Literature Review: An investigation into a mobile solution for controlling audio networks

Sascha Zeisberger

July, 2010

Contents

1 Introduction **2**

2 AES-X170 Protocol **3**
2.1 Connection Management . . . 3
2.2 Routing . . . 4
2.3 Parameters and Desk Items . . . 5
2.4 Protocol Structure . . . 6
2.5 Limitations . . . 10

3 Programming for the iPhone OS **11**
3.1 Specifications . . . 11
3.2 Requirements . . . 11

4 Juce **12**
4.1 Features . . . 12
4.2 Targeting Specific Platforms . . . 13
4.3 Juce Usage . . . 14
4.4 Limitations . . . 14

5 Unos Vision **14**
5.1 Features . . . 16
5.2 Usage . . . 16
5.3 Structure . . . 17

6 Conclusion **20**
Bibliography20

List of Figures

1 Two multicore connections between two devices [13] 4
2 Multicore router routing between two subnets. 5
3 Example of a fader and a meter desk item. 7
4 Structure of the AES-X170 protocol 8
5 Example of using a preprocessor directive to execute iPhone specific code 13
6 Preprocessor directive to include either iPhone or Windows specific code depending on the programming environment. 14
7 Audio network using a Unos Vision workstation[13] 15
8 Devices view of Unos Vision 17
9 Connection Manager view of Unos Vision 18
10 Use case diagram of Unos Vision 19
11 Class diagram of Unos Vision. 20
12 Sequence diagram summarising X170 communications at Unos Vision start-up. 21
13 Sequence diagram describing multicore patching. 22
14 Sequence diagram summarising internal routing of a device. . . . 23

List of Tables

1 Sample table displaying internal routing matrix. 4

1 Introduction

This document explores the Unos Vision application suite, focusing on the main components that enable it to control an audio network. Additionally, the iPhone programming environment and the Juce application framework is discussed so the environment in which a mobile application can be programmed in is familiarised and defined. This will ultimately result in the establishing of an environment that will allow the the Unos Vision application suite to be ported to a mobile platform.

This document is laid out in a hierarchical fashion, focusing on the low level attributes of Unos Vision first. First the AES-X170 protocol is discussed so as to get an understanding on the structure of an X170-based audio network. Following is an explanation of the Juce application framework. This will provide insight as to how the core Unos Vision functions work. The iPhone programming environment is described, focusing on the requirements for compiling to an iPhone device. Finally, the Unos Vision application is analysed, focusing on UML diagramming techniques to describe the programs functionality.

2 AES-X170 Protocol

The AES-X170 protocol, allows for the remote administration of devices in an audio network [11]. 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 [11].

X170 messages will generally fall under three main types [7]:

- 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 Connection Management

The X170 protocol allows for the routing of audio signals between devices through the use of a multicore [13]. A multicore is a network connection between two devices whose concept is similar to the physical cable connections in an analogue audio network [7]. 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 [7]. 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.

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 [6]. 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 [6]. 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:

1. Create a multicore connection between two devices.

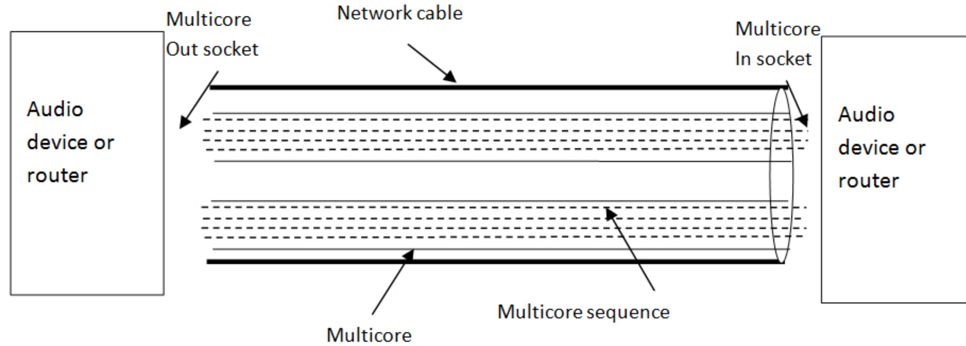

Figure 1: Two multicore connections between two devices [13]

	Analogue Out 1	Multicore1 Out 1	…
Analogue In 1	X	X	
Multicore1 In 1			
…			

Table 1: Sample table displaying internal routing matrix.

