# An investigation into a mobile solution for controlling audio networks

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Sascha Zeisberger

*Grahamstown, South Africa*
November, 2010

**Abstract**

Digital audio networks have become a popular solution to the problem of distributing audio. They provide the ability to manage devices at a software level and allow audio signals to be routed between devices. The administration of these networks and their devices becomes increasingly difficult when they span large distances since most control applications are built on static workstations. This project proposes the development of a mobile application that will control an audio network and give a sound engineer the capability to have this control at any physical location.

The initial stages of this project concentrated on porting an existing desktop application to the iPhone OS. It was found that the ported application did not provide a satisfactory means of control over an audio network, since the application's logic was more suited to the desktop environment.

Various connectivity approaches were analysed so that an improved prototype could be built. The analysis resulted in the design of a grid-based end point to end point patchbay solution. The patchbay was implemented and a final analysis of this prototype showed that the design did provide a satisfactory means to control an audio network. However more work was required in the underlying logic to provide subnet to subnet routing.

**ACM Computing Classification System Classification**

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2010):
**C.1.3** [Other Architecture Styles]: Cellular Architecture
**C.2.2** [Network Protocols]: Routing protocols
**H.5.1** [Multimedia Information Systems]: Audio input/outpu
**General-Terms:** Audio Stream, Multicore

**Acknowledgements**

The process of completing this research project has been a hard and heavy one lasting the entire duration of the year. I would not have made it to the point I am at now without the help and contributions of certain people. I would like to thank and acknowledge all those who assisted me in getting this far.

I acknowledge the financial and technical support of this project of Telkom SA, Comverse SA, Stortech, Tellabs, Easttel, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

I would like to thank my parents who supported me on both an emotional and financial level. Their love and support gave me the strength to overcome the greatest of challenges and lift me up when faced with impossible challenges.

I would like to thank Nyasha Chigwamba and the rest of the Audio Engineering group for assisting me in the technical aspects of my project. Their assistance uncovered the darkest corners of the Unos Vision application which would have otherwise remained unknown to me. Additionally, I would like to thank Joao Lourenco for all his contributions in the initial stages of this project.

Finally, I would like to thank my supervisor, Prof. Richard Foss, for giving me the opportunity to study under him. His support and guidance greatly affected the outcome of my research and allowed me to overcome many obstacles during the course of the year.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem Statement

There are many contexts where it is necessary to route audio signals across large distances, for example in stadiums, convention centres, etc. In digital audio networks, these systems are often controlled from a centralised console using specialised software which allows audio devices to be interconnected and controlled remotely via a protocol that is common among all the devices.

With regards to audio networks that span large distances, they become difficult to manage from a centralised location since the end-hosts may be out of seeing and hearing range of the user. This could result in issues where a network user believes that they have correctly routed an audio signal between two end-hosts; meanwhile, either through an application error or through user error, the signal is routed incorrectly. In situations such as this the user is forced to traverse all the devices in an audio network to diagnose any possible issues that have arisen and then return to the control console to correct the noted issues. This process can become cumbersome and time-consuming in larger network configurations.

One possible solution to this issue would be that of utilising a portable version of an existing audio network management application where a user can access the audio network from any location with a wireless signal, as demonstrated in figure 1.1. With the advent of powerful smart-phone and tablet devices, the development of a mobile-based application that mimics the functionality of a desktop-based solution has become a feasible option. This application should be able to communicate with an audio network via some wireless

medium, be able to give an effective view of the network utilising the limited screen-space of mobile devices and be able to interact with devices on that network.

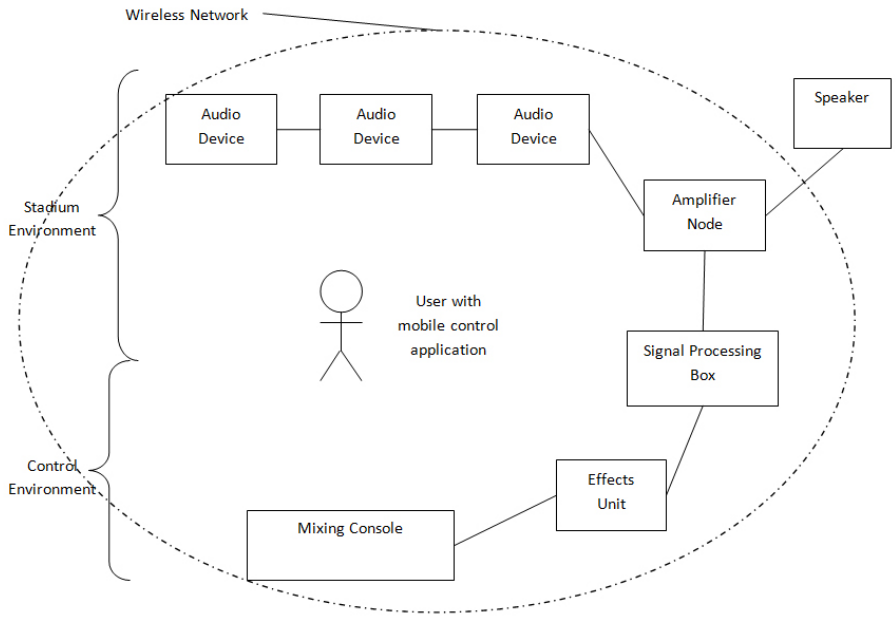


Figure 1.1: Example scenario demonstrating the lack of a centralised control console.

## 1.2 Research Goals and Objectives

The purpose of this research project is to test the feasibility of a portable audio networking application through the development of a prototype. This prototype is based on an existing audio networking application so that research and implementation can be concentrated on providing an effective front-end experience for a mobile platform. The prototype is used to expose any issues that would arise when building an application of this nature.

Primary issues addressed included:

- Performance in terms of hardware resource usage.
- Usability in terms of interaction design.

Secondary issues included:

- Addressing technical design issues specific to the mobile platform. One example is that of file IO operations being handled differently on a mobile platform from a desktop platform.

## 1.3 Project Outline

For the purpose of this research project, the application on which the prototype is based is the Unos Vision application suite. Unos Vision is an audio networking application suite that allows a user to connect, control and monitor devices in a digital audio network. It is able to interact with devices on a firewire-based audio network, with connectivity between the application and the network done using a standard Ethernet connection to a firewire router. Routing an audio signal between two end-hosts is currently a multistage stage process, where the routing internal to the device and the routing between two devices is handled separately. The underlying protocol that enables communications between devices and Unos Vision is the AES-X170 protocol and the application itself is built on Juce, a C++ cross-platform application framework. An extended analysis of Unos Vision, the technical specifications of the AES-X170 protocol and the Juce application framework will be further discussed in chapter two.

The mobile platform on which the prototype application is built is the iPhone OS and the device used was the third generation iPod Touch. The iPod Touch is similar to the iPhone 3G and 3GS models in terms of hardware and software configuration, and all of these devices have gained substantial popularity in mobile markets. This is a contributing factor as to why the iPhone OS was chosen. Additionally, the Juce application framework has been extended to be compatible with the iPhone OS, allowing applications to be compiled for Windows, Linux, Mac OS and iPhone OS platforms. This made the process of porting the application a relatively simple one as only minor changes were required as to get the application compiling for the mobile platform. Issues experienced and modifications made during the process of porting the application from a desktop to a mobile platform will be discussed in chapter three, focusing on the technical requirements for optimising application functionality.

Using the prototype from chapter three, it was discovered that the user interface of the Unos Vision application was less than optimal for small-screen devices. Chapter four explores the design of a patchbay component that aims to simplify the application logic, thereby simplifying the usage of the application on a mobile device. A brief description

of several patchbays is explored and ultimately a grid-based approach was selected. The migration from the original Unos Vision design to a patchbay solution was done by replacing the four stage process for establishing connections between devices into a single step; selecting two end-point IO sockets. All the routing and establishing of connections is done automatically by the application, and the interaction with the network is done via a single grid-based GUI component.

In chapter five the actual implementation of the patchbay and the analysis thereof is discussed. Any further modifications in terms of design and performance required to produce a feasible solution is listed and experimental results are used to compare the mobile patchbay solution to the original Unos Vision design.

Finally in chapter six the project the overall result of this research project is analysed. A final conclusion on the feasibility of a mobile-based audio network application is stated and any further work that could result from this research project is listed.

# Chapter 2

# Background Information

In this chapter the AES-X170 protocol, Juce application framework and the Unos Vision application are introduced and analysed. Additionally, the programming environment in which the prototype will be built is explored, focusing on Apple's iPhone SDK for the iPhone OS versions 3.2 and 4. This analysis is conducted in order to gain a better understanding of the Unos Vision application and to gain an overview as to how a basic X170 audio network functions.

## 2.1 AES-X170 Protocol

The AES-X170 protocol, allows for the remote administration of devices in an audio network [13]. It is a UDP/IP-based peer-to-peer protocol that uses a hierarchical addressing structure to communicate with the parameters on a device and manage audio streams between devices [13].

X170 messages will generally fall under three main types [8]:

- Connection management messages - messages that facilitate the routing of audio streams between devices.
- Control messages - messages that adjust the value of a parameter/parameter group, establish relationships between parameters and provide a means with which to interact with the device.
- Monitoring messages - messages that retrieve parameter information, such as metering values.

### 2.1.1 Connection Management

The X170 protocol allows for the routing of audio signals between devices through the use of a multicore [15]. A multicore is a network connection between two devices whose concept is similar to the physical cable connections in an analogue audio network [8]. Each multicore consists of a several audio channels known as multicore sequences and it is these sequences that are responsible for transmitting an actual audio signal [8]. Figure 2.1 demonstrates two multicore connections within a network cable, where each multicore sequence is portrayed by the dashed lines. Figure 2.1 also demonstrates the multicore sequences being housed in a multicore, and these multicores are connected to either a multicore in or multicore out socket on a device. A multicore transmits data in a single direction, therefore to establish a bi-directional relationship between two devices two multicore connections will need to be created.

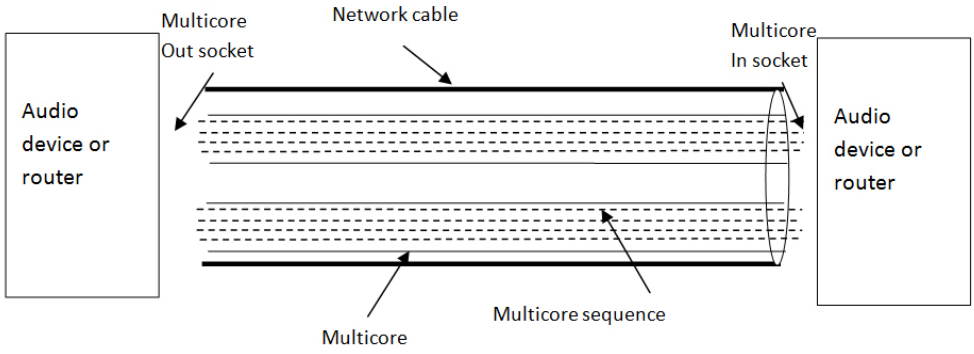


Figure 2.1: Two multicore connections between two devices [15]

Further action beyond establishing a multicore connection is required to transmit an audio signal between two devices. A device may have an array of audio inputs and outputs each of which may be routed to a multicore sequence [7]. For an audio signal to be transmitted via a multicore, the desired source and destination end-points need to be connected to the same multicore sequence of the same multicore [7]. This can be portrayed as the internal routing of a device and a mock-up of this routing is displayed in table 1 in a grid structure. In table 1 we can see that Analogue In 1 is patched to Multicore 1's 1st multicore sequence, and Analogue In 1 is patched to Analogue Out 1 of the same device.

In order to create a basic connection between an audio input of one device and an audio output of a second device where each device is on the same subnet, the process involves three steps:

| | Analogue Out 1 | Multicore1 Out 1 | ... |
|---|---|---|---|
| Analogue In 1 | X | X | |
| Multicore1 In 1 | | | |
| ... | | | |

Table 2.1: Sample table displaying internal routing matrix.

1. Create a multicore connection between two devices.
2. Create a connection between an audio input on a source device and a multicore sequence in the multicore from step 1.
3. Create a connection from the audio output on a destination device to the same multicore sequence of the same multicore in step 2.

### 2.1.2 Routing

In an X170 network, the routing of multicores within a router is done in a mirrored fashion; in other words each input multicore of each interface on a router is routed to the corresponding output multicore of every other interface on that same router [6]. Looking at figure 2.2, if a multicore connection was created between Audio Device 1 and NIC1 (network interface controller) on the router, that multicore would be accessible to any device connected to NIC2. In order for Audio Device 2 to establish a multicore connection to Audio Device 1, a multicore connection would need to be created from Audio Device 2 to the output multicore of NIC2 that has the same ID as the input multicore of NIC1 of the router, this input multicore being connected to the output multicore of Audio Device 1.

In order to connect an audio input and output between two devices on adjacent subnets, the process would involve the following steps:

- Create a connection between an audio input on a source device and a multicore sequence in a multicore.
- Create a connection between an audio output on a destination device and a multicore sequence in a multicore.

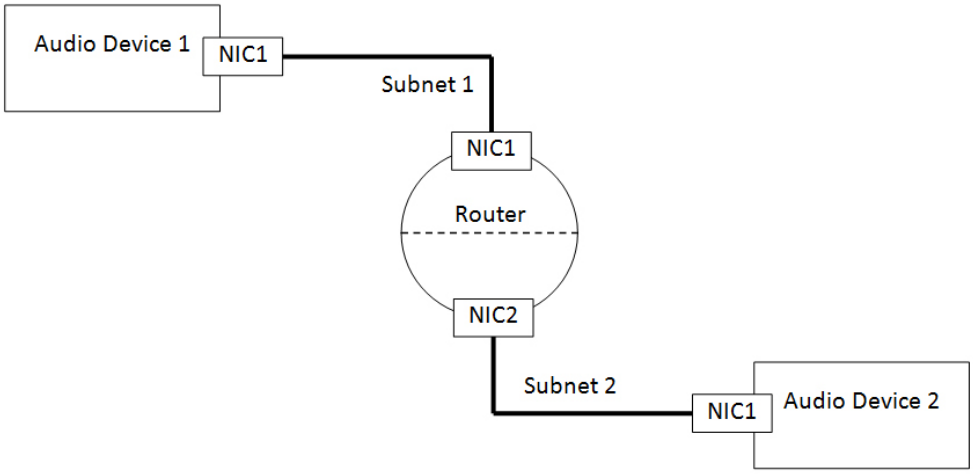


Figure 2.2: Multicore router routing between two subnets.

- Create a multicore connection between the source device and an X170 router.
- Create a multicore connection between the destination device and an X170 router, ensuring that the multicore ID is the same as the one in step 3.

When connecting devices on remote subnets, multiple multicore connections need to be created between routers in every subnet until a single, continuous multicore stream is establish between the two devices.

### 2.1.3 Parameters and Desk Items

There are three methods of access for any parameter that adheres to the X170 stack[8]:

- A value can be polled from the parameter
- A value can be set to the parameter
- A value can be 'pushed' from the parameter to a target remote parameter

A parameter's value can be retrieved or set through a basic 'get' or 'set' X170 command, respectively[8]. This behaviour may become undesirable for cases where diagnostic data is updated in frequent intervals, such as with metering data. The X170 protocol has a

'push' system implemented where a parameter can continuously feed data to a remote parameter. This is done by sending a 'set push' X170 command to a parameter[8].

