The Remote Wireless Multipoint Control of Digital Mixing Consoles.

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Brent Shaw

Grahamstown, South Africa
1 November 2013

Abstract

This thesis aims to discuss various aspects of audio control systems. Through the course of this thesis, discrete field topics of study are brought together to contribute to the understanding of what is required to develop a comprehensive audio control system. An investigation will also be conducted into the viability of a distributed wireless control system for digital mixing consoles.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the NRF and Rhodes University for the financial support that allowed me to complete this research. Finally I would like to acknowledge the financial and technical support of Telkom, Tellabs, Stortech, Genband, Easttel, Bright Ideas 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Chapter 1

Introduction

Technological advancements in the digital era have resulted in a widespread move toward the digital processing of information. This move is reflected in the audio industry, resulting in professional audio equipment becoming more compact, functional and affordable. This movement towards digital hardware has had a major impact on the music industry with the creation of the digital mixing console. These consoles feature a range of different functions such as digital gain control (for microphone preamplifiers), level control, flexible internal audio routing and powerful Digital Signal Processing (DSP) that can be used for effects (such as reverb or delay) or for sound enhancement (audio equalisation or feedback control). These mixing consoles are used in a variety of environments, from live music venues (such as bars or concert halls) to restaurants, recording studios, conference venues and even hotels, making the digital mixing console indispensable to the professional, and amateur, production of audio.

Although smaller than their analogue counterparts, these consoles are still fairly large and often weigh in excess of 20kg, usually having dozens of audio input and output lines connected to the console. For this reason these desks tend to be installed out of the way, in secure locations. These locations may be in back rooms or control rooms, where direct sight or earshot of the stage or podium is lost resulting in the rise of accessibility issues. Furthermore, the mixing console may be placed in a sub-optimal position for listening which leads to acoustic difficulties, forcing the engineer to leave the desk in order to gain better perspective on the acoustic performance.

Typically, audio engineers are required to be in close proximity to the mixing console in order to adjust parameters such as gain or level as required. If a venue has only a single engineer, then the engineer may be required to walk many times between the desk

and the stage or podium in order to change settings or correct any issues that may arise. This also has the implication that to provide adequate control, each desk would require an engineer, although each engineer may not constantly be occupied by making alterations or adjustments, but merely present in order to ensure total control. This is ultimately time consuming for an engineer and has a low efficiency as engineers could better allocate their time to multiple tasks. Maintaining efficient control of the console(s) while remaining mobile is therefore a problem that occurs in many audio engineering environments.

Many digital mixing consoles currently available have a MIDI (Musical Instrument Digital Interface) port on board, through which MIDI messages can be passed to provide control for many of the parameters on the desk itself. This allows for the mixing console to be controlled through the use of separate MIDI control surfaces or through software on a computer equipped with a MIDI interface. Unfortunately the MIDI specification, although fairly comprehensive, does have its own drawbacks; one of which is a maximum cable length of 15 meters.

This makes remote control possible but still rather limited as the controller is then required to be in close proximity to the desk. Some manufacturers of digital mixing consoles (such as Allen & Heath) have built in their own control systems (not based on MIDI) which in some cases can even provide wireless control from tablet devices such as the Apple iPad (the standard in the audio industry). These systems provide comprehensive control, but are limited to particular devices, often only working within their brand, and in some cases only available for specific consoles.

1.1 Statement of Problem

There is no universal or generic system[1] for controlling these desks remotely. From the above discussion, it appears clear that there is a need for a system that can provide remote control for digital mixing consoles, while not limiting its control to individual brands or desks. For such a system to be truly useful, a mobile interface for control would be of benefit to the industry and would increase the usability and efficiency of the system. Finally the ability of the system to provide remote control for multiple digital mixing consoles from multiple control devices would be of great use in larger venues and

[1]Generic control system: A system capable of providing control for digital mixing consoles, regardless of make or model. The system must be capable of accounting for differing parameters and functions held by various digital consoles. At a high level, the system takes user input (through a user interface) and provides the console with parameter control and change messages (via a standard control protocol)

multiroom setups as it would address the problem of inefficiency experienced by sound engineers, as discussed earlier.

1.2 Goal of Research

Digital mixing consoles are becoming increasingly popular and this study will investigate current trends in mixing consoles, as well as examining how these consoles function and how their various features can be accessed and controlled.

The thesis provides a broad review of two audio device control protocols. The Musical Instrument Device Interface (MIDI) has been a popular control protocol for a range of audio devices in past years and its use as a control protocol needs to be examined. Another control protocol that is becoming increasingly popular is Open Sound Control (OSC). These protocols will be reviewed and compared to gain insight into currently available mixer control systems.

As one of the primary goals of this thesis is to provide a new implementation of the MIDINet system, a full analysis and review of the existing MIDINet system will be provided. The purpose behind the system's original inception will be investigated and a complete account of how MIDINet functions will be given.

The project aimed to produce a design and implementation for a distributed control system capable of controlling multiple digital mixing consoles. The system is based on the MIDINet system described by Mosala's MSc thesis (Mosala, 1995). The system is currently only capable of running on standard hardware machines, loaded with Microsoft Windows XP, 7 or 8 platforms.

For the control system to be mobile it would best be deployed on wireless tablet computers. Apple's iPad is the industry standard for audio control applications and so a MIDINet implementation capable of running on this device would be of great use to the industry, thereby achieving the goal of this research.

This thesis examines the various methods by which audio mixing desks can currently be controlled. It investigates currently available systems that allow mixing consoles to be controlled remotely (both through desktop applications as well as via wireless tablet computers). It also investigates existing control protocols and how they are used in digital mixer control.

This thesis assumes a basic knowledge of mixing consoles. Any terms or concepts outside of the most basic are described and detailed through the course of the paper.

