would record audio to a Byte Array and send it to the other caller, and another that would receive incoming Byte Arrays and play them back. At its core, the class that managed these calls, called TechCall, relied on Java DataLines. The class would request a DataLine from the system’s audio mixer that suited a specific audio format (in this case, 8KHz, 16-bit, mono-channel, signed, and little-Endian). For recording, a TargetDataLine was required, and a SourceDataLine was required for playing back audio. Both of these objects natively wrote and read Byte Arrays, perfect for use with standard Java UDP socket libraries.

There were two constructors for this class. One of these required no arguments to be passed to it, and would be used by the Raspberry Pi at startup. The other constructor required an IP address passed as a String. When the startCall() method is called it checks to see if an IP address was actually set by the constructor, and if it was, it would initiate the call to a given IP address. If no IP address was set by the constructor the class would wait for an incoming UDP packet before starting the playback and recording threads.

During testing, it was found that a problem arose when setting up the DataLines on the Raspberry Pi. The following snippet of code would throw a LineUnavailableException once it reached the indicated line:-

private AudioFormat format = new AudioFormat(8000, 16, 1, true, false); DataLine.Info targetInfo = new DataLine.Info(TargetDataLine.class,

format);

targetLine = (TargetDataLine) AudioSystem.getLine(targetInfo); //error

targetLine.open(format);

The targetLine object is used to record audio, and there are no problems with this when running on Windows machines or when running on the Linux machines available in the Swansea University Linux Lab. This problem is present only on the Raspberry Pi, regardless of whether it is running Oracle JDK or OpenJDK. The problem is believed to be related to the interaction between Java’s AudioSystem libraries and the Raspberry Pi’s hardware, or perhaps with the Alsa audio mixer that managed Raspberry Pi audio in Raspbian.

Considering the popularity of Debian, and of Raspbian, and the fact that the libraries used to implement this TechCall class are included in the standard JDK, the idea that this wouldn’t work was completely unforeseen. Research carried out after the fact reveal that it is a fairly common issue that simply has no solution available to developers, however due to the nature of the problem and the fact that it was completely unforeseen meant that no research into whether a library included in the standard Java libraries might not work was carried out.

As stated earlier, using Java SIP libraries to implement VOIP would likely have encountered the same problem when trying to access any microphone connected to the Raspberry Pi.

There are potential solutions to this issue, the first of which would be to re-write the program in a different language (e.g. C# or more likely C++). However, a similar issue may well have been encountered in another language, and since it would require re-writing large portions of the project, it was deemed that this would simply take too long. It would perhaps be possible to only write a small program in a different language that would only record and send audio, then start it as a separate process from RemotePiManager and pass it the IP address to send the packets to, but this would likely be difficult to manage, and might well require a reboot of the device to restart a dropped or ended call.

Suffice it to say, the Raspberry Pi would not allow recording of microphone input, but it could play back the sound frames it received. This allowed one-way communication from the Technician to the end-user, which would be sufficient for the Technician to relay instructions, but doesn’t really allow the end-user to ask questions or explain anything.

In reality, the process of using this device would usually begin with the end-user calling Tech Support on the phone for help with their problem, and only after exhausted other options would the Technician request that the user activate the camera and put it on. Ultimately, it’s likely that the Technician and end-user could and would maintain the phone call they were having, the exception to this being if the original call had been made on a wired phone. If they are already on the phone with each other, they could simply continue the phone call while the end user just wears the Raspberry Pi without a headset. Failing this, one-way communication is still possible using the Raspberry Pi and computer client, but the end-user would have to communicate with hand gestures, or by writing down more complicated questions and showing them to the camera, a cumbersome process but still a functional one.

# Testing

## Methodology

There were 3 practical tests carried out to evaluate the suitability of the program. One was designed to assess the UI and its ease of use. The second was designed to assess how well the overall system worked between a user and a technician. Another test was also performed to test the power requirements of the device and the stability of the battery pack.

UI Ease of Use

This was a very simple test. Anonymous users who had never seen the interface before and had no technical support experience were placed in front of the UI for the first time and asked to perform a short series of tasks relating to the UI. The time between being asked the question and completion of the task was then measured. There were 2 categories of tasks, one where the user would simply point out where controls were, and another where they would actually have to carry out an operation. For the first category a special version of the UI (see Figure xxx) was created in which all controls were enabled at all times, as some controls are not enabled (and therefore greyed out) unless a certain process is followed, and this may confuse the user in tests, while in practical use this would not pose an issue. For the latter category a live version of the client was used. The following tasks were set:-