Parameters can be grouped to establish either a master-slave or a peer-to-peer relationship with other parameters[7]. In a master-slave parameter group, a master parameter controls a set of slave parameters. This means that any change in the master parameter is echoed throughout all the slaves, but a change in any slave parameter does not reflect on the master or any of the other slaves[7]. In a peer-to-peer group, a change in any single parameter will affect all the parameters in the joined group[7].

Desk Items are control items that can be used to provide a virtual representation of a device[8]. They allow for interaction with parameters that are associated with a desk item and can provide a graphical means with which to control and monitor a parameter[14]. Each device stores its own desk items, which include a visual representation of the desk item, the nature of the desk item and the parameters associated with the desk item[8].

Figure 2.3 demonstrates two common types of desk items:

- Image A demonstrates a fader desk item which can be used to adjust the value of a parameter.
- Image B demonstrates a meter desk item which can be used to display metering values, such as volume levels.

### 2.1.4 Protocol Structure

The Open Systems Interconnection model, known as the OSI model, is a communications system that divides network communications into seven manageable sections[11]. The layers in order from highest to lowest level are [11]:

- Application level - responsible for interaction with an application and contains application data.
- Presentation layer - responsible for the format of the data, as well as encryption and decryption. This is an optional layer.
- Session layer (Optional) - responsible for maintaining connectivity between applications on end-hosts. This is an optional layer.

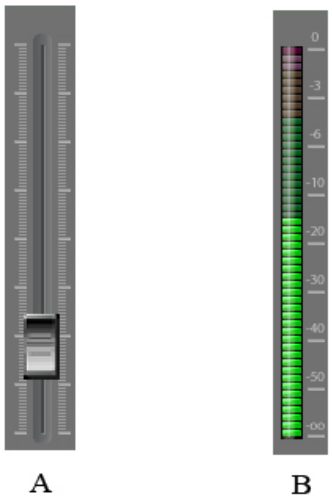


Figure 2.3: Example of a fader and a meter desk item.

- Transport layer - responsible for controlling end-to-end connectivity between hosts in segments, for example the TCP and UDP protocols.
- Network layer - defines the logical addressing for data transmission in packets, for example the IP protocol.
- Data link layer - defines the physical addressing for data transmission in frames, for example a devices hardware address.
- Physical layer - defines the physical medium for data transmission in bits.

In terms of the OSI model, the X170 protocol is sent via the UDP/IP protocol. Each message consists of three main sections [8]:

- an IP header containing the source and destination addresses for the message
- a UDP header containing the source and destination ports for the message
- a data field containing the X170 header and address block

The benefit of this structure is that any network device with IP and UDP support can send and receive X170 messages as long as the X170 stack is implemented as an application layer protocol on a device. Figure 5 demonstrates the structure of an X170 message [8].

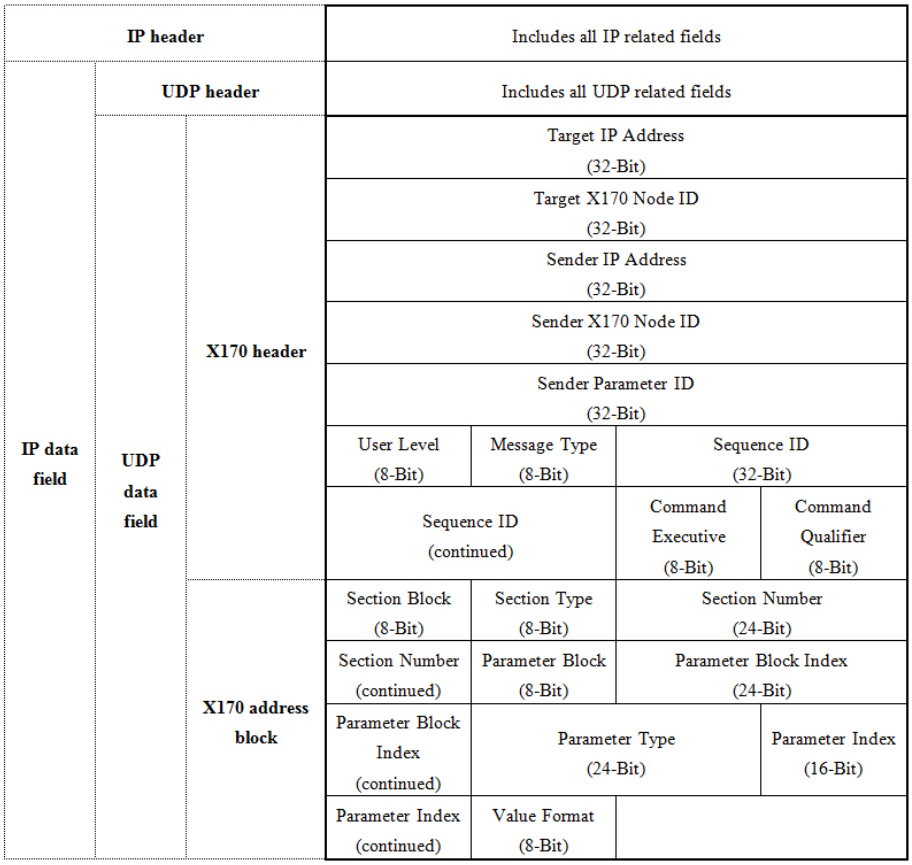


Figure 2.4: Structure of the AES-X170 protocol

The *IP header* and the *UDP header* are used to facilitate the transportation of an X170 message on the Network and Transport layer respectively. The UDP segment is contained within the IP packet's data field, and the X170 message is contained within the UDP segment's data field [8].

The X170 header is used to define the nature of the message, as well as provide some additional addressing information required to identify a device at the Application level[8].

The fields in the X170 header have the following functions [8]:

The *Target* and *Sender IP address* fields contain the IP addresses of the source and destination devices[8]. This field enables the IP addresses of the devices in question to be accessible at an application level.

The *Target X170 Node ID* and *Sender X170 Node ID* are used to uniquely identify a device. In situations where several devices connect to an X170 network via a proxy

device, each device will share an IP address with the proxy device[8]. The node ID field will allow each device to be uniquely identified beyond that of its IP address.

The *Sender Parameter ID* allows a destination parameter to refer back to an originating parameter from which a message was sent[8]. This is for cases where a change in one parameter affects a second parameter, and the second parameter may need to interact with the first parameter.

The *User Level* field allows for the assignment of user levels to a message so that a parameter can modify its behaviour accordingly[8].

The *Message Type* field specifies the role of the message and can identify whether a message is a request or a response[8].

The *Sequence ID* facilitates the transmission of multiple messages. When sending a large array of messages, they are done so without waiting for receipt responses. This sequence ID allows for responses to be paired with an originating message, hence confirming receipt of a message[8].

The *Command Executive* field defines the nature of a message, such as whether it is either getting a parameter value or setting it[8]. The *Command Qualifier* field can be used to specify an attribute of a parameter or it can refer to a group of parameters[8].

The X170 protocol was established as a means of proving a standardised method of control among devices in an audio network. Since an audio network will consist of a variety of devices, each with their own set of parameters, a generic method for addressing parameters was implemented that will allow any parameter of any device to be read and modified. To enable this, a seven level hierarchical addressing structure was created [7].

The levels of the hierarchical addressing structure in order from highest level to lowest are:

- Section Block
- Section Type
- Section Number
- Parameter Block
- Parameter Block Index

- Parameter Type
- Parameter Index

The *Section block* aims to define a device in terms of its abilities. A single device may have several sections, and the section block clusters parameters into a specific section, such as input section or output section[8].

Within a section block there may be different types of components. The *Section Type* address aims to further group a parameter to a specific type in a block. One example is that within an Input section block of a device there may be microphone inputs, line inputs and tape inputs, yet all are part of the same section block[8]. These components will each have their own section type; that is within an input section block all the microphone inputs may be grouped as a single section type.

The *Section Number* is used to identify a channel or interface number and is responsible for the successful processing or routing of a channel[8]. In networks where there are a multitude of active channels, this field allows any channel to be tracked and uniquely identified.

The *Parameter Block* field allows for several parameters to be associated as a group. These parameters are often used in conjunction with each other to allow for the processing and routing of audio channels. An example given by Foss 2010[8] is that of using a block of equalizers to provide a wide range of equalization of an audio channel.

The *Parameter Block Index* allows for parameters joined in a group to each be further classified into sub-groups. Expanding on the aforementioned equalizer example, this would refer to uniquely identifying parameter groups in a parameter block as being responsible for Frequency or Gain[7].

The *Parameter Type* field allows a parameter to be defined as a specific type. Parameter types define what a parameter is responsible for, such as gain, frequency or threshold[7].

The *Parameter Index* is the lowest level of addressing of a parameter and refers to an individual parameter in a group of parameters that process an audio channel[8]. For example, it can be used to distinguish a single parameter in a group of gain parameters processing an audio channel.

This hierarchical structure has the additional benefit of allowing an array of parameters to be accessed via a single message by using a "wildcard" parameter at any level of the

addressing scheme[8]. For example, if a wild card value was passed into the parameter index, the X170 message would be parsed by every parameter at that specific level.

### 2.1.5 Limitations

The specification of the X170 protocol only accommodates routing devices that are in the same subnet[8]. The X170 protocol does not offer any automated routing mechanisms between devices on different subnets. Routing between subnets is achieved by connecting an end-host to a routing device on one subnet and then connecting another end-host to that same router on an adjacent subnet. This action is done on the application level in terms of the OSI stack.

The number of multicore connections that a device may establish is limited and varies between devices. For example, an evaluation board may only support up to two multicore connections at a time, while an X170 router may support eight[7]. This means that special attention needs to be paid to the structure of an audio network as bottlenecks may occur due to a lack of available multicores.

The X170 protocol does not allow for the aggregation of output multicores; however a single output multicore may be distributed to multiple input multicore sockets[8]. This means that a source audio signal may be patched to an array of destinations, but multiple sources may not be patched to a single destination.

## 2.2 Programming for the iPhone OS

The iPhone OS has become a popular platform on which to deploy applications. The iPhone OS has achieved further popularity since its adaptation from the iPhone to the iPod Touch and the iPad. In terms of architecture all three of these devices are identical[2]. The only major differences between the iPhone, iPod and iPad in terms of functionality are the size of the screen and the inclusion of a GSM-based antenna[3]. Of the three devices, the iPad has the larger screen whereas the iPhone and the iPod have identical screens, and the iPod is the only device that does not include the option of a GSM-based antenna.

Of the three devices the third generation iPod Touch was chosen for the research project. A device with GSM capabilities was not required and in terms of network functionality

the iPod offers a standard 802.11 b/g wireless antenna. The iPad, albeit similar to the iPod, was not chosen due to its limited availability.

### 2.2.1 Specifications

The third generation iPod Touch has the following specifications[3]:

- Arm A8 600MHz CPU
- 256MB Ram
- PowerVR SGX GPU
- Capacitive multi-touch touchscreen with a 480x320 resolution
- 16GB or 32GB storage capacity
- 802.11 b/g wireless connectivity
- iPhone OS 3.1.1 (can be upgraded to iPhone OS 4.1)

### 2.2.2 Requirements

In order to compile an application for an iPhone OS-based device using Apple's iPhone SDK, the following requirements must be met[1]:

- The application must be compiled on an Intel-based Apple system
- The system must have at least 1GB RAM
- The host operating system on which the application will be programmed should be Apple's Mac OS X 10.5.7 or later
- Apple's iPhone SDK needs to be installed using XCode 3.1.3 as the IDE
- A device running iPhone OS 3 or higher

The iPhone SDK offers a simulator on which to test applications before deployment; however there are further requirements if a user wants to deploy an application onto a physical device[1]. This comes in the form of a provisioning certificate. This certificate is a form of digital rights management, commonly known as DRM, which pairs an application with a device, allowing the application to run on the device.

## 2.3 Juce

Juce is a cross-platform application framework written in C++ with a strong focus on creating audio-centric applications. Using Juce, a user is able to create a consistent graphical experience across multiple platforms, including Windows, Linux, Apple Mac OS and iPhone OS operating systems. Programming environments that are explicitly supported are Microsoft's Visual Studio, GIT for Linux and Apple's XCode IDE, including the iPhone SDK environment[17]. In addition to the benefit of being cross-platform, Juce also offers a comprehensive API reference guide in the form of a Doxygen generated document [18].

### 2.3.1 Features

The current iteration of Juce offers a large variety of features. Some of these features include[17]:

- cross-platform multi-threading support including synchronisation between threads, events, thread pools and other features that allow the controlling of threads,
- messaging classes that facilitate event-driven applications including asynchronous callbacks, timers, etc.,
- rich GUI elements through the use of a look-and-feel subsystem, MDI windowing and window management system, and other GUI-based components,
- cross platform audio driver classes supporting a variety of audio formats as well as extensive MIDI support,
- software-based graphic rendering system,
- global logging facilities,
- extensive network communication support,
- support for various cryptographic algorithms,
- zip archive decompression.

Juce also has a comprehensive core class that handles elements that would otherwise have to be programmed on a per-operating system basis, such as the management of files and input/output streams[17].

In addition to these features, the Juce framework includes an application called "The Jucer". The Jucer is a UI component builder application that allows a user to construct custom graphical components based on existing Juce controls. This allows the user to tailor the behaviour and look of their application depending on the nature of the application[18].

### 2.3.2 Targeting Specific Platforms

Juce achieves cross platform compilation by including native libraries specific to each supported operating system. Depending on the environment, one of the platform specific classes will be used to compile an application. The targeting of platforms is achieved through the use of pre-processor directives.

Pre-processor directives are sections of code that are exclusively handled by the compiler at compile time and are not included in the runtime application[5]. One example of a pre-processor directive is the #include directive, common to most C-like languages. At compile time, the compiler will treat the #include directive as if it were replacing that line with the contents of the file it was including [12]. Pre-processor directives can also be used to execute specific sections of code depending on the environment variables.

```
#if JUCE_IPHONE
    //execute iphone specific code
#endif
```

Figure 2.5: Example of using a preprocessor directive to execute iPhone specific code

In figure 2.5, a #if directive is used to only compile a section of code if the targeted platform is the iPhone. The #if directive is synonymous with an if-statement, where a block of code will only be executed if a condition is met. The #endif directive is used to indicate the end of the platform-specific code. In short, all the code between the #if and the #endif will be executed if the targeted platform is the iPhone OS version 4.

By combining the #include directive with the #if directive, a user can include specific libraries depending on the platform being used. Figure 2.6 displays a mock-up of this usage:

```
#if JUCE_IPHONE
    #include <iphoneIOLibrary.h>
#elif TARGET_OS_WINDOWS
    #include <windowsIOLibrary.h>
#endif
```

Figure 2.6: Preprocessor directive to include either iPhone or Windows specific code depending on the programming environment.

This usage of native operating system libraries and pre-processor directives means that an application will only work for the platform that it has been compiled for. If an application was compiled in Visual Studio for the Microsoft Windows platform that application will only run on a Windows platform since the application is compiled into platform-specific machine code. When using Juce to get a working version of an application for another platform, that application will need to be compiled by a compiler native to that platform, such as using Apple's XCode environment to create a Mac OS version of the application.

### 2.3.3 Juce Usage

There are two supported methods for including Juce into a project[17]:

- as a statically linked library
- as an "amalgamated" C++ file

To include Juce as a statically linked library, a user needs to include the #include <juce.h> directive in the files where Juce code is referenced. When using this method, the Juce library needs to be compiled separately from the application and acts as an external resource to the project[17].