Chapter 2

Digital Mixing Consoles

A mixing console is a piece of hardware used in the audio engineering industry for summing, processing and routing audio. Analogue desks vary in design, size and features, from beginner desks that feature only one or two channels, to small public address (PA) desks, to large split console desks found in recording studios. (Izhaki, 2013)

Digital mixing console are in many ways similar to analogue mixing consoles, as they perform the same functions and can be of similar size, but the difference while possibly not immediately evident, is substantial. Digital consoles convert audio from its original analogue signal to a digital signal as the audio enters the desk. From this point on, the sound remains in digital form while within the confines of the mixing console. (Izhaki, 2013, Ballou, 2008)

All the functions of an analogue mixing console such as summing, processing and routing are available on digital console as well. The only difference is that the audio is being digitally modified in order to mimic the analogue console. The debate regarding the quality differences between digital and analogue mediums is an argument that cannot be easily settled, if settled at all. (Izhaki, 2013)

But audio quality aside, digital mixing consoles feature remarkable flexibility. These consoles possess a number of pots and faders which are thought to control various aspects of the mixer. These faders and pots can be linked to parameters, thereby allowing them to control specific, chosen parameters. In comparison, the faders and pots found on an analogue mixing console are not directly connected to modules and thus do not directly control anything. The capability of digital mixing consoles to control anything is due to the increasing amount of processing power that can be built directly into digital mixers.

(Ballou, 2008)

2.1 Advantages of digital mixing

Digital mixers have a distinct advantage over their analogue counterparts when it comes to flexibility and features. In an analogue mixing console, signals could only be routed to a finite number of destinations. With digital mixing however, inputs can be internally routed to anywhere within the console. The same goes for processes in digital consoles such as equalisation, compression or gating. There is almost no limit to the combination of processes and routings that can be achieved when using digital mixing console.

Another advancement that digital mixing consoles have made popular is that of motorized faders. This allows for levels to be saved so that they can be restored at any point. This feature also allows for digital mixing consoles to feature more channels than physical inputs on the desk itself. The consoles can have virtual channels such as group, effect and reference channels that can be controlled using the same faders and pots used to control the channels fed by the microphone or line inputs.

The desks can also have digital inputs and outputs that can be used to add extra channels (signals in or out) to the desk. Alesis Digital Audio Tape (ADAT) is an optical audio specification that allows for 8 channels of digital audio to be transmitted over a single optical core. This allows engineers to add extra inputs and microphone preamplifiers to compact digital consoles.

2.2 Yamaha 01v96

For the purposes of this research the Yamaha 01v96 digital mixing console, seen in Figure 2.1, has been selected for investigation into mixing console control system. The decision to use this console is based on the availability of the Yamaha 01v96 to the project. This will allow for an investigation into the methods by which this console can be controlled. (Yamaha, n.d.b)

Figure 2.1: The Yamaha 01v96.

2.2.1 Features

The Yamaha 01v96 features both analogue and digital audio inputs. The console has a built in MIDI interface which can be used for saving settings as well as sending control messages.

A feature that the Yamaha range of digital consoles possesses which aids in expandability and future proofing is an expansion card slot. Yamaha has developed a number of expansion cards that can be used for adding additional features to their consoles. (Yamaha, n.d.b)

2.2.2 Control

Digital mixing consoles differ from analogue mixers in the way by which they are controlled. None of the controls on the console are directly attached to any modules such as levels, panners or equalisers, but are rather linked to parameters. The parameters are then used internally to provide settings to the console's DSP unit. (Yamaha, n.d.b)

Since these parameters are all digital and can be adjusted through the console interface, fewer physical controls need be placed on the console. A single rotary encoder on the console may be used to control a number of different parameters. The console's display can provide feedback as to the current state of parameters held by the console. (Yamaha, n.d.b)

Owing to the ability of the digital console to provide control for differing parameters using limit control, the mixing console provides a single set of rotary encodes that can be used to control the equalisers, gates, compressors and other functions to every channel of the desk. A set of controls for each channel is not necessary therefore allowing the console to remain considerably more compact than an analogue mixer with the same capabilities. (Yamaha, n.d.b)

The Yamaha 01v96 provides 16 linear motorised faders. The console itself has a total of 32 audio input channels and number of remote and master channels that require level control as well. As only 16 faders are provided, Yamaha has implemented a system that allows the other channel to be controlled through selecting of different layers. (Yamaha, n.d.b)

2.2.3 Remote Control

With such a wide range of parameters and features, the Yamaha 01v96 is a powerful console. Yamaha has equipped the console with a MIDI interface which allows the desk to send and receive MIDI messages. This allows the desk to be used to control external consoles and devices via MIDI as well as allowing other devices equipped with MIDI to control the mixer remotely.

Yamaha has also provided a set of MIDI messages that can be used to control the internal parameters. The messages can also be used to return parameter updates to external devices when parameters on the mixer are modified through the mixer's physical controls.

Yamaha has developed a desktop control application that can be used to provide remote control of the console through the use of a computer MIDI interface. This and other control applications are discussed further in Chapter 3.

Chapter 3

Remote control applications for mixing consoles

There are many remote control applications available which provide varying amounts of control, ranging from proprietary systems to open source projects. However, despite the range of applications available, there are tends to be a number of similarities between systems such as their interfaces, how they send control messages and how they interface physically with the hardware.

No control applications manage to provide a level of usability and control that sets them far enough apart from other systems to become an industry standard. This chapter will look at a few control applications that are currently available and how they compare to each other.

The applications mentioned below are only a few of the control applications available and their styles of control represent only a few of the schools of thought in terms of digital mixer control.

The comparison should aid in the formulation of a clearer understanding of what a control application for digital mixing consoles should entail, and whether the current range of control applications is moving toward any particular standards.

3.1 Matrix Mixer

A novel concept for mixing console control and routing is embodied in the Matrix Mixer developed by Philip Foulkes. (Foulkes, 2006) The thought behind the concept was that routing by setting sources and destinations on a mixing console could be simplified by viewing the possible connections as a grid.