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.

2.2 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 [5]. 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.

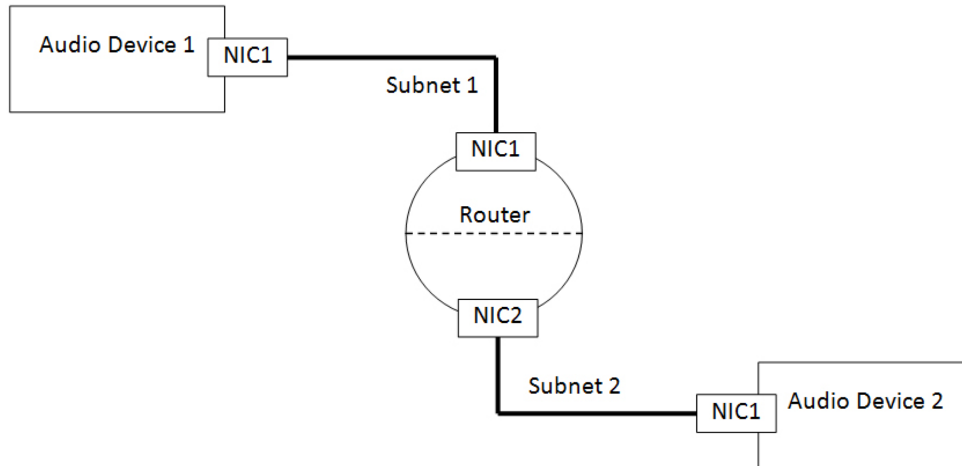

Figure 2: Multicore router routing between two subnets.

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.
- 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.

2.3 Parameters and Desk Items

There are three methods of access for any parameter that adheres to the X170 stack[7]:

- 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[7]. 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[7].

Parameters can be grouped to establish either a master-slave or a peer-to-peer relationship with other parameters[6]. 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[6]. In a peer-to-peer group, a change in any single parameter will affect all the parameters in the joined group[6].

Desk Items are control items that can be used to provide a virtual representation of a device[7]. 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[12]. 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[7].

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.

2.4 Protocol Structure

The Open Systems Interconnection model, known as the OSI model, is a communications system that divides network communications into seven manageable sections[9]. The layers in order from highest to lowest level are [9]:

- 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.
- Transport layer - responsible for controlling end-to-end connectivity between hosts in segments, for example the TCP and UDP protocols.

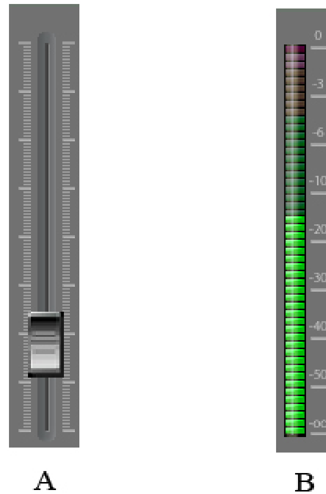

Figure 3: Example of a fader and a meter desk item.

- 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 [7]:

- 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 [7].

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 [7].

<table>
<tr><td colspan="3">IP header</td><td colspan="4">Includes all IP related fields</td></tr>
<tr><td rowspan="12">IP data field</td><td colspan="2">UDP header</td><td colspan="4">Includes all UDP related fields</td></tr>
<tr><td rowspan="11">UDP data field</td><td rowspan="7">X170 header</td><td colspan="4">Target IP Address
(32-Bit)</td></tr>
<tr><td colspan="4">Target X170 Node ID
(32-Bit)</td></tr>
<tr><td colspan="4">Sender IP Address
(32-Bit)</td></tr>
<tr><td colspan="4">Sender X170 Node ID
(32-Bit)</td></tr>
<tr><td colspan="4">Sender Parameter ID
(32-Bit)</td></tr>
<tr><td>User Level
(8-Bit)</td><td>Message Type
(8-Bit)</td><td colspan="2">Sequence ID
(32-Bit)</td></tr>
<tr><td colspan="2">Sequence ID
(continued)</td><td>Command Executive
(8-Bit)</td><td>Command Qualifier
(8-Bit)</td></tr>
<tr><td rowspan="4">X170 address block</td><td>Section Block
(8-Bit)</td><td>Section Type
(8-Bit)</td><td colspan="2">Section Number
(24-Bit)</td></tr>
<tr><td>Section Number
(continued)</td><td>Parameter Block
(8-Bit)</td><td colspan="2">Parameter Block Index
(24-Bit)</td></tr>
<tr><td>Parameter Block Index
(continued)</td><td colspan="2">Parameter Type
(24-Bit)</td><td>Parameter Index
(16-Bit)</td></tr>
<tr><td>Parameter Index
(continued)</td><td>Value Format
(8-Bit)</td><td colspan="2"></td></tr>
</table>

Figure 4: Structure of the AES-X170 protocol

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[7].