An alternative method to including the Juce libraries is to use an amalgamated version of Juce. This amalgamated version contains the entire Juce library in a single file providing a conceptually simple method of including the Juce libraries. A problem with this method is that some compilers may have issues parsing the amalgamated file due to its size. To use this method, the #include <juce_amalagamated.h> directive needs to be used where any Juce-based class is referenced[17].

### 2.3.4 Limitations

The current version of Juce at the time of conducting this research was 1.51. At this point the iPhone support for Juce was at an immature state, and as such still had several iPhone OS specific issues. One such issue is in regards to the rotation of applications. When using the rotation mechanism on an iPhone device certain GUI components become distorted[19].

## 2.4 Unos Vision

Unos Vision is a management suite that provides a visual representation of an audio network and allows users to interact with the devices on that network using the X170 protocol[15]. Interaction between the user and the devices on an audio network is achieved by providing a graphical interface that[15]:

- monitors each device on an audio network, as well as displaying each device's status and parameters,
- allows the user to control the parameters of a device, such as manipulate volume levels and create relationships between parameters among a variety of devices,
- Allows the user to create connections between devices, including the routing of audio signals.

At the time of conducting this research, Unos Vision only had support for IEEE1394 networks with Ethernet AVB network support currently in development. The tested version of Unos Vision is able to interact with an IEEE1394 network through the use of a Firewire router which bridges an Ethernet-based client hosting the Unos Vision application to an IEEE1394 network[15].

Figure 2.7 displays a typical network configuration for a digital audio network using a Unos Vision client:

An audio network will typically consist of audio and routing devices. Multicore routers act in the same fashion as Ethernet routers as they forward traffic between two different subnets. In most cases, each device in a subnet will have a device with routing capabilities as its default gateway so that any routing of multicores and X170 messages between subnets is managed by that device.

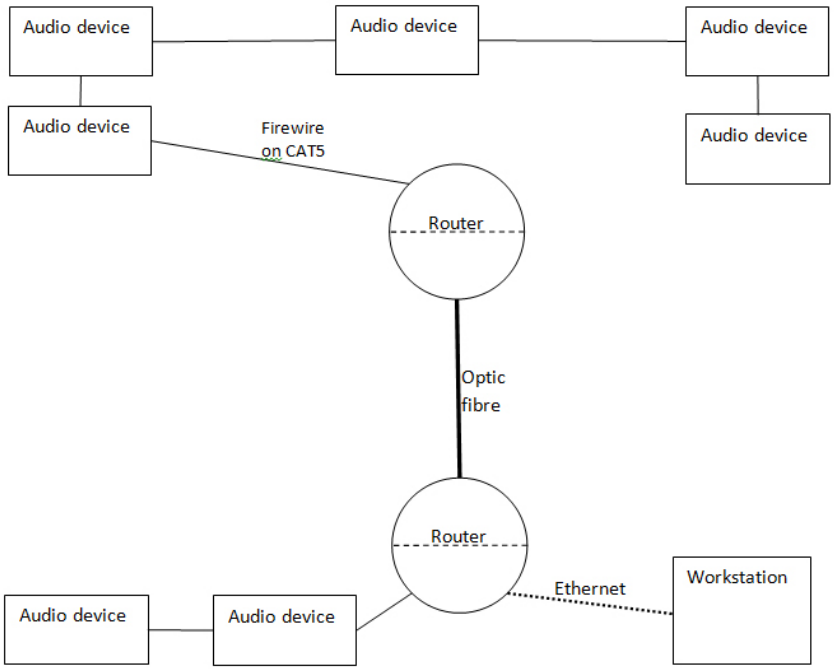


Figure 2.7: Audio network using a Unos Vision workstation[15]

### 2.4.1 Features

The current version of Unos Vision offers the following main features[15]:

- Create and destroy multicore connections between devices and manage connections internal to a device.
- Display desk item data from a device and control the parameters bound to that Desk Item.
- Establish relationships between parameters and parameter groups, including master-slave and peer-to-peer relationships.
- Create desk item components by using an alternative version of Unos Vision called Unos Creator[14].

Unos Vision has no support for creating end-to-end connections between the audio inputs and outputs of remote devices. Connectivity between devices is handled in the same fashion as described in section 2.1.2.

### 2.4.2 Usage

The Unos Vision application is divided into two main tabs[15]:

- Devices tab
- Connection Manager tab

Figure 2.6 demonstrates the Devices tab which provides an overview of an entire audio network and displays a summary of the devices for a selected subnet. Container 1 displays the devices on a specific subnet, as well as the status of each device. Container 2 lists all the subnets on the network and provides a preview of the devices on each subnet.

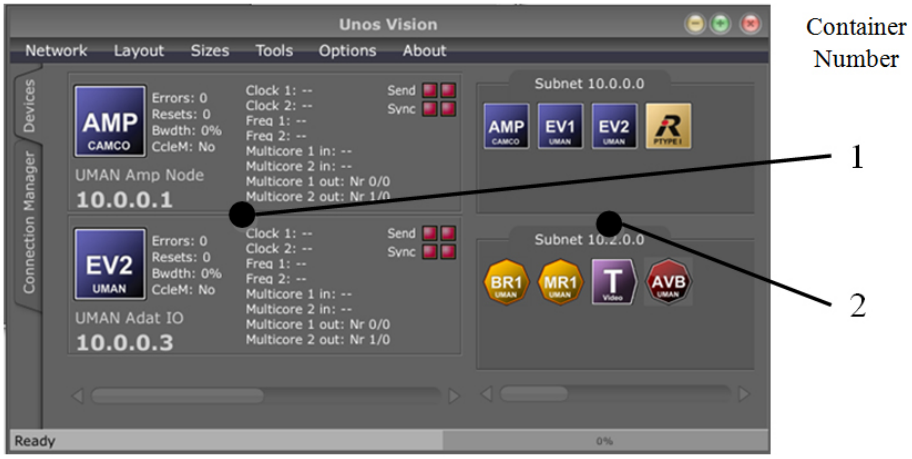


Figure 2.8: Devices view of Unos Vision

Figure 2.7 demonstrates the Connection Manager tab, which is divided into six main containers. The containers 1, 2, 4 and 5 allow the user to create and destroy connections. The columns of the grid views display the outputs, while the rows display the inputs.

- Container 1 displays all the devices in a specific subnet. By selecting a cross-point in this container, all the other containers will be updated with the data pertaining to the devices related to that cross-point[15].
- Container 2 displays the possible multicore sockets for each device, where selecting a cross-point will create a multicore connection between the devices[15].

- Containers 4 and 5 allow the user to manage the internal routings of the two devices selected in container 1, where container 4 represents the source device and container 5 the destination device. For example, the cross-points in containers 4 and 5 allow a user to connect a multicore sequence to an Analogue, ADAT or AES input or output[15].

- Containers 3 and 6 display a device's graphical desk item component, allowing a user to interact with the parameters of a device[15].

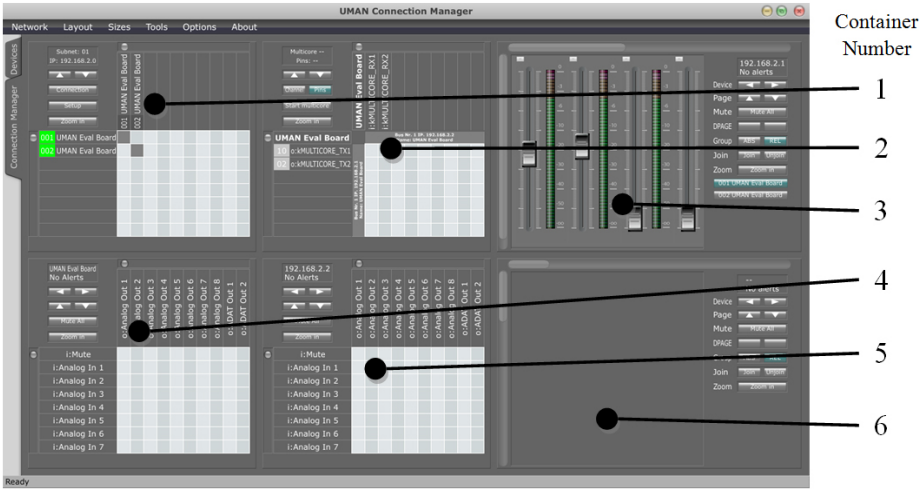


Figure 2.9: Connection Manager view of Unos Vision

### 2.4.3 Structure

The Unified Modelling Language will be used to describe the functionality and structure of Unos Vision. The Unified Modelling Language, commonly known as UML and hereafter referred to as such, provides a visual means to represent the structure, behaviour or the interactions of an application through several types of diagrams[10]. Three types of diagrams will be used to demonstrate these attributes; use case diagrams to demonstrate application behaviour, sequence diagrams to demonstrate interactions, and class diagrams to demonstrate structure[10].

Figure 2.8 is a Use Case diagram demonstrating the high-level functionality of Unos Vision. The *"User"* and *"X170Stack"* icons represent external actors. These are entities that are not directly involved with the functioning of the application, but are necessary in the execution of the application. The User actor represents the *user* that will interact

with the application, while the *X170Stack* actor represents the interface between the application and the actual audio network. The *X170Stack* is responsible for the sending and parsing of X170 messages.

The *"Start up"*, *"Make connections"*, *"Break connections"*, *"Load desk items"*, and *"Control parameters"* labels refer to use cases. These use cases describe the high-level functions of the application that the external actors interact with.

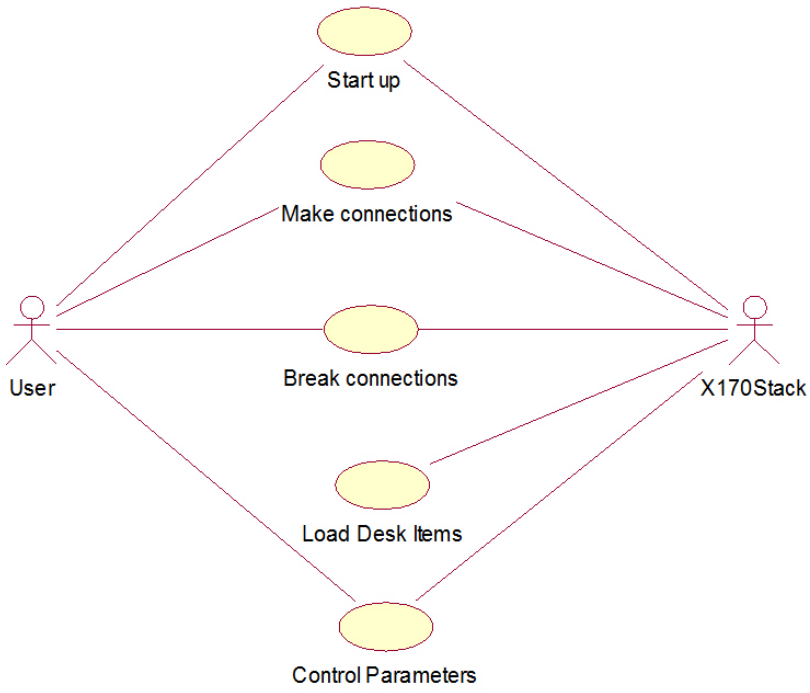


Figure 2.10: Use case diagram of Unos Vision

Figure 2.9 is a class diagram that describes the overall structure of the application. Main areas of focus are:

*Network* - The *Network* class provides a virtual representation of the audio network and allows the application to interact with devices, their parameters and their multicore connections. The *devices* in the *network* object are divided into busses, where a *bus* is synonymous with a subnet. The *Network* class is populated via the *NetworkClient* class which interacts with the *UMANDLL* class

*UMANDLL* - The *UMANDLL* class provides an interface between the application and the *X170Stack*. This class is responsible for initializing the *X170Stack*, as well as constructing and parsing X170 messages.

*Device View* and *NetworkView* - These components are responsible for displaying the containers in figure 2.6 and 2.5 respectively. Additionally, this class handles some high-level connectivity logic, such as checking if a connection between two multicore sockets is possible.

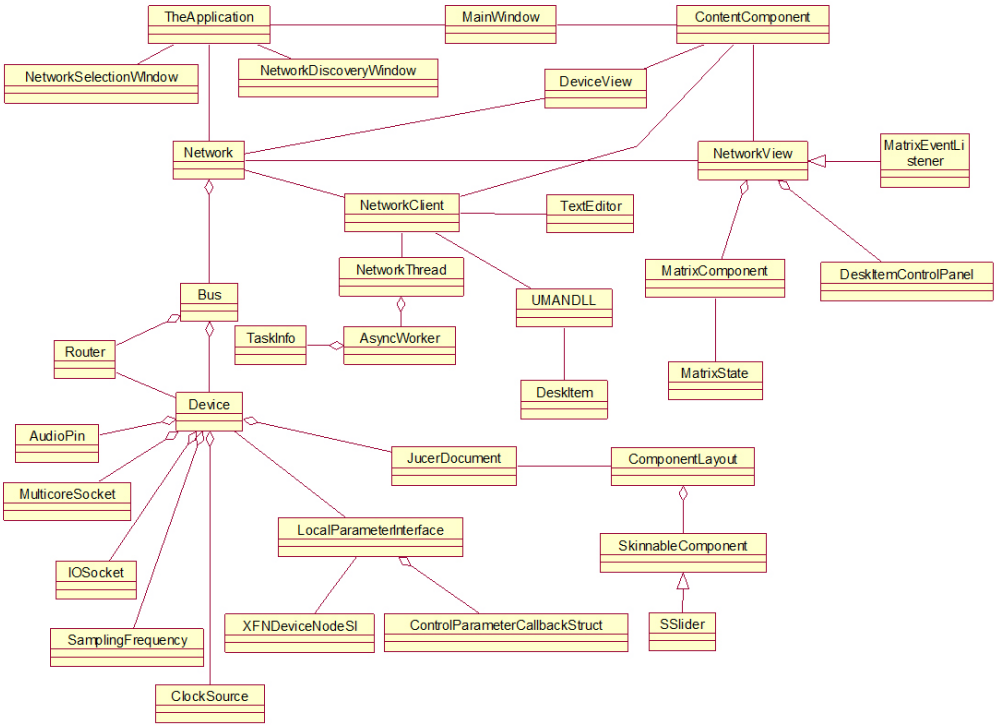


Figure 2.11: Class diagram of Unos Vision.

Figure 2.10, 2.11 and 2.12 demonstrate sequence diagrams focusing on the interactions between an application and the X170 stack.

Figure 2.10 details the portion of the start-up procedure that is responsible for initialising the XFNStack and scanning the network for devices.

Figure 2.11 demonstrates the toggling of a firewire multicore connection.

Figure 2.12 demonstrates the establishing of an internal connection.