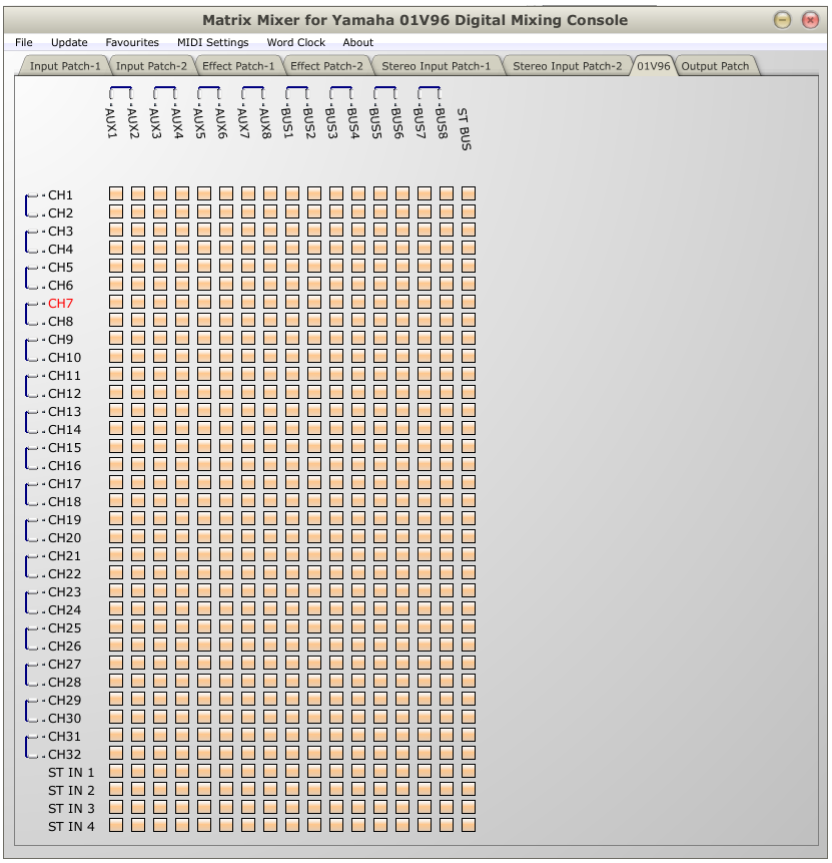

Figure 3.1: Example of matrix based connection management.

Figure 3.1 shows the user interface for Matrix Mixer, which provides a simple overview of connections, linking inputs (listed along the left hand side) to outputs (along the top on the matrix). By selecting an intersection point between an input and an output, a connection can be made. For control of parameters associated with a single input or output, the channel can be selected and a channel parameter page is displayed. Parameters such as level, equalisation settings and panning can be altered from this view. (Foulkes, 2006)

Philip Foulkes's Matrix Mixer application was originally designed to work with the Yamaha 01X mixing console, but its generic design allows it to be used on a variety of mixing

consoles. Mixing consoles are represented using Extensible Markup Language (XML), allowing for the application to take on the form and parameters of any desk. (Foulkes, 2006) This makes the representation of mixing consoles, both old and new, rather simple, making it possible to integrate this application with a wider range of products.

Connection using a proprietary format would lead to the application being less generic, and so the MIDI standard was implemented for the control messaging. The desk and application are synchronised (so that both are in the same state and all parameters match). MIDI also allows for control messages (from parameter changes on screen) to be both sent to the desk from the desktop application and sent from the desk (as parameters on the desk are changed) to the application. These control messages are transmitted via a MIDI interface connected to the computer. No touch implementation is currently available. (Foulkes, 2006)

3.2 TouchOSC

TouchOSC, shown in Figure 3.2, is a multitouch enabled application available for Android and iOS devices, developed by Hexler(Hexler, n.d.). The application has a number of different control layouts that come standard with the installation.

The application provides users with faders, buttons, pads and rotary pots, each set to send a predefined OSC message. The developer has also provided a designer application that allows users and developers to create their own control surfaces for which they can define their own messages. Owing to OSC's open and undefined messaging format, it is important that controls are set up to send OSC messages which the server or application on the receiving side can interpret and act upon.

Figure 3.2: Example layout of a TouchOSC control window. Image from www.hexler.net .

Jabarudian Industries, creators of the Missing Link[1], have provided a control layout that is compatible with the Yamaha 01v96. When used in conjunction with the Missing Link, the OSC messages are capable of giving TouchOSC basic control over the mixing console's commonly used functions.

[1]The Missing Link is a hardware device that features a MIDI interface (both imput and output port) and a wireless module. It is capable of receiving OSC over a wireless and producing MIDI messages, and vice versa.

3.3 Yamaha

Yamaha Pro Audio Inc. developed their own control application, Studio Manager, which is currently available in its second version offering support for a variety of Yamaha products. Figure 3.3 shows the Yamaha Studio Manager software, which allows users to exert control over compatible Yamaha products connected to the controlling computer. Devices such as the Yamaha 02R96, 01V96 and 01X (Yamaha, n.d.c) are all compatible with software, allowing a user to adjust parameters, route signals and change levels on these devices remotely. (Yamaha, n.d.a)

Figure 3.3: Yamaha Studio Manager desktop application for the 01x. Image from www.yamahaproaudio.com .

In the case of the 01V96 and 01X, Yamaha has implemented four layers, allowing these desks to support more mixing and channels than they have physical controls for. This is an issue as engineers would need to switch layers in order to adjust the level on a channel that is not part of the current layer. With the Yamaha control software, viewing more than one layer at a time is not a problem, and thus the software can provide further control over the device. (Yamaha, n.d.a)

Yamaha have also released an application called StageMix that can provide control for their CL, M7CL and LS9 mixing consoles. The application is designed for the Apple iPad and aims to give the user remote control of the above mentioned mixing consoles. The application requires a wireless connection to an access point connected to the mixing

console, but allows the user to move freely within the confines of the wireless network and control the mixing console via the tablet.(Yamaha, n.d.a)