The fields in the X170 header have the following functions [7]:

The *Target* and *Sender IP address* fields contain the IP addresses of the source and destination devices[7]. 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[7]. 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[7]. 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[7].

The *Message Type* field specifies the role of the message and can identify whether a message is a request or a response[7].

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[7].

The *Command Executive* field defines the nature of a message, such as whether it is either getting a parameter value or setting it[7]. The *Command Qualifier* field can be used to specify an attribute of a parameter or it can refer to a group of parameters[7].

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 [6].

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[7].

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[7]. 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[7]. 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[7] 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[6].

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[6].

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[7]. 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[7]. 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.

2.5 Limitations

The specification of the X170 protocol only accommodates routing devices that are in the same subnet[7]. 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[6]. 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[7]. 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.

3 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.

3.1 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)

3.2 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.

4 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[14]. 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 [15].

4.1 Features

The current iteration of Juce offers a large variety of features. Some of these features include[14]:

- 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[14].

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[15].

4.2 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[4]. 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 [10]. 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 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:

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 appli-

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

Figure 6: Preprocessor directive to include either iPhone or Windows specific code depending on the programming environment.

cation 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.

4.3 Juce Usage

There are two supported methods for including Juce into a project[14]:

- 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[14].

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[14].

4.4 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[16].

5 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[13]. Interaction between the user and the devices on an audio network is achieved by providing a graphical interface that[13]:

- 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[13].

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

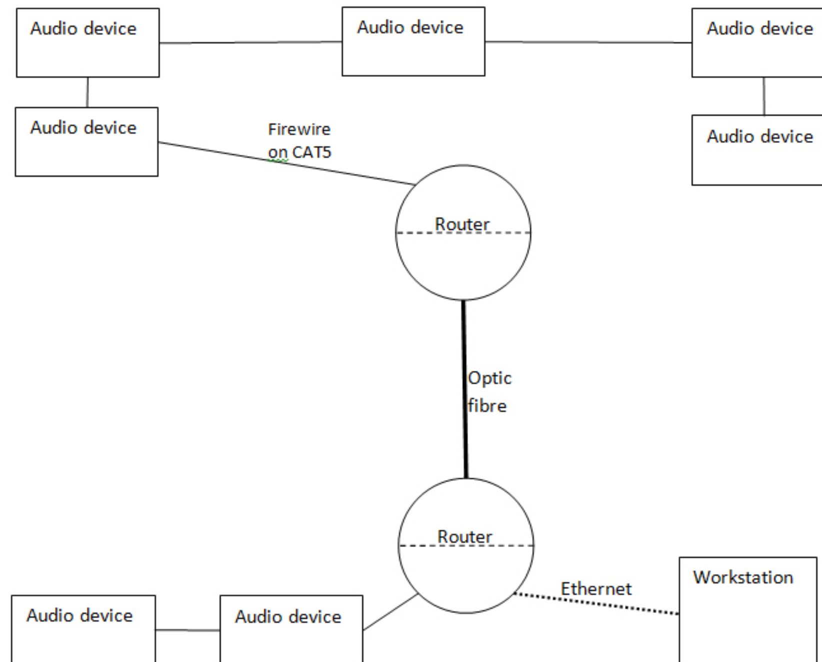

Figure 7: Audio network using a Unos Vision workstation[13]

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.

5.1 Features

The current version of Unos Vision offers the following main features[13]:

- 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[12].

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.

5.2 Usage

The Unos Vision application is divided into two main tabs[13]:

- 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 2.7 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[13].
- Container 2 displays the possible multicore sockets for each device, where selecting a cross-point will create a multicore connection between the devices[13].