## 2.5 Conclusion

This chapter summarised some of the key concepts required to understand an X170-based audio network focusing on the connection management aspects of the Unos Vision ap-

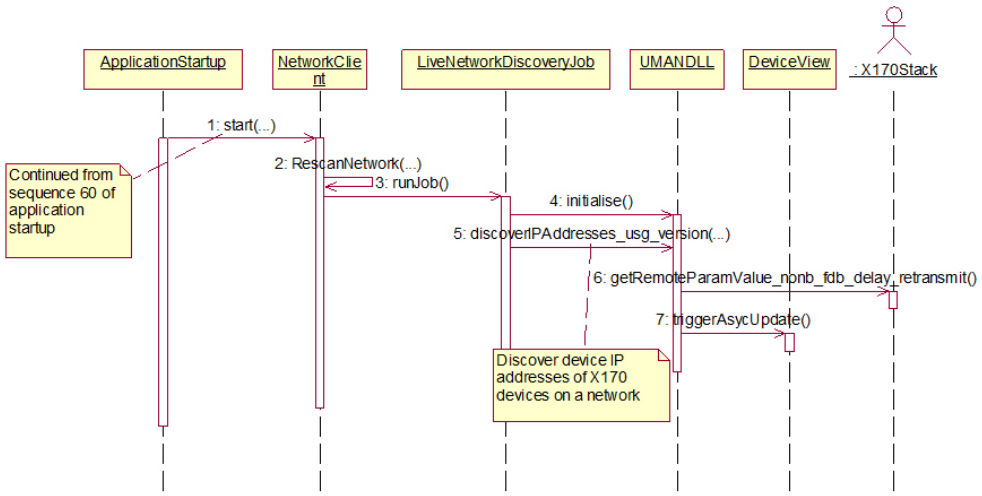


Figure 2.12: Sequence diagram summarising X170 communications at Unos Vision start-up.

plication suite. A description of how an audio signal may be routed has been discussed and a high level overview of the Unos Vision application has been presented using UML diagramming techniques. An introduction to the iPhone OS platform and the Juce application framework gives a basic understanding of some aspects to consider when porting an application to a mobile platform, and a possible programming approach to maintaining cross-platform compatibility has been discussed.

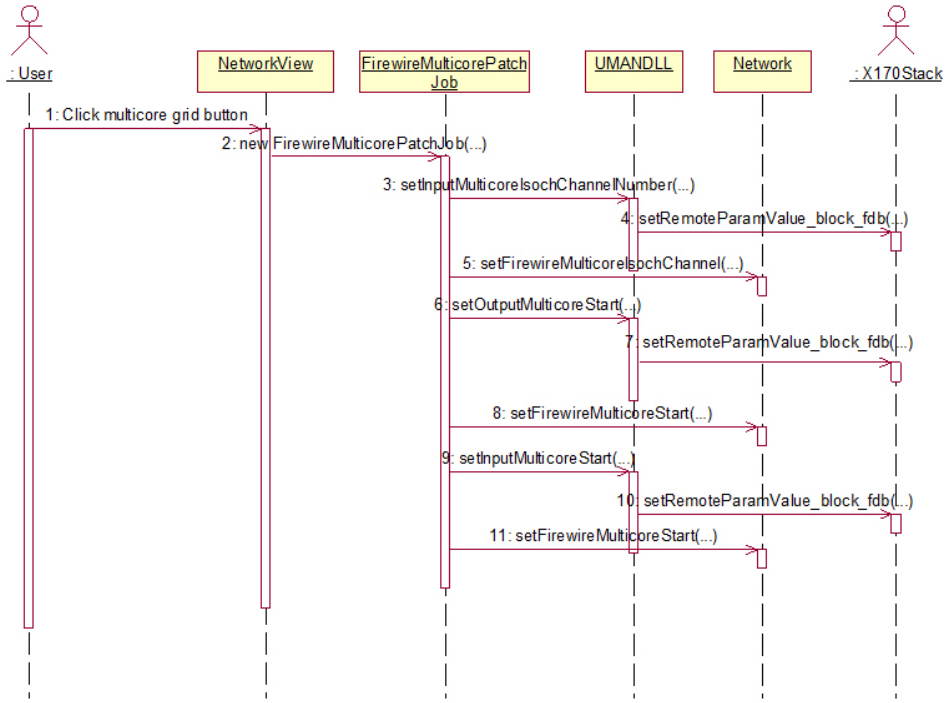


Figure 2.13: Sequence diagram describing multicore patching.

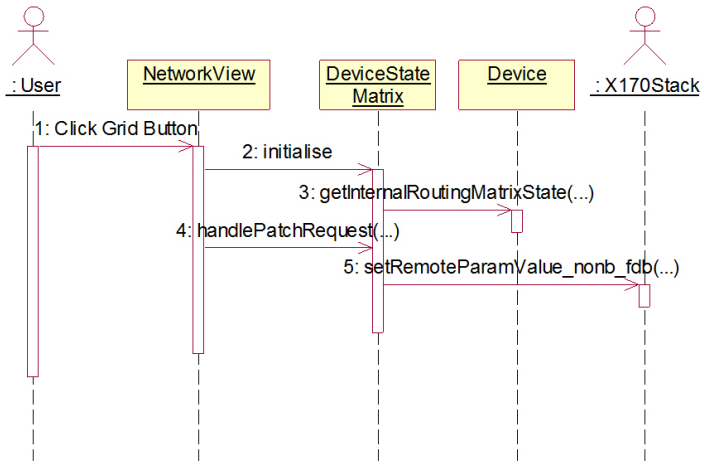


Figure 2.14: Sequence diagram summarising internal routing of a device.

# Chapter 3

# Porting Unos Vision

This chapter summarises the process for porting a desktop application to a mobile platform. The iPhone OS development environment is briefly introduced, detailing several utilities that aid in debugging mobile applications. This is followed by a description of the modifications required of the mobile application, and finally there is an analysis of the functionality and the usability of the mobile application performed.

The initial prototype was built to test the underlying functionality of the Unos Vision application and its compatibility with a mobile platform. It was therefore chosen to limit the changes implemented into this prototype so that any major issues in the application's design could be exposed. This prototype provided a starting point from which a more suitable mobile-based audio networking application could be based.

## 3.1 Environment

The Apple iPhone Software Development Kit (iOS SDK) allows a user to create applications and compile them into iPhone OS compatible Arm machine code using the XCode Integrated Development Environment. In addition to being able to compile and deploy an application to an iPhone OS device, the iOS SDK integrates a simulator component in the the XCode IDE that allows an application to be tested in a simulator environment, simulating either an iPhone OS 3, iPad or iPhone OS 4 device. Ballarb [2007] [4] recommends to not solely depend on the simulator environment during the testing phase of a mobile application, but to use the simulator to test application logic and thereafter use an actual device to test usability. Factors behind this reasoning include [4]:

- The simulator and the device are based on different architectures, where the simulator uses an x86 instruction set and the device uses an Arm instruction set.
- The simulator depends on a mouse pointer for interaction, while the device uses touch-based interaction. The end-user will interact with the simulator in a different manner from that of the device.
- Simulator code may work differently from device code.

Another limiting factor of the simulator is that of the hardware configuration. When using the simulator the application relies on the host computer's hardware, while the device will use its own hardware. This may cause the simulator to hide any potential hardware issues in terms of performance. It is for these reasons that the simulator was only used to test application functionality in the initial stages of the porting process. For the final stages of development, the focus was on optimising the application for a physical device.

Apple's Instruments utility allows a user to measure the hardware resource usage of a variety of Apple devices, including iPhone OS devices, and can aid a user in identifying hardware bottlenecks. It features a variety of tools that allow a user to customize which resources to monitor, the majority of which provide a graph-structure for displaying the data. For the purpose of the prototype the following tools were used to test the application's performance:

- WiFi monitor - Measures the 802.11 wireless antenna's usage
- Memory monitor - Measures the physical memory and virtual memory usage
- CPU monitor - Measures the CPU load of a device and can distinguish the load distribution between the operating system and user applications.

## 3.2 Implementation

This section describes the changes made to the Unos Vision application to get a satisfactory level of functionality on the iPhone OS platform. The implemented changes focused on providing a basic level of interaction between the user and an audio network. The main objectives of this prototype were:

- Get Unos Vision to compile and start up on an iPhone OS device.
- Create and destroy multicore connections, as well as manage the internal routing of a device.
- Load and control desk item components.

### 3.2.1 Prerequisites

The XFNDLL and Juce source files have to be compiled as static libraries in the XCode environment. This will produce a libjuce.a and a libXFNStack.a file, which can be added as an external resource in the Unos Vision application. This will allow the application to use the XFNDLL set of classes and the Juce Library with no changes required of the base code.

When compiling the Unos Vision prototype application the Unos Vision, XFNDLL and the Juce library source code all need to be compiled to the same architecture. When deploying to a physical device, all three sets of source code need to be compiled for the Arm architecture, and when deploying to the simulator environment all three sets of source code need to be compiled for the architecture of the operating system hosting the simulator.

### 3.2.2 Modifications

Several modifications to the application logic and the user interface had to be made before the prototype was at a useable level.

In the desktop version of Unos Vision, on application start up a modal dialogue window is presented to the user listing all the IP addresses of the network interfaces on a host computer. The purpose of this window is so that the user can select the IP address connecting the host computer with the X170 network. The modal behaviour caused issues in the iPhone environment where the modal window would not display and would cause the application to lock-up. Since current iPhone devices have a single wireless interface called ”en0” [2], this issue was resolved by disabling the modal window and setting the IP address to the value taken from the ”en0” network interface.

The horizontal layout of the devices view, as displayed in figure 2.6, did not display correctly due to the limited horizontal screen space of the iPhone device. This view was

modified so that it had a portrait layout. Additionally, the size of the scroll bars in this view was increased to accommodate touch-based input.

It was found that six container layout, as displayed in figure 2.7, provided a suboptimal user experience on a small-screen device. Unos Vision has a tab-based layout that divides the six containers into six separate tabs, as shown in Figure 3.1, which can be used to accommodate small-screen devices. The prototype application was modified so that it would start in this tab-based layout.

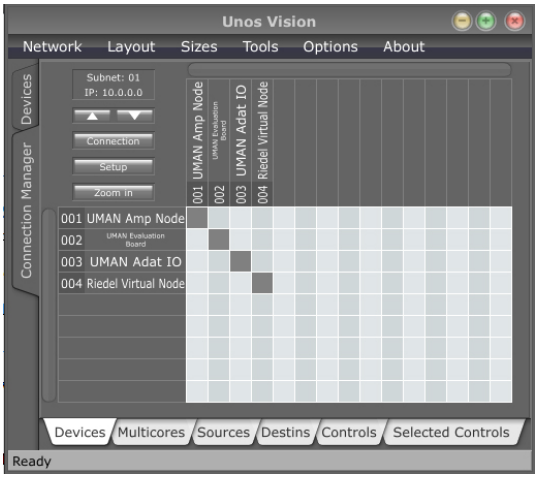


Figure 3.1: Screenshot of Unos Vision's tab-based layout.

The iPhone OS provides an application with a "sandboxed" environment that manages the resources that an application has access to [2]. One of the resources that are restricted is that of access to the file system. The directories that an application has read and write access to are:

- <Application_Home>/Documents/
- <Application_Home>/Library/ - including child directories
- <Application_Home>/tmp/

To adhere to the sandbox rules, any components of the Unos Vision application that required access to the file system, including desk items, were modified to use one of the above directories.

The prototype application was further modified to remove redundancies, such as toggling the full screen mode of the application. The final effect of these changes can be seen in

figure 3.2, demonstrating the devices view, the multicore matrix and a desk items control panel respectively.


Figure 3.2: Screenshots of the final iPhone application prototype.

## 3.3 Evaluation

After implementing the changes in section 3.2, it was found that the core functionality of the Unos Vision application was preserved. The prototype was able to provide the same level of functionality as the desktop application, allowing a user to manage multicore connections between devices, manage the internal routing of a device and display and manage desk item components.

An issue that was experienced with the prototype application was that of hardware performance. When the application interacts with devices on an audio network and downloads desk item files, it needs to parse all the data that it receives. This can be a hardware intensive process in terms of network bandwidth and CPU usage. Figure 3.3 shows a screen captured from Apple's Instrument Utility over a period of one minute. This is a recording of the prototype application while it was downloading desk item data from each device in a network containing four X170 devices.

It can be seen that during this process the device's wireless network and memory usage is at 100%. The CPU spends the majority of this time interval at 100% usage with usage dropping at irregular intervals. This performance issue becomes apparent when navigating the tab-based layout of the application as response times of interactions from the user become large. This prompted an evaluation of the user interface so as to provide

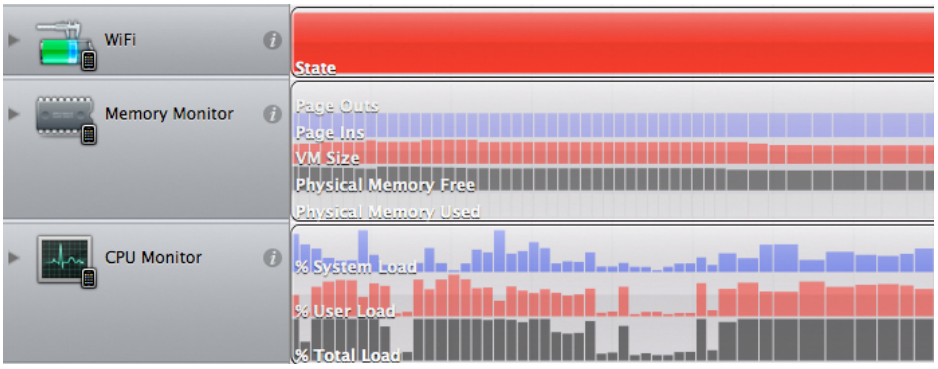


Figure 3.3: 1 Minute recording of Apple's Instrument utility measuring the prototype's hardware usage.

a basis for designing a simplified means of interaction with the application that required minimal interactions from the user.

### 3.3.1 User Interface Evaluation

A simplified version of the Keystroke-level model was used in order to test the effectiveness of the tab-based GUI. The Keystroke-level model, commonly known as KLM, is a predictive evaluation model that aims to estimate the execution time of a task by creating an abstract representation of that task [20]. The analysed tasks are represented by the actions that are required in order to perform a function. For the purpose of this project the following abstractions were used:

- "C" to represent the user tapping the screen
- "F" to represent the user finding a desired cross-point by using the scroll bars on each axis of the grid.

Any character encased in square brackets, such as "[F]", portrays an optional operator that may be required under certain circumstances.

Two scenarios were used to test the effectiveness of the user interface, each consisting of the processes as mentioned in section 2.1.2. In both scenarios, the "[F]" variable indicates a user navigating the grid structure for the required cross-point. This variable is optional since its occurrence is related to the number of items on either axis of the grid structures. In some cases, the small screen of the iPhone OS device prevents all items from being

displayed, thus requiring a user to scroll along the items of a grid to find the desired cross-point.

Table 3.1 and 3.2 each summarise their respective scenarios and provide the number of interactions required for the actions in terms of a worst and best case scenario, where the worst case scenario assumes all optional interactions are required to complete a task and the best case scenario excludes any optional tasks.

In both scenarios it is assumed that the device grid of the connection manager view is set to the initial subnet.

### Scenario 1

This scenario involves connecting two end-points that exist on the same subnet.

| Action | Abstract Representation | Total interactions (worst case scenario) | Total interactions (best case scenario) |
|---|---|---|---|
| Select 2 devices | [F]C | 2 | 1 |
| Switch to next tab and set multicore connection | C[F]C | 3 | 2 |
| Switch to next tab and set internal routing for input device | C[F]C | 3 | 2 |
| Switch to next tab and set internal routing for output device | C[F]C | 3 | 2 |
| Total | | 11 | 7 |

Table 3.1: KLM analysis on creating a connection between 2 devices in the same subnet.