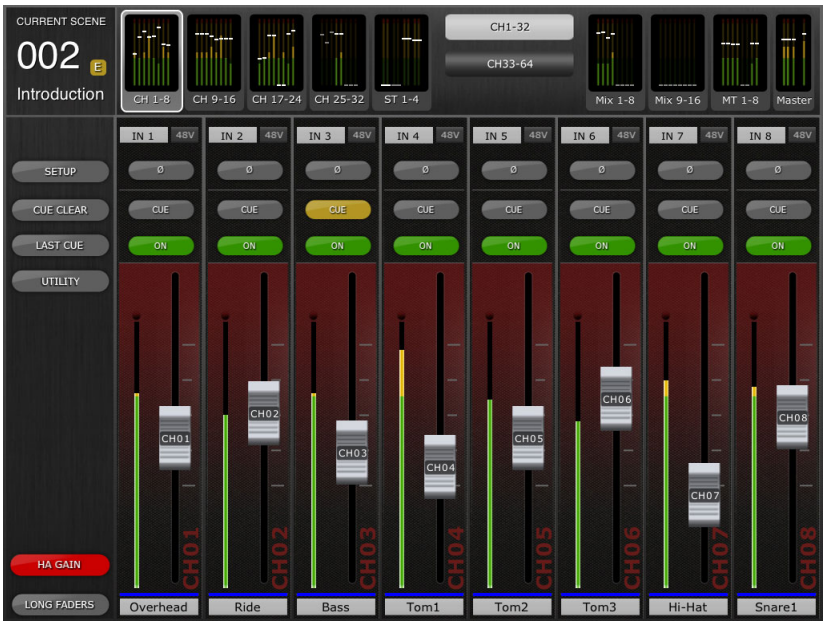

Figure 3.4: Yamaha Stage Mix iPad application for the LS9 digital mixing console. Image from www.yamaha.com .

Figure 3.4 shows the interface for the StageMix application for iPad. The touch screen interface provides the engineer with a more natural and familiar form of control as the interface provides control that resemble those of most mixing consoles.

3.4 Allen & Heath

Allen & Heath Ltd., like Yamaha, provide applications for the control of their own brand mixing consoles. Allen & Heath provide both desktop and tablet based applications for their iLive range of digital mixing consoles (Allen&Heath, n.d.b). iLive Editor is a control application aimed at desktop and laptop computers and is available both for PC and Apple Mac (Allen&Heath, n.d.c). The computer application communicates with the iLive mixer using a standard Ethernet cable. Wireless control can be achieved by connecting the iLive mixer to a wireless access point. Allen & Heath have a version of the application available for the Apple iPad called iLive MixPad, which uses WiFi to connect to mixing consoles which are attached to a WiFi access point. The iPad application provides touch control for parameters, giving the user a more intuitive and "closer to the real thing" experience. These applications provide a wide range of control, but are limited to controlling only Allen & Heath iLive Series mixing consoles. (Allen&Heath, n.d.d)

A similar application to the iLiveMix application for Apple iPad, GLD Remote, is available for Allen & Heath's GLD Series of mixing consoles. GLD Remote is aimed at providing total wireless control and allows for a single iPad to control many GLD mixers. It also allows for multiple iPads to control a single desk, separating control for various sections of the desk to different iPads. (Allen&Heath, n.d.a)

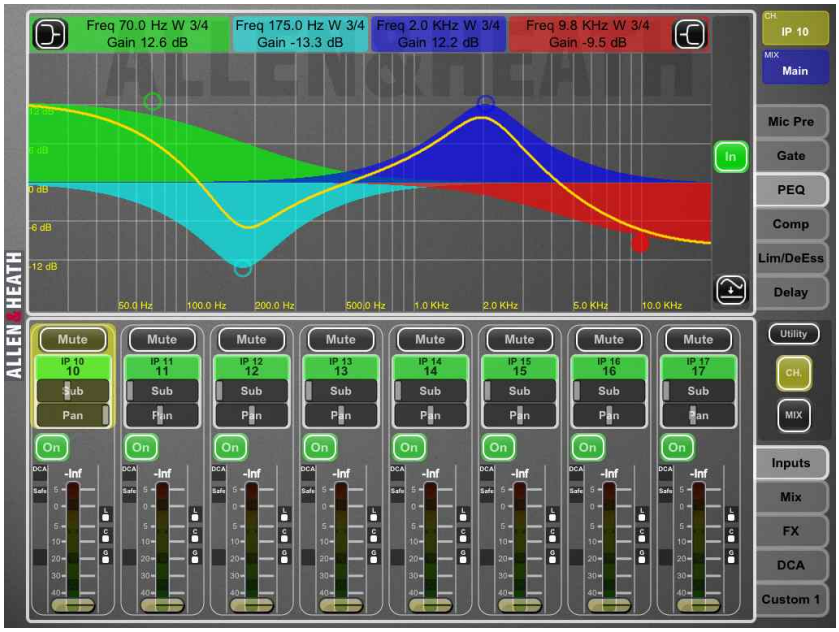

Figure 3.5: Allen & Heath's iLive iPad mixing application. Image from www.allen-heath.com .

3.5 Conclusion

It is clear that there is no industry standard digital mixer control surface. While desktop control applications can provide extensive control over a digital mixer, the desktop computer is simply not a practical solution if mobility is to be considered in any way. Desktop computer applications also often tend to be less intuitive than touch screen based systems due to the use of alternative input devices such as a keyboard and mouse.

Touch screens provide a sense of familiarity as engineers use their fingers to control faders and gain pots on touch screen systems in much the same way they use professional mixing consoles. This familiar feel greatly contributes to the ease and efficiency of using touch screen devices as control systems.

How mixing consoles communicate with control application seems to be another area where the industry cannot agree. Various Ethernet implementations, MIDI and OSC have been used as communications media between the mixing console and the control application.

Mobile touch screen devices such as tablets would form a good platform for a control application to be built around, as they provide high resolution multi touch displays, power efficient, high performance processors and wireless networking. This would enable the tablet to host a comprehensive control application capable of sending control messages over WiFi.