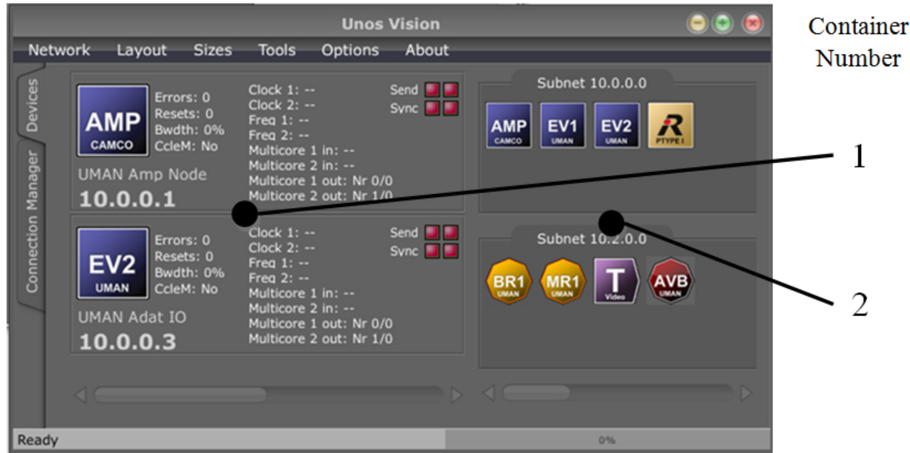

Figure 8: Devices view of Unos Vision

- 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[13].
- Containers 3 and 6 display a device's graphical desk item component, allowing a user to interact with the parameters of a device[13].

5.3 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[8]. 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[8].

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 ac-

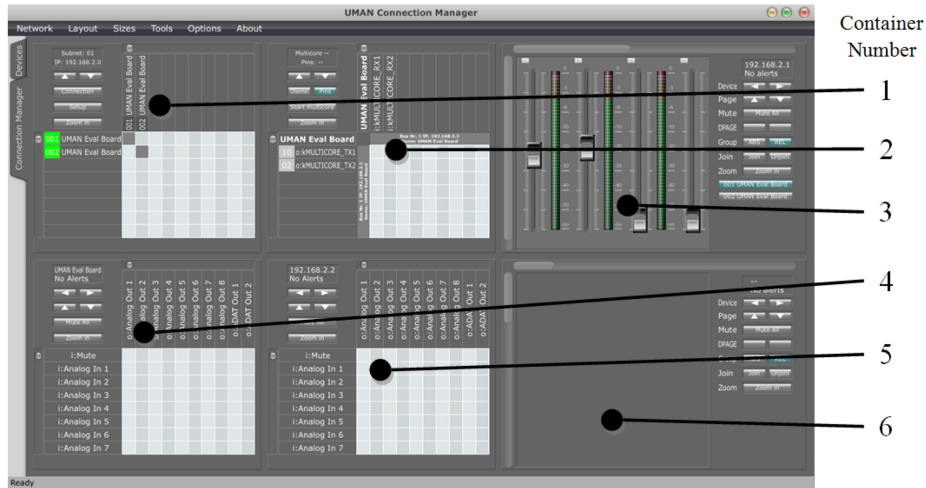

Figure 9: Connection Manager view of Unos Vision

tual 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 2.9 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 2.10, 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.