### Scenario 2

This scenario involves connecting two end-points that exist on remote end-points. The shaded region represents the task of connecting two routers in a subnet, thus bridging two remote subnets. This action needs to be repeated for every subnet dividing the two desired end-points.

| Action | Abstract Representation | Total interactions (worst case scenario) | Total interactions (best case scenario) |
|---|---|---|---|
| Select cross-point between device and router | [F]C | 2 | 1 |
| Switch to next tab and set multicore connection | C[F]C | 3 | 2 |
| Switch to next tab and set internal routing for input device | C[F]C | 3 | 2 |
| Switch to device grid and switch to next subnet | CC | 2 | 2 |
| Select cross-point between routing devices | [F]C | 2 | 1 |
| Switch to next tab and set multicore connection | C[F]C | 3 | 2 |
| Switch to device grid, change subnet and select cross-point between the router and device | CC[F]C | 4 | 3 |
| Switch to next tab and set multicore connection | C[F]C | 3 | 2 |
| Switch to next tab and set internal routing for output device | C[F]C | 3 | 2 |
| Total | | 25 | 17 |

Table 3.2: KLM analysis on creating a connection between 2 devices in remote subnets.

Looking at the data from Scenario 1, it can be assumed that any connection between two end-points in the same subnet should take between seven and eleven actions to complete. Alternatively, looking at the data from Scenario 2, an equation to describe the action of connecting two end-points is derived in figure 3.4.

```
Number of interactions = Number of compulsory actions + Number of router-to-router connections * Number of actions required to
                                                                                                  create a router-to-router connection

This can be resolved to:

Worst case scenario = 18 + number of router-to-router connections * 7
Best case scenario = 12 + number of router-to-router connections * 5
```

Figure 3.4: Equation describing relationship between the number of subnets and the number of actions required to complete a task.

This attribute of the tab-based layout is demonstrated in figure 3.7. It can be said

that when establishing connections between devices on different subnets, the number of interactions required to create a connection is proportional to the number of subnets that need to be traversed.

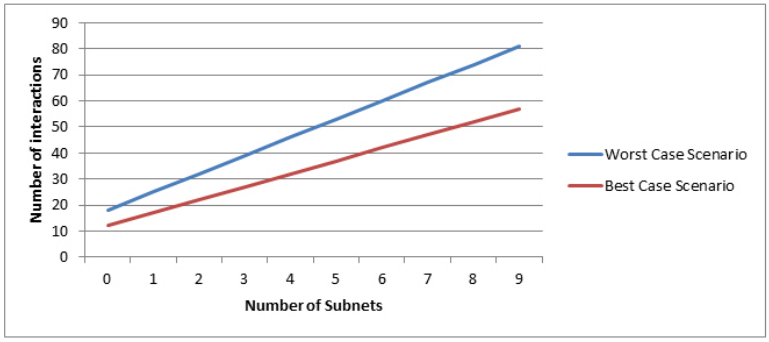


Figure 3.5: Graph showing relationship between the number of subnets in a network and number of actions required to complete an end-to-end connection.

## 3.4 Summary

This chapter provided a description of the process undertaken when porting an application. While the application logic functioned correctly, the user interface provided an unsatisfactory level of usability. It was thus determined that for the prototype application to become feasible, it would require a modification of the application logic to provide a simplified means of controlling an audio network.

# Chapter 4

# Patchbay Analysis

In chapter three it was suggested that the tab-based interface of the Unos Vision application did not provide an effective means of control over an audio network. This chapter explores an alternative "plug" oriented patchbay mechanism to handle the connection management component of Unos Vision.

An overview of connection management systems is discussed, providing some comparisons to the X170 mechanisms for network connection management. Several patchbay designs are described and a grid-based mock up is designed and analysed. This mock up is then compared to the native Unos Vision mechanism for connection management and it is concluded that a patchbay alternative to the Unos Vision application would provide benefits in terms of user interaction.

## 4.1 Patchbays

A patchbay is a system that allows for audio signals to be routed between devices. They are commonly classified into two groups [9]:

- Hardware patchbays
- Software patchbays

In a hardware patchbay system, analogue audio signals are routed using physical cable connections between an audio source, an intermediate patchbay device and an audio destination [9]. In digital audio networks, audio signal routing is managed by a control

application that initiates the connections between audio devices [9]. In terms of patching audio signals, a hardware patchbay has the disadvantage of requiring individual physical cable connections between end-points for each audio signal that needs to be routed, while in a software patchbay a single cable connection may support an array of virtual connections between devices, thus reducing cable clutter [9].

For the purpose of this research project only software-based patchbays will be discussed and from here on patchbays will refer to software patchbays.

### 4.1.1 Plug Oriented Patchbay

Unos Vision currently provides a user with low-level access to an audio network by allowing a user to control the multicore routing between devices as well as the internal routing of devices. The term "plug" oriented refers to the abstraction of multicores away from the user, thus preventing a user from managing the routing of multicore connections. This means that the user controls the end-to-end connections between the plug parameters of devices while the same underlying application logic controls the multicore routing, facilitating the end-to-end routing. This has the advantage of providing a simpler means of controlling a network since the user's focus is shifted to only managing the end-to-end connections in an audio network.

#### Patchbay Variations

Following is a list detailing four common patchbay approaches [9] [16]:

- List-based
- Tree-view-based
- Tree-grid-based
- Graphical

*List-based* patchbays display all the plugs available on an audio network generally using two list structures, one focusing on outputs and the other on inputs as shown in figure 4.1. A user can specify which two plugs to route audio between by selecting an item from each list [16].

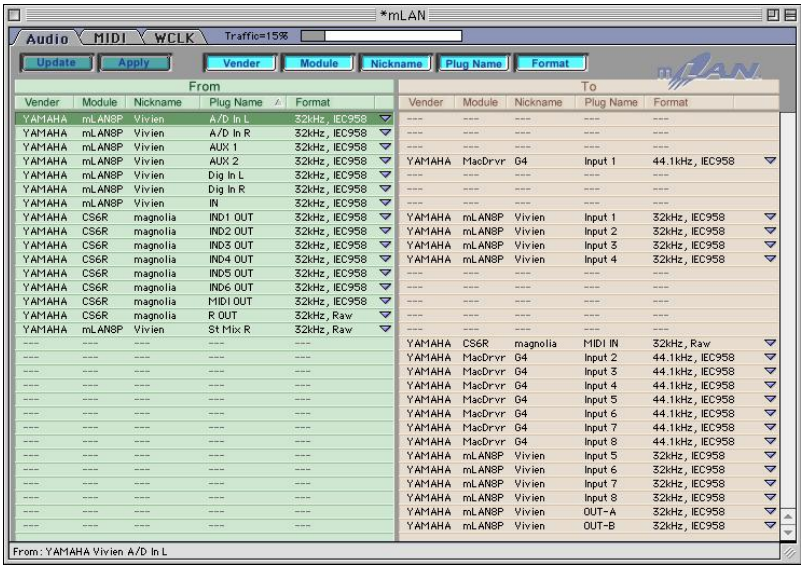


Figure 4.1: mLan List-based Patchbay [16]

Figure 4.2 demonstrates a *Tree-view-based* patchbay. This approach is similar in structure to the list patchbay but instead of listing all the available plugs on devices in an audio network, a collapsible tree-like structure is used to group the plugs by their associated device and each device is grouped by its subnet or bus[9].

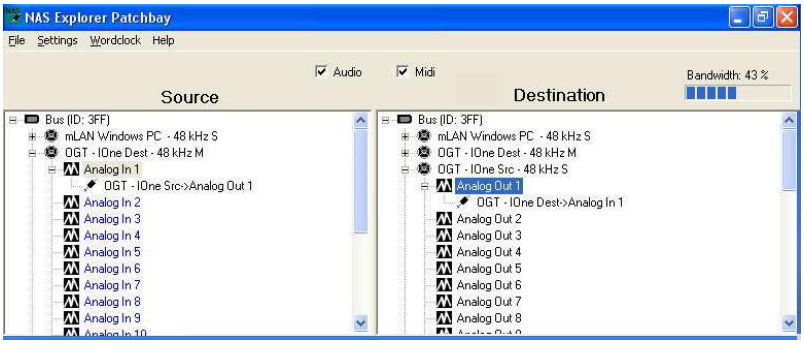


Figure 4.2: NAS Explorer Patchbay [16]

Figure 4.3 demonstrates a *tree-grid-based* patchbay displaying all the busses, devices and a device's associated plugs using a grid structure. Each axis is populated with a tree-view structure containing busses, devices and plugs, where each bus is expandable to reveal the devices in that bus and devices are expandable to reveal all the plugs on that device [9] [16]. The x axis of the grid is used to represent outputs and the y axis represents inputs. A user can specify between which two plugs an audio signal will be routed by selecting a cross-point between the desired two plugs.

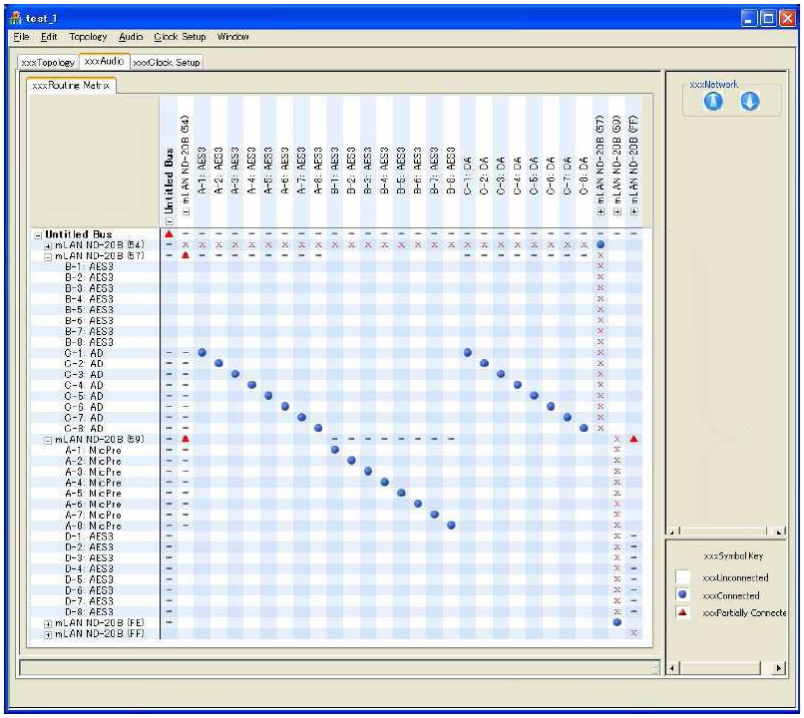


Figure 4.3: Tree-Grid Patchbay [16]

Figure 4.4 demonstrates a *graphical* patchbay that uses a visual representation of each device and its plugs to represent an audio network. In most systems, a user is required to select the desired input and output plugs on the device graphic to specify the end-points of a connection [**?**] [16].

Foulkes [2008] analysed the number of clicks required to navigate to the desired plugs and to create a connection between those two plugs in each of the above approaches. The results are summarised in Figure 4.1.

| | List patchbay | Tree-view list patchbay | Tree-grid patchbay | Graphic patchbay |
|---|---|---|---|---|
| Navigate to plugs | 0 | 6 | 4 | 2 |
| Make connection | 3 | 4 | 1 | 2 |
| Total clicks | 3 | 10 | 5 | 4 |

Table 4.1: Table of results for Foulkes' [2008] study on the number of interactions required to complete a task on various patchbay solutions.

Foulkes [2008] determined that the tree-grid patchbay would be the most optimal solution to audio network connection management even though it is not the most optimal solution in terms of the number of clicks. He goes on to say the tree-grid's benefit comes from not having to select the input and output plugs explicitly when attempting to route an

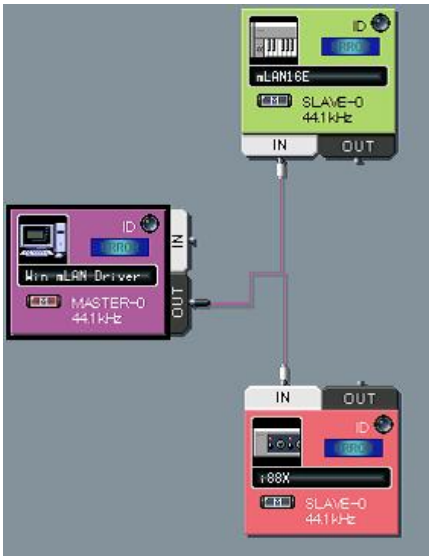


Figure 4.4: Yamaha mLan Graphic Patchbay[16]

audio signal. Figure 4.5 demonstrates this implicit behaviour; by selecting a cross-point between two plugs in the grid structure, a user is specifying both the input plug and the output plug in a single action. The grid structure can be used to display the structure of the network in a logical manner and the cross-points can be used to display the connection status between any two plugs [16].

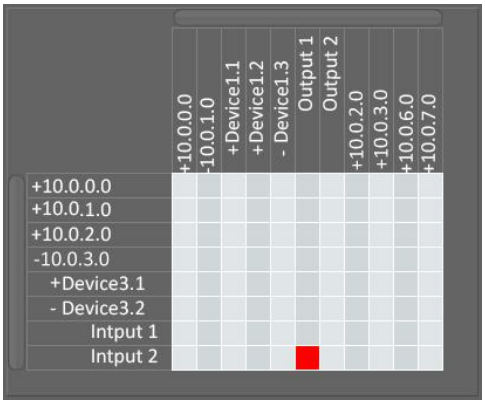


Figure 4.5: Grid-based patchbay mock-up

## 4.2 Evaluation of Patchbay Design

To test the effectiveness of the tree-grid plug oriented patchbay solution the simplified version of the Keystroke-Level Model from section 3.3.1 was used. This allows for a direct

comparison to be made between the plug oriented and Unos Visions native connectivity approach.

### 4.2.1 Scenario

Consider using the plug oriented approach to connection management to fulfil scenarios 1 and 2 section 3.3.1. Both of these scenarios will use the same actions since only a single grid-structure is used and no navigation between tabs is required. This applies to scenarios where a connection is established between two devices in the same subnet and when connections are established between remote subnets.

| Action | Abstract Representation | Total interactions (worst case scenario) | Total interactions (best case scenario) |
|---|---|---|---|
| Scroll to and expand the subnets' cross-point | [F]C | 2 | 1 |
| Scroll to and expand the devices' cross-point | [F]C | 2 | 1 |
| Scroll to and select the plugs' cross-point | [F]C | 2 | 1 |
| Total | | 6 | 3 |

Table 4.2: KLM analysis on tree-grid patchbay.

Using the data from table 4.2, the worst case scenario for routing audio between any two end-points requires six actions from the user and in the best case should require three actions. Figure 4.6 demonstrates this behaviour and combines the data from table 3.2 so that a comparison can be made with the native Unos Vision design.

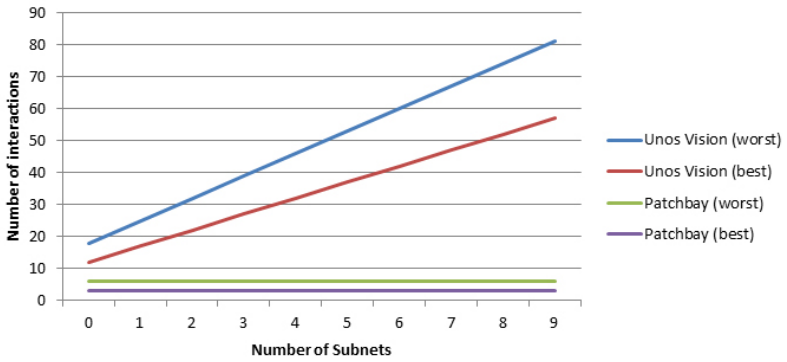


Figure 4.6: Graph comparing tree-grid patchbay to the native Unos Vision design.