Chapter 4

Musical Device Control Protocols

4.1 Musical Instrument Digital Interface

4.1.1 History

The advent of synthesizers revolutionized the music industry in the 1960's and 1970's with musicians being capable of producing a wide range of new sound. But an early issue that arose with synthesizers was that many could play only one note at a time. This had obvious limitations and manufacturers attempted to rectify this by implementing control voltage systems that allowed for a keyboard to control a synthesizer. This allowed for more than one note to be played at a time, but resulted in synthesizers becoming covered in control cables and setups becoming increasingly complicated. (Rothstein, 1995)

Dave Smith and Chet Wood, engineers working with Sequential Circuits, began work on a new protocol that would allow for the interoperability of synthesizers across different manufacturers. In 1981 they proposed their USI (Universal Synthesizer Interface) protocol to the Audio Engineering Society (Huber, 2012). Manufacturers and members of the audio engineering community began working on a standard for synthesizer control, and in 1983 the MIDI 1.0 Specification was released (Hosken, 2010).

The Musical Instrument Digital Interface (MIDI) standard was widely accepted with many new technologies at the time being released with MIDI compatibility. Today MIDI is still widely used and has been incorporated into synthesizers, discrete keyboards, effects units, guitars, mixing consoles, lighting consoles and is a key component in computer music production (Huber, 2012, Hosken, 2010, Jacobs & Georghiades, 1991).

4.1.2 Hardware

Most MIDI devices offer a selection of three MIDI ports, input, output and through. The through (often written "thru" as showing in Figure 4.1) is optional and allows for the daisy-chaining[1] of MIDI devices. Typically, The MIDI standard uses 5 pin DIN connections, but not all 5 pins are operational: only pins 2, 4 and 5 are used. (Jacobs & Georghiades, 1991)

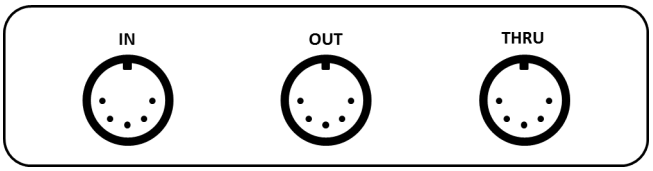

Figure 4.1: Example setup of MIDI connections on a MIDI device.

The MIDI specification requires that the interface operates at 31.25 kbaud in an asynchronous manner. The interface's receiver is required to be opto-isolated and an output is not allowed to feed more than one input. (Huber, 2012) Single in... Why? Resistance and signal concideration s

4.1.3 Messaging Protocol

MIDI communicates using a series of bytes, either being status bytes or data bytes. A message is constituted by an initial status bytes followed by one or more optional data bytes. There are two types of messages, namely channel messages and system messages. (DeFuria & Scacciaferro, 1990)

Channel Messages

MIDI devices operate on a selected MIDI channel, and so send control messages pertaining specifically to that channel. These messages are referred to as channel messages, which can further categorised as channel voice messages and channel mode messages. (DeFuria & Scacciaferro, 1990)

The former, voice messages, can be used to transmit note messages, program messages and expressive messages. Note messages are generated when a key on a keyboard controller

[1]Where each device provides a connection to be made to the next device in a sequence. Information received by a device that is repeated out to the next device in the 'chain'.

is pressed; this is referred to as a "note-on" messages. When the depressed key is lifted, a "note-off" message is sent. (Kirk & Hunt, 1999)

Table 4.1 shows how the voice messages (including note-on and note-off) are structured. The necessity for a both "on" and "off" messages is owing to the fact that if the system was required to wait for the user to complete playing a note before transmission, considerable delay would be incurred. (Rothstein, 1995) When a key is pressed, the "note-on" message is sent with information such as which key was pressed (key number) and how hard (velocity). (Hosken, 2010)

Status Byte	Hex	Decimal	Data Byte 1	Data Byte 2
Note off	80 - 8F	128 - 143	Note number	Release velocity
Note on	90 - 9F	144 - 159	Note number	Attack velocity
Poly key pressure	A0 - AF	160 - 175	Note number	Pressure value
Control change	B0 - BF	176 - 191	Controller ID	Controller value
Program change	C0 - CF	192 - 207	Pressure value	-
Channel pressure	D0 - DF	208 - 223	Pressure value	-
Pitch wheel change	E0 - EF	224 - 239	Pitch bend (LSB)	Pitch bend (MSB)

Table 4.1: Table showing various types of channel voice messages. Adapted from "MIDI Programmer's Handbook" (DeFuria & Scacciaferro, 1990).

System Messages

In contrast to channel messages (which are channel specific), MIDI systems messages are broadcast to all MIDI devices. One feature of system messages is the ability to send device-specific messages known as "Exclusive Messages". System messages can also be used to communicate "Time" messages allowing for devices to be synchronised, as well as "Common" messages used for tuning and setup. (Kirk & Hunt, 1999)

4.1.4 Controlling with MIDI

MIDI, although not initially designed for such use, can be implemented for use in controlling other devices in the recording or production studio. Devices such as effects units and mixing desks can be controlled using MIDI keyboard controllers or other control surfaces. One distinct advantage gained through using MIDI controlled mixing consoles is that of automation. Automation allows levels to fade over a period of time without the

engineer having to slowly pull down on a fader. The ability to program the change of parameter so that the user need not constantly change it, be it a level, panning or an equalisation parameter, can not only make mixing a multitrack recording a lot simpler, but can also allow for engineers to attempt newer and more intricate mixing techniques. (Huber, 2012)

This can all be achieved by using a MIDI sequencer to store MIDI messages and have them played back during mixdown. The incorporation of MIDI into computers and applications has resulted in the creation of a simple method of providing control for mixing consoles and effects processors. Users can change parameters on an on screen application and have the changes reflected on the device being controlled. These changes can be recorded, modified and re-used by employing software based sequencers. (Huber, 2012)

4.1.5 Limitations and the future of MIDI

Since the MIDI specification has never been revised since its release, MIDI has a few limitations to its operation and its capabilities. MIDI cables are limited (in the standard) to being up to 15 meters long, to avoid the risk that attenuation might result in signal loss. (Jacobs & Georghiades, 1991) This, although seeming fairly inconsequential, remains an issue as large synthesisers, despite being capable of being distanced from their keyboard controller, are restricted by this maximum length of cable. Although there are systems available that can extend this distance, and some systems that even make it possible to send MIDI point to point wirelessly, this is still a restriction of MIDI.