1. Point out the button to open a camera view
2. Point out the button retire a device
3. Point out the button to start a call
4. Point out the button to end a call
5. Point out the button to refresh the device list
6. Enter the password and log into the client
7. Switch the camera view to ‘all registered’ mode
8. Select the device at Cardiff
9. Add a piece of equipment called ‘computer’ to the list for this device
10. Remove the piece of equipment from the list for this device

Performing physical tasks

In this test 2 users worked together to find an object in a room they were both familiar with. One user (hereafter referred to as the technician) sat in front of a computer running the client the Technician would have access to, and had already been familiarised with. This user would also have been shown how the device works. The other user (hereafter referred to as the remote-user) was to stand in the room with the device next to them, but they did not know how the device worked. The technician was the same individual throughout the tests, while the remote-user changed for each test, to simulate the technician who would be used to the process and a remote-user who wouldn’t know what was happening.

The technician knew what the object was and what it looked like, but was not allowed to tell the remote-user what the object was (they could however tell them what it looked like). The remote-user was to call the technician, and the technician would walk them through using the device, and then finding the object. Since only one-way communication was possible, the technician was given the choice of continuing to use the phone they had started the conversation with, or whether to switch to the client’s own voice communication.

Power Consumption and Supply Reliability

This was a simple test to assess the viability of using a battery pack to power the Raspberry Pi as it rendered video, broadcasted it to the internet, and ran the required software. Every 10 minutes for 4 hours the Voltage and Current being supplied to the device was measured using a small USB Voltage and Current meter, and the power supplied (in Watts) calculated. This was designed to give a reliable idea of whether the device could operate in a stable manner over a period of time long enough to complete a troubleshooting process. 4 hours would be more than enough time, and if the device functioned reliably and the power consumption was stable during this time, then it would be known that the battery pack was a viable power supply.

## Results and Analysis

UI Ease of Use

Performing physical tasks

Power Consumption and Supply Reliability

The raw results of the tests are as follows:-

|  |  |  |  |
| --- | --- | --- | --- |
| Time | Voltage (V) | Current (A) | Power (W) |
| 13:00 | 5.03 | 0.43 | 2.16 |
| 13:10 | 5.04 | 0.40 | 2.01 |
| 13:20 | 5.04 | 0.39 | 1.96 |
| 13:30 | 5.04 | 0.39 | 1.96 |
| 13:40 | 5.04 | 0.38 | 1.91 |
| 13:50 | 5.04 | 0.39 | 1.96 |
| 14:00 | 5.04 | 0.39 | 1.96 |
| 14:10 | 5.04 | 0.42 | 2.11 |
| 14:20 | 5.04 | 0.35 | 1.76 |
| 14:30 | 5.04 | 0.39 | 1.96 |
| 14:40 | 5.04 | 0.35 | 1.76 |
| 14:50 | 5.04 | 0.36 | 1.81 |
| 15:00 | 5.04 | 0.39 | 1.96 |
| 15:10 | 5.04 | 0.42 | 2.11 |
| 15:20 | 5.04 | 0.41 | 2.06 |
| 15:30 | 5.04 | 0.38 | 1.91 |
| 15:40 | 5.04 | 0.40 | 2.01 |
| 15:50 | 5.04 | 0.40 | 2.016 |
| 16:00 | 5.04 | 0.44 | 2.2176 |
| 16:10 | 5.04 | 0.39 | 1.9656 |
| 16:20 | 5.04 | 0.44 | 2.2176 |
| 16:30 | 5.03 | 0.46 | 2.3138 |
| 16:40 | 5.04 | 0.48 | 2.4192 |
| 16:50 | 5.04 | 0.49 | 2.4696 |
| 17:00 | 5.04 | 0.45 | 2.268 |

The results are also displayed here as a line graph to display the change over time of the Voltage, Current, and Power consumption.

The main issue this test anticipated to find was that over time the power being supplied by the battery would drop, but as can be seen from these results there was very little variation in power draw during the 4 hour period tested.

The largest variation showed an increase in power draw towards the end of the test, indicating that the battery pack was still well able to meet the higher demands of the device even after 4 hours of testing. We can see due the fact that the small variations evident are both increases and decreases that changes in power draw are more likely due to processes taking place within the Raspberry Pi rather than because of the battery pack’s ability to provide power.

This is further supported by the fact that the current draw was the varying factor, while the voltage remained nearly constant. If the issue was with the battery, we would have observed a drop in the voltage rather than the current draw, since the current is dependent on the load being supplied. However, we can see that the voltage only varies by 0.01 Volts during the entire duration of the test, while it is the current draw (which is dependent on the device) that is changing over time.

# Achievements

# Future Recommendations

# Retrospective Analysis

# Conclusion

# List of References

# Bibliography

# Appendices

Red = references required

Blue = add to abbreviations section