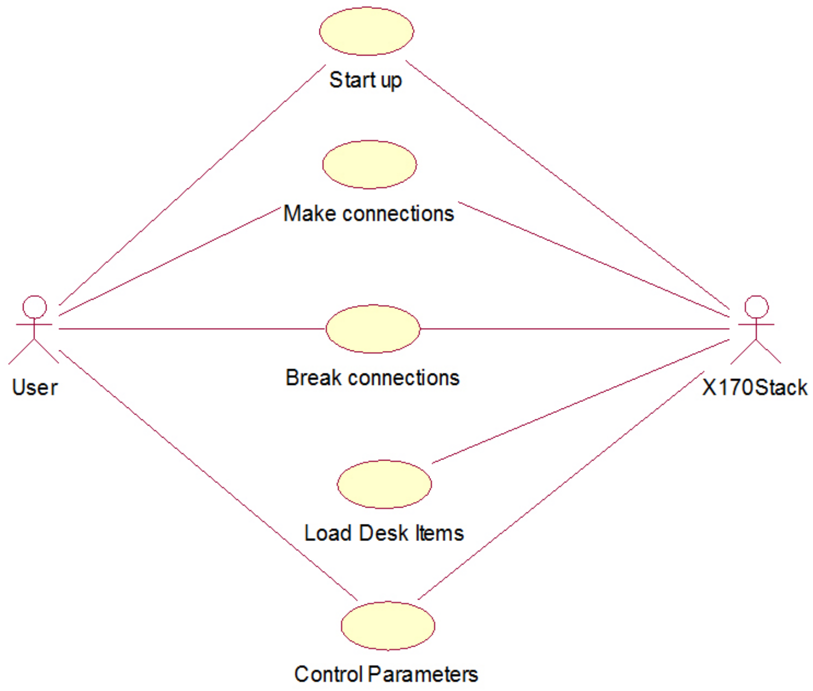

Figure 10: Use case diagram of Unos Vision

Figure 2.12 demonstrates the establishing of an internal connection.

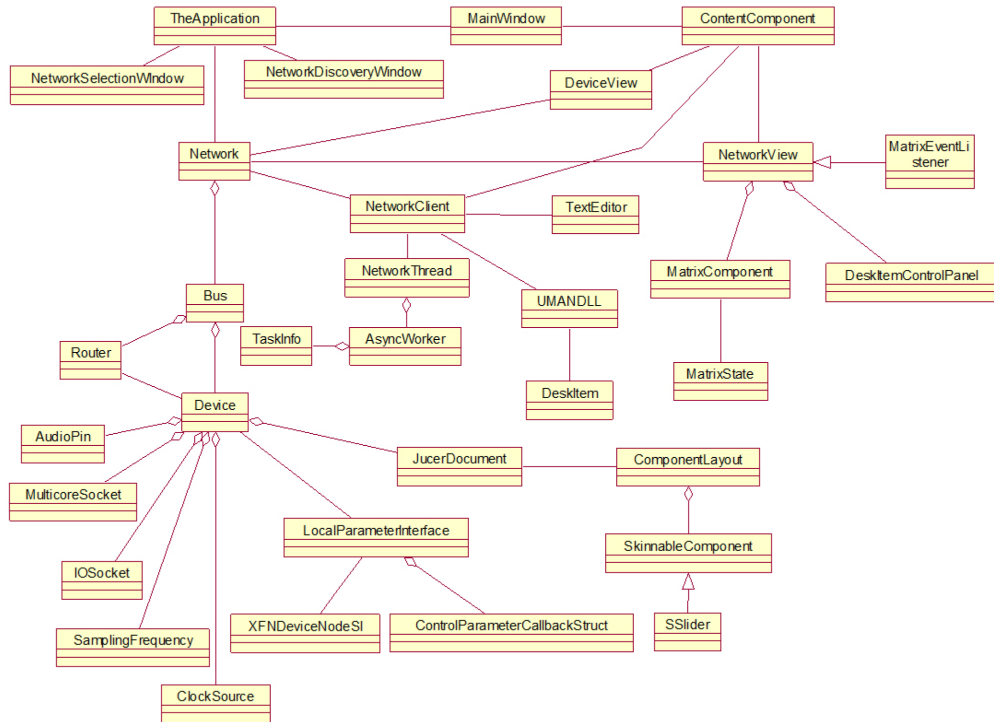

Figure 11: Class diagram of Unos Vision.

6 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 application 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.