Another issue with MIDI is its speed. MIDI operates serially, transferring bits one after another at what is considered a slow speed: 31.25 kbaud. This can lead to the problem of "MIDI lag". Where many complex MIDI messages and many channels are being used, MIDI messages can begin to lag owing to the slow speed of transmission. This speed limit makes MIDI less reliable in high demand areas such as recording and production studios. The same lag can be experienced when many MIDI devices are daisy-chained together. (DeFuria & Scacciaferro, 1990)

Furthermore, MIDI is limited to only 16 channels. For a single device, in general, 16 channels is a fair limit, with most synthesisers only being capable of generating a limited number of voices. Unfortunately, with most setups sporting multiple MIDI devices, 16 channels is too restricting.(Kirk & Hunt, 1999)

may only be due to the protocol's age. MIDI would definitely benefit from an increase

in speed and with modern architecture this would not pose much of an issue. The issue with attempting to move MIDI forward would seem to lie in its widespread acceptance. So many devices already incorporate MIDI, an update to MIDI would need to remain backward compatible with existing MIDI devices.

MIDI has never been revised, and thus many of these limitations might only be due to the protocol's age. MIDI would definitely benefit from an increase in speed and with modern architecture this would not pose much difficulty. The issue with attempting to move MIDI forward would seem to lie in its widespread acceptance. Because so many devices already incorporate MIDI, an update to MIDI would need to remain backward compatible with existing MIDI devices.

4.2 Open Sound Control

4.2.1 History

Open Sound Control or OSC was developed by Adrian Freed and Mathew Wright as a "protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology" (Wright & Freed, 1997).

The OSC protocol was designed independently of its method of transport. The protocol is therefore able to be used on standard network technology such as TCP and UDP, running over Ethernet or 802.11 wireless networks. (Wright, 2005)

The OSC protocol has been used to simply encapsulate other protocols (including MIDI messages) for transport over networks, but has also be employed for use in lighting control, Digital Audio Workstation (DAW) control as well as home do it yourself projects. (Wright, 2005) some examples or something.

OSC is becoming increasingly popular in the audio engineering industry due to it being an open protocol. This allows manufacturers to widely and freely implement and use OSC in products, and encourages developers to implement OSC in their applications.

4.2.2 Specification

The protocol defines a data transport format that can be used to send control messages over networks. OSC does not define a set of standard messages for devices to use, but rather encourages manufacturers to create and use their own message sets: Schemas. Schemas refer to the configurations that define the structure of a device. Therefore the address space is application specific. There are no standard schemas at the moment and manufacturers need to define the structure of their device in order to use the OSC messaging standard. (Wright, 2002)

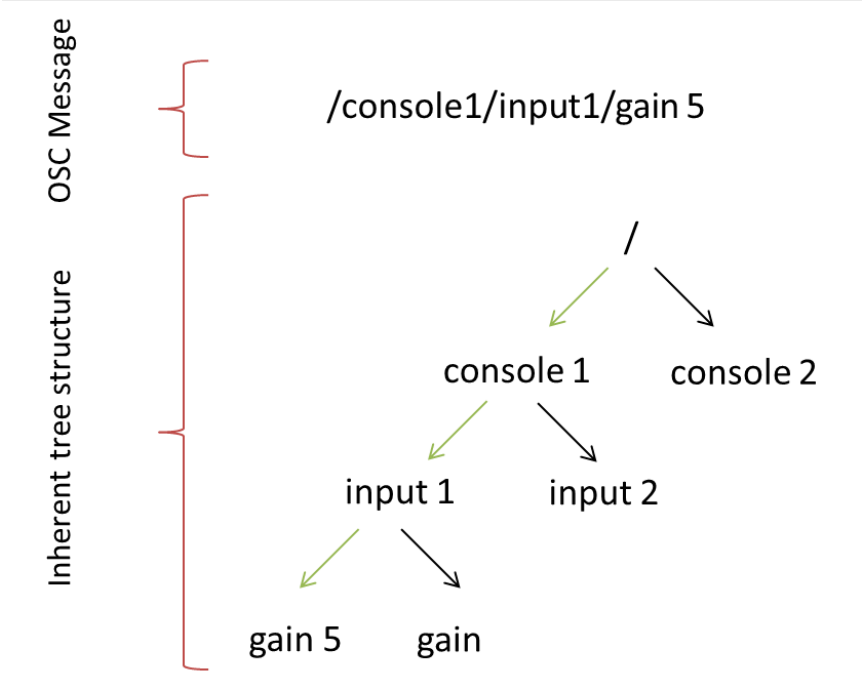

Figure 4.2: The structure of an OSC message.

parsed.

Figure 4.2 shows the layout of an OSC message that would be sent to the OSC server in order to adjust the gain on a mixer to ”5”. The figure also shows how the natural tree structure of the message assists in the representation of the device the schema is describing. (Eales, 2012, Wright, 2002)

4.2.3 Comparison with MIDI

The designers of OSC state that it is superior to MIDI in several ways. The MIDI standard was never intended for use over LANs, let alone the Internet. OSC provides larger address space for messages, allowing for more complex and specific control messages. Modern synthesizers can sometimes exceed the MIDI specification in terms of channels and parameters, areas where OSC is not limited. (Eales, 2012)

As OSC is software based and requires a set of messages and function callbacks to be defined, it lends itself to easier integration into applications in comparison with physical hardware. This does not mean that it cannot be used to provide hardware control though, but rather that the hardware would require some form of system integrated into it that can manage OSC before the protocol can be used. control protocols

Chapter 5

MIDINet: A distributed approach to MIDI control

5.0.4 History

MIDI was originally designed as a means of communication for keyboards and synthesizers, and its simple implementation has led to high uptake by manufacturers. Although the protocol was initially used for the remote control of synthesisers and sound generation devices, it quickly found uses in other areas, such as control for MIDI patchbays, effects units and mixing consoles. As manufacturers started to adopt MIDI and build MIDI interfaces into various devices, it became possible to control a large number of devices in a studio using MIDI control surfaces. (Mosala, 1995)

Studio equipment is expensive and is often bulky, resulting in studio resources having to be tied to a single location. MIDI provides a method of controlling these resources remotely, as well as the ability to control many devices from a single MIDI controller. Rhodes University created a network to provide control for these devices: the Rhodes Computer Music Network (RHOCMN). The network's purpose was to provide an efficient and cost-effective method of sharing music resources. (Mosala, 1995)

Since the inception of the MIDINet system it has been fully implemented Richard Foss(Foss, n.d.) in C++ using MFC.