Figure 4.6 shows that, even in small networks consisting of single subnets, the plug oriented patchbay approach requires fewer interactions form the user to effectively manage

an audio network. One scenario omitted from figure 4.2 is that of managing the routing between devices in the same subnet. In section 3.3.1, it was said that this scenario would require seven interactions for the best case scenario and eleven in the worst, which still compares unfavourably to the plug oriented approach.

## 4.3 Conclusion

In terms of interaction design and by using predictive evaluation methods it can be concluded that a plug oriented patchbay design using a tree-grid approach provides a higher level of usability when compared to the traditional design of the Unos Vision tab-based layout. A grid-based plug oriented approach is able to give an end user an overview of an audio network and can simplify the management of an X170 audio network by limiting the number of actions required to manage the routing of audio signals between end-points. The disadvantage is that it does not provide the same level of control over multicore utilization.

# Chapter 5

# Patchbay Implementation

In chapter four it was suggested that a plug oriented patch bay system, when compared to the native tab-based layout of the Unos Vision application suite, would provide a more satisfactory means to control an audio network. This chapter explores extending the prototype application developed in chapter 3 through the addition of a patch bay component. The focus of the prototype will be on the low-level logic requirements of a patchbay system to determine whether an end-to-end connectivity solution based on the X170 protocol is possible.

Initially a list of requirements relating to a patchbay system is discussed, detailing some mechanisms that need to be implemented to make the X170 protocol function as an end-to-end connectivity solution. A pseudo code solution is developed to aid the integration process of the patchbay logic and the structure of the proposed patchbay solution is diagrammatically explained using the UML modelling techniques discussed in section 2.4.3. Finally, the patchbay design is implemented so that any issues regarding the programming of such a system are exposed.

For the purpose of this project, the prototype was developed as a proof-of-concept application designed as an alternative means of connection management. It was therefore limited to only function within a single subnet. A basic algorithm on which to base a potential cross-subnet implementation is proposed, but it lacks the functionality of a full plug-oriented patchbay system and is not a feasible solution.

## 5.1 Goals

The goals of this prototype can be summarised into the following points:

- Identify the requirements of a patchbay that uses the X170 protocol to manage a network.
- Identify any shortcomings of the X170 protocol when implementing it as a patchbay solution.
- Identify any issues with the Unos Vision application, Juce application framework and the XFNDLL component when implementing a patchbay solution.

## 5.2 Patchbay Design

This section defines the logic and design of a patchbay component concentrating on theoretical concepts. The basic requirements of a patchbay system are defined and modified to complement the Unos Vision application. A possible algorithm is proposed and a model is generated to define where the Unos Vision needs to be modified to prove patchbay capabilities.

Following is a proposed list of high-level requirements that a grid-based patchbay application should be able to perform [16]:

- Establish audio connections between device plugs.
- Break audio connections between device plugs.
- View network topologies at different levels (Network, Bus and Device levels).

Figure 5.1 displays an updated version of the use case diagram in chapter 2 that reflects the above requirements. Due to the difference in the way that Unos Vision and the plug oriented patchbay solution manage connections, the *"Make connections"* and *"Break connections"* use cases are considered to be different from the connection management approach of a plug oriented connectivity solution, hence the additions of a *"Make Plug Connection"* and *"Break Plug Connection"* use cases.

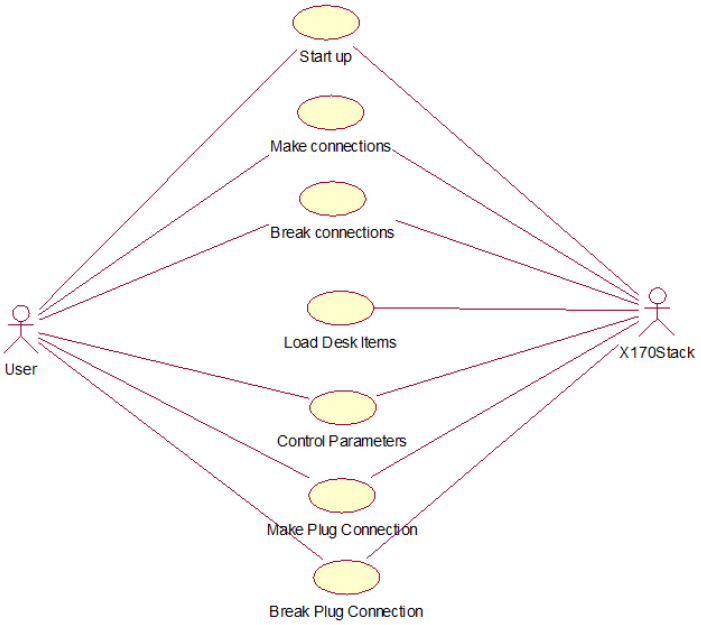


Figure 5.1: Updated version of Unos Vision use case including patchbay components.

### 5.2.1 Patchbay Algorithm

In chapter 2 the X170 protocol was discussed and it was mentioned that the protocol provided no means of establishing a connection directly from an audio input to an audio output between remote devices. The user needs to first establish a multicore connection between the devices before an audio signal is routed. It is for this reason that a list of requirements needs to be compiled relating to the special mechanisms required for managing multicore connections.

The following considerations need to be made when implementing a plug oriented patchbay that uses the X170 protocol:

- Multicores need to be managed by the patchbay application since the user only interacts with audio inputs and outputs.
- If there are no active multicore connections between two devices that need to be patched, a new multicore connection needs to be created. If there is an active multicore connection, that multicore should be reused. This limits the number of active multicore connections between any two devices.
- If a multicore connection exists between two devices and there are no active multicore sequences in that multicore, that multicore needs to be disconnected. If two

end-points are being disconnected and there are still active multicore sequences in that multicore, the multicore must not be disconnected. This limits the number of inactive multicore connections in an audio network and prevents any multicore connections from being unintentionally disconnected while an audio signal is still being routed.

- If an error occurs during the multicore connection process or while managing the internal routing of a device, any changes to the multicore connections between devices and to the internal routing of devices need to be reverted. This prevents any unintentional changes to a network without the user's knowledge.

Annex 5.2, 5.3 and 5.4 describe a basic algorithm for providing end-to-end connection management in an X170 network. These algorithms do not indicate any form of error handling, so if an error occurs during the patching process and changes made to the network will persist until the errors are manually fixed.

There are three scenarios to handle when it comes to patchbay routing.

- Connecting an audio input on a device to an audio output on that same device
- Connecting an audio input on a device to an audio output on a remote device on the same subnet
- Connecting an audio input on a device to an audio output on a remote device on a remote subnet

Figure 5.2 provides a pseudo code algorithm for handling these three scenarios.

```
If the plugs are on the same device
    Handle internal routing
Else if the plugs are on remote devices, but on the same subnet
    Handle device routing
Else if the plugs are on remote devices on remote subnets
    Handle remote routing
```

Figure 5.2: Pseudo code handling the 3 possible connection scenarios.

When a connection is required between two audio pins on the same device a simple X170 message can be constructed to handle the procedure. There is no need for a complex routing algorithm since an audio signal can be streamed without the need of a multicore

connection. This means that scenario 1 can be achieved in a single action; construct an X170 connect message using the specified audio pin addresses.

Figure 5.3 describes the process for creating a connection between audio pins on remote devices that are on the same subnet, as described in scenario 2.

```
if there is an active multicore between the two device
    Set it as the current multicore
else
    create a new multicore and assign it as the current multicore
go through all the sequences of the current multicore on the source device
    if a multicore sequence is unused
        go through all the sequences of the current multicore on the destination device
          if a multicore sequence is unused
            send x170 message patching source plug and multicore sequence
            send x170 message patching destination plug and multicore sequence
```

Figure 5.3: Pseudo code handling the routing of an audio signal between 2 devices on the same subnet.

Figure 5.4 describes the process for creating a connection between audio pins on remote devices that are on remote subnets as described in scenario 3. This section is subdivided into for smaller components:

1. Connect the source device to a router
2. Find a route between routers until the destination subnet is found
3. Connect the final router to the destination device
4. Connect the audio pins on both devices to the same multicore sequence

### 5.2.2 Object Model and Sequence Diagram

Figure 5.5 demonstrates the objects with which the patchbay component will need to interact. The objects have the following functions:

The *MainWindow* object is responsible for the overall application layout and behaviour. All components, including the *NetworkView* tab and its children tabs are added to the application's interface in this class.

```
(1)
Go through all the routers on the source subnet
    Get the routers routing table
    If the destinations subnet is in the routing table
        If there is an active multicore between the source device and the router
            Set it as the source multicore
        else
            create a new multicore and assign it as the source multicore
(2)

Go through all the routers on the current subnet
    Get the router's routing table
    If the destinations subnet is in the routing table
        If there is an active multicore between the source and the destination
            Set it as the destination multicore
        else
            create a new destination and assign it as the destination multicore
Repeat (2) until you have reached the destination subnet

(3)
If there is an active multicore between the destination device and the router
    Set it as the current multicore
else
    create a new multicore and assign it as the current multicore

(4)
go through all the sequences of the source multicore on the source device
if a multicore sequence is unused
    go through all the sequences of the destination multicore on the destination device
        if a multicore sequence is unused
            Send x170 message patching source plug and source multicore sequence
            Send x170 message patching destination plug and destination multicore sequence
```

Figure 5.4: Pseudo code handling the routing of an audio signal between 2 devices on remote subnets.

The *NetworkView* object is responsible for the layout of the *Connection Manager* tab in figure 2.7 and contains the method responsible for grid cross-point button clicks. This is the class that contains the patchbay connection logic.

The *PatchBayMatrix* inherits from the *MatrixComponent* and contains the code to represent the patchbay grid object. This is the class that is responsible for representing the devices on the *MatrixComponent* and contains the logic for expanding and collapsing a subnet or device group.

The *Network* object is a singleton object that represents the network. This class allows access to the devices on a network.

The *Bus*, *Device*, *MulticoreSocket* and *DeviceStateMatrix* represent the subnets, devices, multicore sockets and the internal routings of a device respectively.

Figure 5.6 displays a sequence diagram outlining the basic flow of a grid button click. This diagram has been simplified to abstract any interactions with lower-level classes

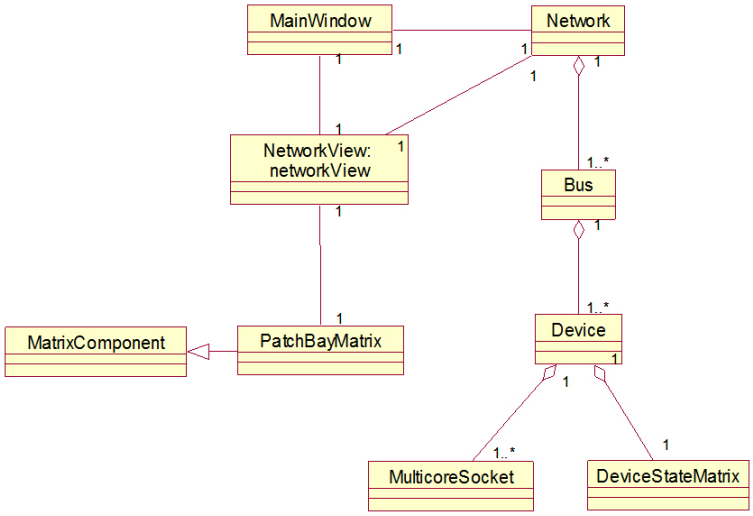


Figure 5.5: Object model of Unos Vision patchbay prototype.

and entities, such as the UMANDLL and XFNDLL components, since they have been described in chapter 2.

## 5.3 Implementation

This section concentrates on the integration and implementation of a patchbay component into Unos Vision. The structure of the application will be based on the design listed in section 5.2 and will be used as a prototype to demonstrate the feasibility of a plug oriented patchbay component in the Unos Vision application.

### 5.3.1 Tree-Grid Structure

In chapter 4 it was concluded that a tree-grid approach to connection management provides a more effective means of managing connections when compared to the native Unos Vision design. The *MatrixComponent* class is able to provide some of the functionality of a tree-grid design.

The *MatrixComponent* handles the process of creating a grid object which consists of a matrix of buttons. Each button has a toggle state which is used to track the current status of a cross-point. The class inheriting from the *MatrixComponent* needs to pass two component arrays to the *MatrixComponent*, one for the grid row and the other for

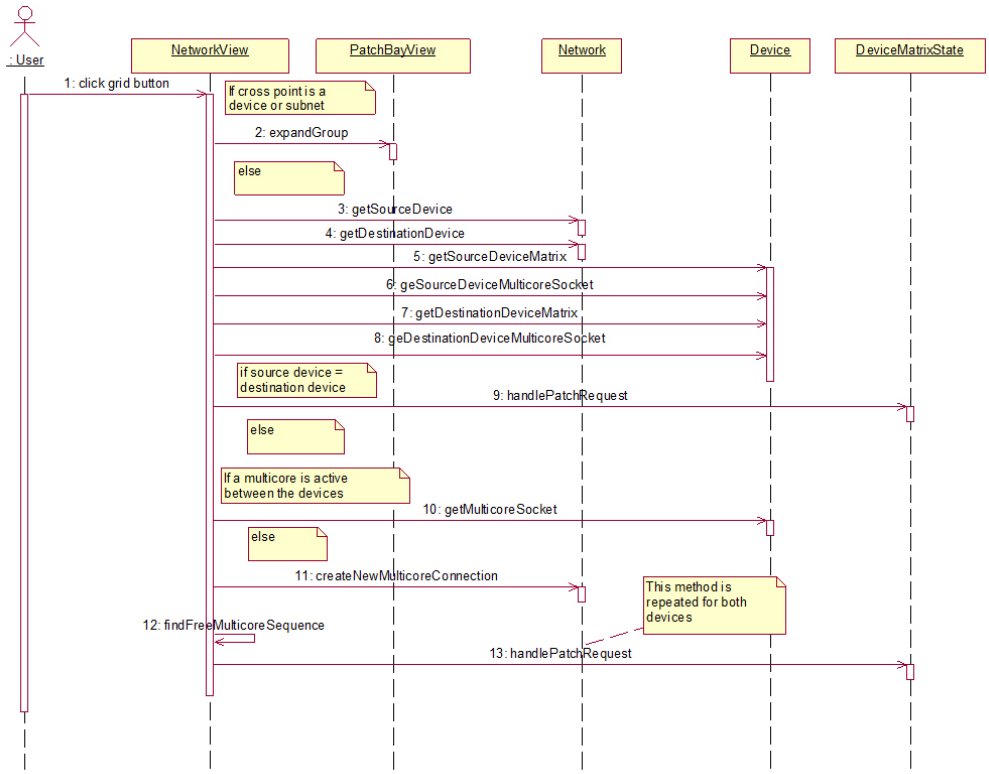


Figure 5.6: Sequence diagram of a patchbay grid button click.

the grid column, each consisting of labels. The *MatrixComponent* then creates a grid based on these arrays and adds button listeners to each cross-point. This listener passes the matrix object containing the cross-point, the button clicked and the row and column index of the button to the *gridButtonClicked()* method in the *NetworkView*.