References

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

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

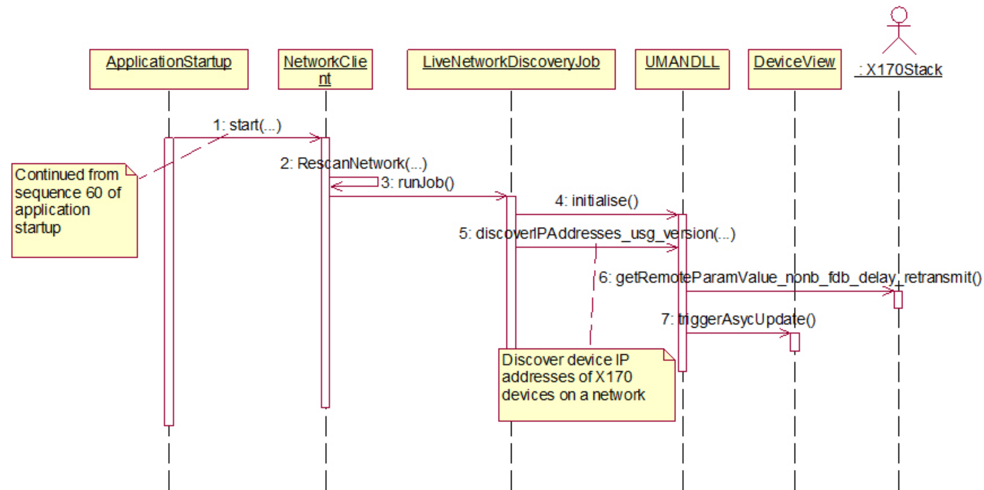

Figure 12: Sequence diagram summarising X170 communications at Unos Vision start-up.

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

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

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

[6] 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.

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

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

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

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

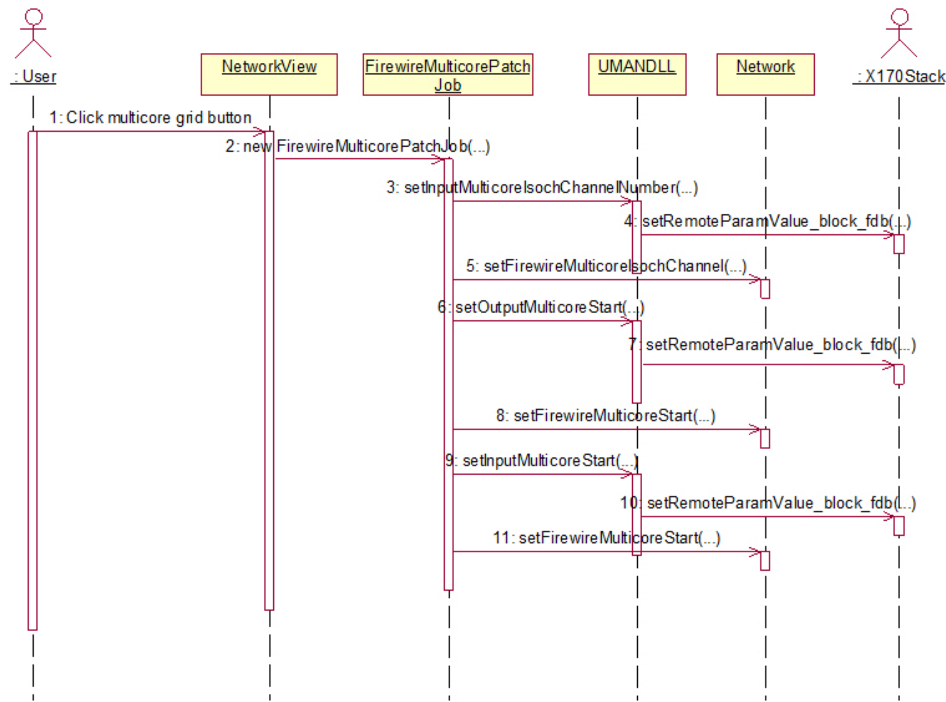

Figure 13: Sequence diagram describing multicore patching.

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

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

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

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

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

[16] 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].

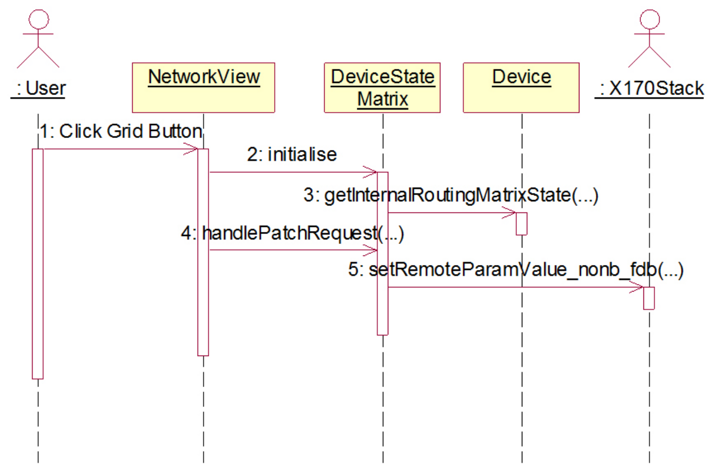

Figure 14: Sequence diagram summarising internal routing of a device.