5.0.5 Concept

Since many devices have MIDI connections and it is possible to provide a computer with a MIDI interface, a computer running control software is capable of providing control for these audio devices. A chief drawback to the current MIDI specification is that the cable lengths are limited to 15m. The MIDINet system was created to overcome these limitations by providing a distributed transport system through the encapsulation of MIDI messages in packets so they can be sent over a Local Area Network (LAN). (Mosala, 1995)

> "The MIDINet system allows the user to look at the collection of his MIDI devices as a single, unified system." (Mosala, 1995)

MIDI devices can be connected to a computer running the MIDINet system, enabling them to control devices connected to other computers on the LAN running the MIDINet system. Each computer is equipped with a network interface controller (NIC) and a MIDI interface card; these computers are referred to as MIDINet units. MIDINet units are interconnected using a standard Ethernet network. (Mosala, 1995)

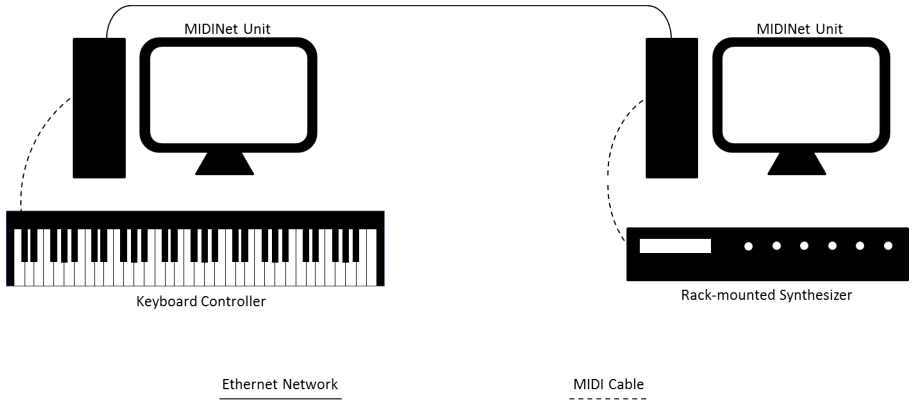

Figure 5.1: Example set-up for a small MIDINet system.

It is possible for computers running the MIDINet system and computers being employed for other purposes to co-exist on the same local area network. This is due to the manner in which the MIDINet system communicates with other machines on the network. When the MIDINet application is started, the host machine sends a join request, adding the machine internet protocol (IP) address to a multi-cast group. The system uses this group for set-up and configuration messages. Messages are sent to the multi-cast group so that only machines running the MIDINet application receive these messages.

5.0.6 MIDINet protocol

The MIDINet system was developed with its own multilayer protocol. Based on the OSI reference model, the MIDINet protocol consists of 3 layers: The application layer, the MIDINet layer, and the Transport layer. (Mosala, 1995)

MIDINet layer model

The Application layer is the top layer, and is the layer with which the user interacts. This layer provides methods of adding and removing connections from the system.

The middle layer is the MIDINet layer, which is responsible for the routing of messages from the application layer to the transport layer.

The transport layer is the lowest layer and is tasked with controlling inter MIDINet unit messaging across the network.

MIDINet messages

MIDINet messages consist of two segments: A code segment and a data segment. The code segment of the message consists of 3 bytes while the data segment's size can vary up to 1497. It is important to note that "the data part can be as large as the minimum length allowed by an Ethernet packet" (Mosala, 1995).

5.0.7 Implementation

User interface

The MIDINet application consists of two windows: A window naming the MIDINet unit, and a window managing connections. When the user starts the application they are presented with a window in which a name can be provided for the MIDINet unit. Figure 5.2 shows the naming window, which is made up of a field for entering text, and a button that allows the application to proceed with the set-up.

Figure 5.2: MIDINet Identification Dialogue.

Once the name for the MIDNet unit has been confirmed, the user is then presented with a window which is used for managing connections between the various MIDINet systems on the network. Figure 5.3 shows the connection management window, which has the main sections which are used for selecting ports and connecting them.

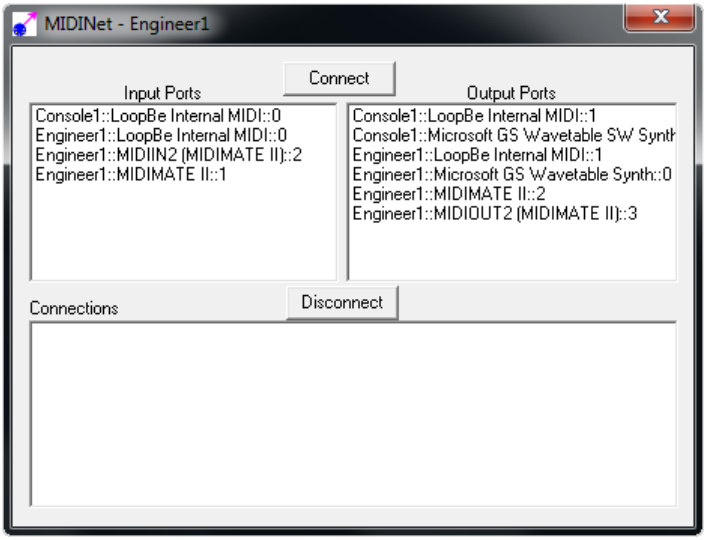

Figure 5.3: MIDINet Connection Management Dialogue.