The *MatrixComponent* has no mechanisms for providing a tree-grid-based grid approach so the *GridMetaData* class in annex A.1 was created. The *GridMetaData* class describes each individual row and column in terms of the group it represents, its index in its parent's group and whether it is in a collapsed state. When a cross-point is clicked, the *NetworkView* class can determine the type of the cross-point as demonstrated in annex A.2. If the cross-point is neither an audio input nor an audio output the group is expanded, else the *patchBayConnect()* method is called.

The logic that expands and collapses a group is contained in the *PatchBayMatrix* class and is demonstrated in annex A.3. The *toggleGroup()* method uses the *GridMetaData* class to traverse the array that contains the labels for each axis. Once it has found the row or group to expand, it retrieves the device or subnet object from the network class

and adds its children to the array. The current grid object is then cleared and reinitialised using the new arrays containing the added children. Collapsing groups work in a similar fashion except that labels are removed from the arrays.

The initial populating of the grid is done by iterating through the network object and adding each subnet to the row and column array in the *PatchBayMatrix* class, as shown in annex A.4.

### 5.3.2 Connection Management

The connection management component of the patchbay is handled in the *NetworkView* class in the *gridButtonClicked()* method. As with the *toggleGroup()* method, the connection management component uses the *GridMetaData* class to parse the network object. In annex A.4 the *GridMetaData* class is used to get the indices of the source and destination subnets, devices and audio pins with respect to its parent *Network*, *Subnet* and *DeviceMatrixState* class respectively. Shown in annex A.5, these values are passed to the *patchBayConnectJob()* method which determines if the connection is an internal routing job, a routing between two devices on the same subnet or a routing between two devices on different subnets. The application determines whether a connection is being created or destroyed by checking the current status of the cross-point.

The procedure for handling the internal routing job is simple as it requires no routing logic. Annex A.6 demonstrates the procedure for handling such an event using the *DeviceMatrixState*'s object from the source device to call the native *handlePatchRequest()* method.

The handling of the routing between devices in the same subnet becomes more complex when compared to the internal routing mechanism and as such is divided into two main components; selecting the correct multicore connection between the two devices and selecting a common, inactive multicore sequence between the two devices.

Annex A.7 shows the establishing of the multicore connection between the two devices. The first part of this procedure iterates through all the multicore sockets on the source and destination devices in an attempt to find an already established multicore. If this fails, a new multicore connection is created. A *wasMcCreated* Boolean variable allows the application to track changes made to the network so that any changes may be reverted in the case of a patch job failing. The *setPatchBayMulticore()* method is responsible for

the actual creation of the multicore stream between the two devices and if this method fails, a false value is returned to indicate this failure to the calling function.

Following Annex A.7, annex A.8 demonstrates the high-level process of managing the internal routing of both devices. The *wasMcCreated* variable is used to undo any previously modified multicore connections if the *setPatchbayInternalRouting()* method fails to correctly handle the internal routing of both the source and destination devices. The *setPatchBayMulticore()* method is called using the inverse of the *createCon* variable to revert the changes.

Interacting with *Subnet*, *Device* and *MulticoreSocket* objects has been achieved by using the inbuilt indexing system. This index allows:

- A multicore socket to be accessed relative to the other multicores in a device.
- A device to be accessed relative to the other devices on a subnet.
- A subnet to be accessed relative to the other subnets on a network.

In addition to this indexing feature, each *Subnet*, *Device* and *MulticoreSocket* object has an alternative means of being identified. For example, an application is able to identify whether a Device object refers to a router or whether a *MulticoreSocket* is an input or an output. The *DeviceMatrixState* has no such facilities. The *DeviceMatrixState* is a simple grid-like structure that contains two arrays, each describing a row or column using a string object, and a set of cross-points representing the state of the internal connections on a device. This means that there is no explicit means of distinguishing a specific cross-point as being either a multicore sequence or an audio input or output.

To define the cross-points in a *DeviceMatrixState* an application needs to use some form of string comparison or regular expression to identify the column or row type. For the purpose of patching multicore sequences to audio inputs and outputs, a method using string comparisons was used. This was because only multicore sequences need to be iterated to connect a user selected audio input or output to an application selected multicore.

Annex B9 shows the procedure for identifying a multicore sequence in a *DeviceMatrixState* using a string tokenizer and direct string comparisons. A multicore sequence in a *DeviceMatrixState* takes the form of "MulticoreA B", where the first nine characters indicate that the cross-point belongs to a multicore, the "A" represents the multicore socket index and "B" represents the multicore sequence number.

Once a multicore sequence in a *DeviceMatrixState* object has been identified the code in annex A.10 is executed which checks if the point is already connected to the desired audio input or output. If it is not, a connection is created by calling the *handlePatchRequest()* method of the *DeviceMatrixState* class. Annex B.10 also demonstrates a mechanism for reverting changes if the process does not complete. The *wasConCreated*, *srcConCreated* and *destConCreated* Boolean variables monitor the state of the connection creation process allowing a decision to be made as to what changes need to be reverted to return the internal routing to its previous state.

Once the multicore connection has been established and the internal routing has successfully been patched, the *patchConnectJob()* method returns a true value is returned to the *gridButtonClicked()* method in the *NetworkView* class which then toggles the state of the patchbay grid button. Figure 5.7 displays the final product running on an iPhone OS device.

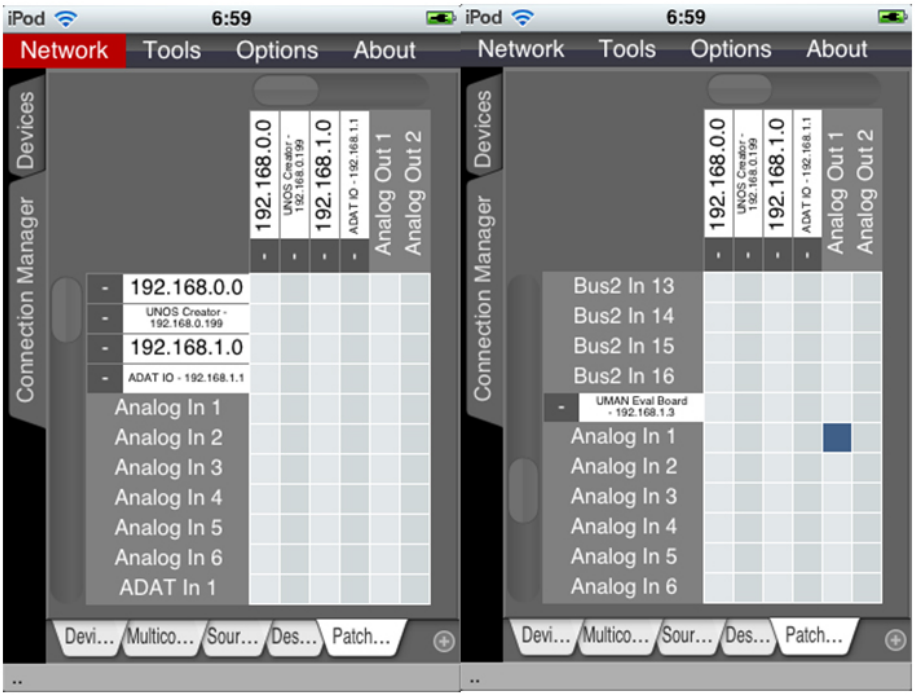


Figure 5.7: Screenshots of the final patchbay implementation running on an iPhone OS device.

## 5.4 Limitations of the Prototype

The discussed implementation of the patchbay prototype does not provide a real-time view of the network status. On start-up, the application does not pre-scan the network to check for any active patchbay connections between audio pins. Any changes made to the network by any other client applications on an audio network are not detected by the prototype.

The prototype does not allow for the routing of audio signal across subnets and is limited to routing between audio pins in the same subnet.

The prototype has a heavy reliance on the desktop implementation of Unos Vision. Various attributes of audio networks, such as the establishing of clock sources and sampling frequencies, need to be set by a desktop version of Unos Vision as the iPhone OS prototype has no mechanisms for setting these options.

When destroying connections, the prototype does not check for any active sequences in a multicore socket and thus does not destroy a multicore connection between two devices. This will leave unused multicore connections in an audio network.

## 5.5 Testing

The prototype application was designed to test the possibility of implementing automatic routing mechanism between devices in the same subnet. A simple audio network consisting of two evaluation boards, a wireless router, the iPhone prototype application and a Unos Vision desktop client was created to test the functionality of the prototype. Figure 5.8 shows the topology of the audio network. The router in figure 5.8 refers to a standard Ethernet router with wireless capabilities and not an X170-based router.

### 5.5.1 Testing Scenarios

The following list contains the high-level functionality that was tested:

- The ability to correctly represent a network in a grid environment.
- The ability to create a patch job between two audio pins on the same device.

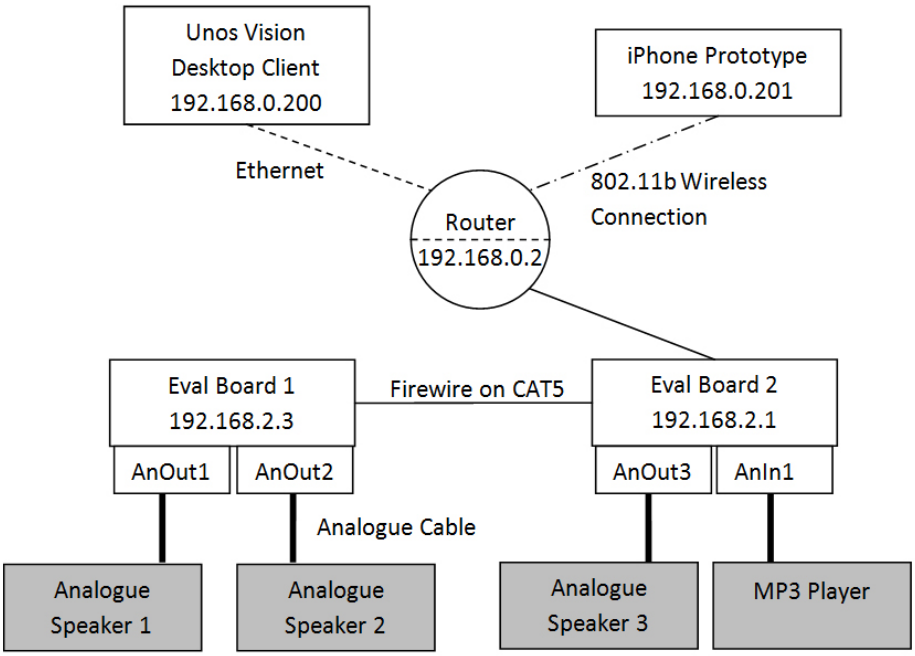


Figure 5.8: Network topology of the testing environment.

- The ability to create a patch job between two audio pins on the remote device.

To test the above functions four scenarios were defined:

1. Load the patchbay program from a fresh instance. This will determine if the prototype application scans the network and correctly represents the devices.
2. Create and destroy an internal connection from the MP3 player to Analogue Speaker 3. This would test the ability to manage the internal routing capabilities of the prototype.
3. Create and destroy a connection from the MP3 player to Analogue Speaker 1. This would test the ability to manage the internal routing mechanisms as well as the multicore management of the prototype.
4. Create a connection from the MP3 player to Analogue Speaker 1 and create and destroy a connection between the MP3 player and Analogue Speaker 2. This would test the ability to manage the internal routing mechanisms as well as the concept of multicore reuse when dealing with multicore management.

**Expected Results**

1. The patchbay grid should display the network topology in a grid-like structure.

2. When the cross-point between AnIn1 and AnOut3 is selected no multicore connections should created and audio should stream from the MP3 player to analogue speaker 3. When the cross-point is selected for the second time, the connection between AnIn1 and AnOut3 should be destroyed and audio should not be streamed.

3. When the cross-point between AnIn1 and AnOut1 is selected a multicore connection between the two devices should be created, the internal routing of each device should be correctly set and audio should be streamed from the MP3 player to analogue speaker 1. When the cross-point is selected for the second time, it is expected that the connection between AnIn1 and AnOut1 is destroyed and audio is no longer streamed. Since no mechanisms have been implemented for destroying multicore connections when no streams are active, it is not expected that the multicore connection will be destroyed.

4. For scenario 4, it is expected that when the cross-point between AnIn1 and AnOut1 is selected, a multicore connection between the two devices will be created, the internal routing of each device will be correctly set and audio will be streamed from the MP3 player to analogue speaker 1. When the cross-point between AnIn1 and AnOut2 is selected the multicore connection between the two devices will be reused, the internal routing of each device will be correctly set and audio will streamed from the MP3 player to analogue speaker 1 and analogue speaker 2. When the cross-point between the AnIn1 and AnOut2 is selected for the second time, the connection between AnIn1 and AnOut2 will be destroyed and audio will not be streamed to Analogue Speaker 2. The multicore connection and the internal routing of the Eval Board 2 should remain unchanged, and the internal routing from AnOut1 to a multicore sequence should remain unchanged.

To confirm the behaviour of the application and measure the results the Unos Vision Desktop Client was used as a monitoring tool to scan the multicores in the network and the internal connections of the devices. For a scenario to pass its behaviour would have to match that of the expected results listed above.

**Results**

- The prototype successfully displayed the network structure in a tree-like structure.

- A connection was successfully established between the two audio pins and audio was played from Analogue Speaker 3. No additional multicore connections were created. On disconnection of the audio pins the stream was halted and Analogue Speaker 3 stopped playing audio.

- A connection was successfully established between the two devices and audio was allowed to stream from the MP3 Player to Analogue Speaker 1. When destroying the connection, the audio stream was discontinued and Analogue Speaker 1 stopped playing audio. The internal connections were destroyed, but the multicore connection remained.

- The first connection action completed successfully resulting in audio being streamed from the MP3 player to Analogue Speaker 1. When the connection between AnIn1 and AnOut2 was established, audio continued being streamed to AnOut1 and was now being streamed to AnOut2, resulting in both Analogue Speaker 1 and Analogue Speaker 2 playing audio. No additional multicore connections were created and both streams was properly routed using a single multicore connection. When destroying the connection between AnIn1 and AnOut2, the multicore connection was left intact and the internal routing of Eval Board 2 was left unchanged. Analogue Speaker 1 still played audio and the internal routing of Eval Board 1 was correctly modified to stop the streaming of audio between AnIn1 and AnOut2.

## 5.6 Summary

This chapter summarised the process for designing and implementing a simple plug oriented patchbay application interacting with audio devices using the X170 protocol. Through the use of various UML diagramming techniques and pseudo code algorithms, a basic model was developed. This model provided the basis on which to structure a prototype application based on the Unos Vision Application.

When testing the application it was found that it provided sufficient mechanisms for creating connections within a single subnet. The application did however have issues of not destroying inactive multicore connections resulting in the possibility of a network becoming fragmented. This issue of fragmentation may become more apparent in networks consisting of a large amount of devices.

The ability to route multicores across subnets has been discussed with a basic starting algorithm proposed, but more work needs to be done to develop an algorithm that will handle all the requirements of cross-subnet multicore routing.

# Chapter 6

# Conclusion

The main objective of this research project was to design a portable audio networking application that will provide an audio engineer the flexibility of being able to control an audio network from any location. The application needed to represent the topology of an audio network and be able to manage the connections and devices within that network.

A prototype based on a desktop application was developed for the iPhone OS. The prototype was analysed using a predictive analysis technique which resulted in the conclusion that the application interface and logic needed to be redesigned. After analysing alternate methods of network connectivity a new prototype was designed and implemented. After comparing the two prototype applications it was discovered that the redesigned prototype provided better mechanisms for controlling audio networks, but lacked the technical functionality to completely replace a desktop-based audio connectivity solution.

## 6.1 Outline

It was decided to base the application off of an existing audio network application due to the complexities and the vast requirements of an audio networking application. For this purpose the Unos Vision application suite was chosen. In chapter 2 the capabilities of Unos Vision and its structure is discussed, providing descriptions on the underlying X170 protocol that enable Unos Vision to communicate with audio devices. During the analysis, it was discovered that Unos Vision is based on the Juce application framework which provides support for cross-platform compilation. One of the platforms supported

by Juce is the iPhone OS, which motivated the decision of using the iPhone OS as the platform for the mobile application.

Chapter 3 describes the process for porting the Unos Vision application to the iPhone OS to create an initial prototype. This prototype was used to identify weaknesses in the design of Unos Vision in situations where screen space is limited. It was concluded that the native design of Unos Vision provided an unsatisfactory means with which to control an audio network as the interface was too complex for a small-screen device.

Chapter 4 explored the possibility of modifying the Unos Vision application to provide a "plug" oriented patchbay solution. Various connectivity methods were analysed and a tree-grid patchbay was chosen. This design was then compared to the native Unos Vision design and it was found that the patchbay solution provided a more optimal means of control over an audio network when using small-screen devices.

Chapter 5 discusses the design, integration and testing of a tree-grid patchbay solution based on the Unos Vision application. A simple prototype that handled the routing of audio signals within a subnet was built so that any immediate issues with the patchbay logic concerning the X170 protocol could be exposed. It was concluded that a patchbay solution using the X170 protocol is possible, but extensive work needs to be done to provide an optimal method for routing audio signals across subnets.

## 6.2 Success of the Project

The objective of this project was to provide a mobile audio network connectivity solution that would provide a high level of control over an audio network. By analysing the results from section 5.5 it can be determined that this objective has been partially achieved. The final prototype provided control over an audio network as, but the establishing of end-to-end connections was limited to patching audio pins that were in the same subnet. Additionally, omissions of some connection management rules in the prototype's logic meant that multicore connections managed by the prototype could become fragmented. Some solutions were proposed detailing how to improve the prototype but further work is still required to develop an optimal mobile application that can substitute a desktop application.

## 6.3 Extensions

**Optimisation of the Unos Vision GUI**

Many of the features of Unos Vision are inaccessible due to the touch-based interaction of the iPhone OS. The prototype application needs to be analysed from a Human-Computer Interaction perspective so that the user interface can be optimised for the iPhone OS and other touch-based platforms.

**Removal of desktop application dependency**

The prototype application has a heavy reliance on a desktop application setting up the various aspects of an audio network, such as clock synchronization between devices, before being able to establish connections between devices. The backend of the application requires further work to optimise the connectivity aspects of the application so that it no longer relies on the desktop version of Unos Vision.

**Extending the routing algorithm**

The prototype is unable to route audio signals across subnets. An algorithm may be developed to allow for this functionality keeping in mind the requirements mentioned in chapter 5.

# Bibliography


[1] Apple Inc. *About Xcode and iPhone SDK*, February 2010.

[2] Apple Inc. *iOS Application Programming Guide*, August 2010.

[3] Apple Inc. *iPod Touch Specifications Sheet*, Online : www.apple.com [Accessed : 21/09/10].

[4] Ballard, B. *Designing the Mobile User Experience.* John Wiley & Sons Ltd, 2007.

[5] Banavige, M. *Preprocessor Directives - Design Practice*, Online : http://wiki.asp.net/page.aspx/596/preprocessor-directives—design-practice/, [Accessed 15/04/2010].

[6] Chigwamba, N. Email correspondence - internal patching of x170 routing devices, April 2010.

[7] Foss, R. Aes informative document for the standard for audio applications of networks - integrated control, monitoring, and connection management for digital audio and other media networks abstract, December 2009.

[8] Foss, R. Aes standard for audio applications of networks - integrated control, monitoring, and connection management for digital audio and other media networks, December 2009.

[9] Foulkes, P. A grid based approach for the control and recall of the properties of ieee1394 audio devices. Master's thesis, Rhodes University, 2008.

[10] Fowler, M. *UML Distilled A Brief Guide to the Standard Object Modeling Language*, third edition ed. Addison Wesley, 2003.

[11] James Kurose, K. R. *Computer Networking - A Top-Down Approach.* Addison Wesley, 2008.

[12] MSDN. *The #include Directive*, Online : http://msdn.microsoft.com/en-us/library/36k2cdd4(VS.80).aspx, [Accessed 08/04/2010].

[13] Richard Foss, Robby Gurdan, B. K. N. C. An integrated connection management and control protocol for audio networks. In *Audio Engineering Society* (October 2009).

[14] Richard Foss, R. G. *Unos Creator User Manual*. Universal Media Access Networks GmbH, Martinstrae 48-50 40223 Dsseldorf Germany, February 2010.

[15] Richard Foss, R. G. *Unos Vision User Manual*. Universal Media Access Networks GmbH, Martinstrae 48-50 40223 Dsseldorf Germany, March 2010.

[16] Sibanda, P. Connection management applications for high-speed audio networking. Master's thesis, Rhodes University, 2007.

[17] Storer, J. *Juce Readme*. Raw Materials Software, April 2010.

[18] Storer, J. *Juce Home Page*. Raw Material Software, Online : http://www.rawmaterialsoftware.com/jucefeatures.php, [Accessed 08/06/2010].

[19] Storer, J. *Juce forums - Rotate the Application for iPhone/iPad*, Online : http://www.rawmaterialsoftware.com/viewtopic.php?f=4&t=5325 [Accessed : 07/07/10].

[20] Thinyane, H. Predictive evaluation. Human-Computer Interaction Presentation, April 2010.

# Appendix A

# Patchbay Source Code Samples

## A.1 GridMetaData Class

```
class GridMetaData
class GridMetaData
{
public:
    GridMetaData(int _devType, int _index);
    ~GridMetaData(void);

    void toggleCollapsed();
    void setCollapsedStatus(bool _collapsed);
    bool getCollapsedStatus();
    int getType();
    int getIndex();

private:
    int deviceType /*0 = subnet/b 1 = Device 2 = IOSocket */
        ,index;
    bool isCollapsed;
};
```

## A.2 gridButtonClicked() method in the NetworkView class

```
void NetworkView::gridButtonClicked(UMANMatrix* source, Button* button, const int row, const int column) {
    if (source == this->GetPatchBayMatrix()){
        PatchBayMatrix * tempPBM = this->GetPatchBayMatrix();
        if (tempPBM->getCPType(row,true) != 2 || tempPBM->getCPType(column,false) != 2){
                    if (tempPBM->getCPType(row,true) != 2)
                        tempPBM->toggleGroup(row,true);
                    if (tempPBM->getCPType(column,false) != 2)
                        tempPBM->toggleGroup(column,false);
                }
        else{
            if (patchBayConnectJob(tempPBM->getParentBusIndex(row, true)
                                   , tempPBM->getParentDeviceIndex(row, true)
                                   , tempPBM->getCPIndex(row,true)
                                   , tempPBM->getParentBusIndex(column, false)
                                   , tempPBM->getParentDeviceIndex(column, false)
                                   , tempPBM->getCPIndex(column, false)
                                   , !button->getToggleState()))
                                   button->setToggleState(…);
        }
    }
    (...)
}
```

## A.3 Code sample from toggleGroup() method

```
void PatchBayMatrix::toggleGroup(int index, bool isRow){
    clearMatrix(false);
    int tempIndex = index;
    Array<GridMetaData*> tempArray;
    Array<ComponentCollectionComponent*> tempLabels;
    int orientation = 0;
    if (isRow){
        tempData = rowData;
        tempLabels = rowLabels;
    }
    else {
        tempData = columnData;
        tempLabels = columnLabels;
        orientation = -90
    }
    Network* network = Network::getInstanceWithoutCreating();
    if (tempData[index]->getType() == PatchBayMatrix::BusPT){
        Bus* currentBus = network->GetBus(tempData[index]->getIndex());
        if (tempData[index]->isCollapsed()){
            for (int deviceIndex = 0; deviceIndex < currentBus->GetDeviceCount(); deviceIndex++){
                Device* currentDevice = currentBus->GetDevice(deviceIndex);
                String labelText = currentDevice->GetName();
                RotatableLabel* rl = createLabelForGrid(labelText, labelText, true, orientation);
                tempIndex++;
                tempLabels.insert(tempIndex, rl);
                tempData.insert(tempIndex, new GridMetaData(PatchBayMatrix::DevicePT,deviceIndex));
            }
        }
        else{
            int i = index+1;
            while(tempData[i]->getType() != 0){
                tempLabels.remove(i);
                tempData.remove(i);
                i++;
            }
        }
    }
    else if (tempData[index]->getType() == PatchBayMatrix::DevicePT){
    (…)
    }

    if (isRow)
        rowData = tempData;
    else
        columnData = tempData;
    setUp(rowLabels, columnLabels);
}
```

## A.4 Populating of Grid object

```
void PatchBayMatrix::setPatchBayMatrix(Network* network){
    if (network == NULL)
        return;
    rowData.clear();
    rowLabels.clear();
    columnData.clear();
    columnLabels.clear();
    setRowHeading(String::empty);
    setColumnHeading(String::empty);
    for (int busIndex = 0; busIndex < network->GetNumberOfBuses();busIndex++){
        Bus* currentBus = network->GetBus(busIndex);
        String labelText = currentBus->GetSubnetIPString();
        RotatableLabel* rl = createLabelForGrid(labelText, labelText, true, 0);
        rowLabels.add(rl);
        rowData.add(new GridMetaData(PatchBayMatrix::BusPT,busIndex));
        rl = createLabelForGrid(labelText, labelText, false, 0);
        columnLabels.add(rl);
        columnData.add(new GridMetaData(PatchBayMatrix::BusPT,busIndex));
    }
    clearMatrix(false);
    setUp(rowLabels, columnLabels);
}
```

## A.5 Code sample distinguishing the different patch job types

```
bool NetworkView::patchBayConnectJob(const int srcBusIndex
                                     , const int srcDevIndex
                                     , const int srcIOIndex
                                     , const int destBusIndex
                                     , const int destDevIndex
                                     , const int destIOIndex
                                     , bool createCon) {
Device * srcDevice = network->GetBus(srcBusIndex)->GetDevice(srcDevIndex);
Device * destDevice = network->GetBus(destBusIndex)->GetDevice(destDevIndex);
if (srcDevice == destDevice){
        Handle internal routing logic
    }
else if (srcBusIndex == destBusIndex){
        Handle single subnet routing
    }
else{
        Handle cross-subnet routing
    }
}
```

## A.6 Code to establish a connection between 2 audio pins on the same device

```
DeviceMatrixState * DMS = srcDevice->getInternalRoutingMatrixState();
DMS-> handlePatchRequest (srcIOIndex, destIOIndex, !createCon);
```

## A.7 Code sample demonstrating the selection and creation of a multicore connection between 2 devices in the same subnet

```
bool isMcSet = false;
bool wasMcCreated = false;
MulticoreSocket * srcMC;
MulticoreSocket * destMC;
for (int y = 0; y < srcDevice->GetNumberOfOutputMulticores();y++){
    srcMC = srcDevice->getMulticoreAtIndex(false,y);
    for (int x = 0; x < destDevice->GetNumberOfInputMulticores();x++){
        if (destDevice->getMulticoreAtIndex(true,x)->isPatchedToSource(srcMC)){
            destMC = destDevice->getMulticoreAtIndex(true,x);
            isMcSet = true;
            break;
        }
    }
    if (isMcSet)
        break;
}
if (!isMcSet){
    for (int y = 0; y < srcDevice->GetNumberOfOutputMulticores();y++){
        if (srcDevice->getMulticoreAtIndex(false, y)->IsRunning()){
            for (int x = 0; x < destDevice->GetNumberOfInputMulticores();x++){
                if (!destDevice->getMulticoreAtIndex(true, x)->IsRunning()){
                    srcMC = srcDevice->getMulticoreAtIndex(false, y);
                    destMC = destDevice->getMulticoreAtIndex(true,x);
                    isMcSet = true;
                    break;
                }
            }
        }
        if (isMcSet){
            if (!setPatchBayMulticore(srcDevice, destDevice, srcMC, destMC, createCon))
                return false;
            wasMcCreated = true;
            break;
        }
    }
}
```

## A.8 High level code sample handling internal connections of 2 devices if a multicore connection has been established

```
if (!setPatchbayInternalRouting(srcDevice, destDevice, srcMC, destMC, srcIOIndex, destIOIndex,createCon)){
    if (wasMcCreated)
        setPatchBayMulticore(srcDevice, destDevice, srcMC, destMC, !createCon);
    return false;
}
```

## A.9 Work-around code distinguishing multicores from audio pins

```
DeviceMatrixState * srcDMS = srcDevice->getInternalRoutingMatrixState();
DeviceMatrixState * destDMS = destDevice->getInternalRoutingMatrixState();

bool srcConCreated = false;
bool destConCreated = false;
for (int y = 0; y < srcDMS->getNumColumns(); y++){
    if (srcDMS->getColumnName(y).length() > 11)
        String columnLabel = srcDMS->getColumnName(y);
        if (columnLabel.substring(0,9).compareIgnoreCase(T("Multicore")) == 0) {
            StringArray columnTokens;
            columnTokens.addTokens(columnLabel.substring(9),false);
            if (columnTokens[0].compare(String(srcMC->GetID()))==0)
                String columnSequenceNumber = columnTokens[2];
                for (int x = 0; x < destDMS->getNumRows(); x++)
                    if (destDMS->getRowName(x).length() > 11)
                        String rowLabel = destDMS->getRowName(x);
                        if (rowLabel.substring(0,9).compareIgnoreCase(T("Multicore")) == 0)
                            StringArray rowTokens;
                            rowTokens.addTokens(rowLabel.substring(9),false);
                            if (columnTokens[0].compare(String(destMC->GetID()))==0) {
                                String rowSequenceNumber = columnTokens[2];
                                if (columnSequenceNumber.compareIgnoreCase(rowSequenceNumber) == 0) {
                                    (...) Handle Actual Routing
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
return false;
```

## A.10 Sample code that reverts network changes if an error occurs in the connection process

```
bool wasConJustChanged = false;
if (srcDMS->isCrosspointEnabled(srcIOIndex,y,pendingRequestCount))
    srcConCreated = true;
else{
    srcConCreated = srcDMS->handlePatchRequest(srcIOIndex, y, createCon);
    if (srcConCreated == true)
        wasConJustChanged = true;
}
if (srcConCreated && destDMS->isCrosspointEnabled(srcIOIndex,y, pendingRequestCount))
    destConCreated = true;
else if (srcConCreated)
    destConCreated = destDMS->handlePatchRequest(x, destIOIndex, createCon);
if (wasConJustChanged && !destConCreated){
    srcDMS->handlePatchRequest(srcIOIndex, y, !createCon);
    srcConCreated = false;
}
if(destConCreated && srcConCreated){
    return true;
}
```