In Figure 5.3 a list box component can be seen on the upper left hand side where the input ports are listed. The list box is populated with all input ports discovered on the MIDINet system, from both the local and remote MIDINet units. Ports are displayed with the name of the MIDINet unit, followed by the description of the port (MIDI interface). The list box on the upper right of the window functions in much the same manner, but instead with the output ports being displayed. Each line of the list boxes is selectable, allowing for the ID of both the MIDINet unit and port to be used when creating a connection.

Lastly, the lower half of the window contains a list box where all current connections are

MIDINetMsg[0]	HEX	Callback
WHO_IS_THERE	0x01	IamHere(Sender);
I_AM_HERE	0x02	ConfigRequest(Sender);
CONFIG_REQ	0x03	ProvideConfig(Sender);
MNUNIT	0x04	NewRMNUnit();
DELETEMNUNIT	0x0A	DeleteMNUnit();
INPORT	0x04	NewRInPort();
DELETEINPORT	0x0D	DeleteRInPort();
OUTPORT	0x05	NewROutPort();();
DELETEOUTPORT	0x0E	DeleteROutPort();
CONNECT	0x07	NewConnection();
DISCONNECT	0x09	Disconnect();
SYSEXMSG	0x0B	RouteMIDI();
ENDCONFIG	0x0C	none

Table 5.1: Table of ReceiveMIDINet message headers and related callbacks

displayed. The connections are displayed by a MIDNet unit's input port followed by a MIDINet unit's output port. Each of the connections in this list box can be selected so that existing connections can be removed when they are no longer required or necessary.

Network class

The networking class provides a set of methods for initialising the network, handling messages and calling various functions from other classes.

The class makes use of a BlockingSocket library which provides the functions for creating sockets, binding to ports, listening to ports and other socket-related operations. The library uses Microsoft's WinSock library to communicate with the network drivers.

When the network class is instantiated, the network is set up and the process is started. The ConfiguredState variable is set to START [1], which indicates that in the system's current state it is not ready to operate. The networking class then sets up the socket connection for messages by binding to the MULTIPORT [2] and the host's IP address. The machine's IP address is then added to the MIDINet multi-cast group.

The ReceiveMIDINet function is responsible for responding appropriately to all received MIDNet messages. Table 5.1 shows a summary of how the MIDINet system responds to received MIDINet messages.

[1]START is is a preprocessor definition in the MIDINet header file referring to port 2051.
[2]MULTIPORT is is a preprocessor definition in the MIDINet header file.

The network class also maintains the list of remote MIDINet units.

MIDINetUnit class

The MNUnit class is used to form a base class that provides storage of information pertaining to a single MIDINet unit. When instantiated, the class will provide two private data items: A string containing the unit's name and an unsigned integer containing the unit's ID.

The structure provides basic accessors for the data in the form of SetName and SetID functions as well as GetName and GetID. The class is simple and it is required to be generic so that the LocalMIDINetUnits and RemoteMIDINetUnits can inherit their base's members from this class.

LocalMIDINetUnit class

The LocalMIDINetUnit class inherits its base's member from the MIDINetUnit class but also holds its own data items, such as lists, of both the LocalInputPorts and LocalOutputPort held by this MIDINetUnit.

The class contains a CreatePorts function which handles the opening, closing and transmitting of all input MIDI devices on the host machine. These devices are then created as new LocalInputPort objects, opened and added to a list of the MIDINetUnits LocalInputPorts. The equivalent procedure is then carried out for all MIDI output devices attached to the machine.

RemoteMIDINetUnit class

The RemoteMIDINetUnit class contains a number of functions that are required for storing the details of another MIDINetUnit. The class includes functions for adding and attaching both RemoteInputPorts and RemoteOutputPorts as well removing them from the MIDINetUnit.

In a similar manner to the LocalMIDINetUnit's class, its remote counterpart stores two lists containing the input and output ports associated with the unit. Few handling functions are contained within the class as most work is done by the LocalMIDINetUnit before transmitting the unit to other units on the network.

Port class

The Port class provides a basic container for a Port, storing only the port's Name (as a string) and ID (as an unsigned integer). The class provides Get and Set accessors for the port's data items in much the same way the MIDINetUnit class does.

The Port class forms the base of the more specialised LocalInputPort, LocalOutputPort, RemoteInputPort and RemoteOutputPort classes and thus is a more generic class providing on the more basic port Name and ID functions.

LocalInputPort

The LocalInputClass provides port identification through use of the Port class but requires an external library in order to communicate with the lower level MIDI devices associated with these ports. MaxMidiIn is a part of the greater MaxMidi library which provides all the necessary functions for opening, closing, starting, stopping and handling the MIDI messages received by a MIDI input port. The class also holds a data member for identifying the MIDINet unit to which the LocalInputPort belongs.

Packaging of MIDI messages into MIDINetMessages is done by the classes ProcessMidiData function. The function is passed a MIDI message which it initially sends to local ports using the classes SendToLocals call. The function then reads the status byte of the MIDI message in order to determine whether the message is a standard MIDI message, or whether it is a SysEx message. In the call of a MIDI message, the MIDINet message's initial byte is set to MIDIMSG so that when received by the network, the correct callback is triggered. Should the message have been a SysEx message, a similar procedure is applied, but the initial byte is instead set to SYSEXMSG. These messages are then transmitted by the network.

The class also provides a number of functions to GetLocalOuts as well as GetRemoteOuts, functions which add each port to a list. Other functions for Connecting and Disconnecting from output ports (both local and remote) are used to add specific output ports to the relevant lists.

LocalOutputPort

The LocalOutPort class inherits from MaxMidiOut (a part of the MaxMidi library) and the port class for storing the port name and ID. MaxMidiOut provides all the functions

needed to open and close MIDI output devices, as well as output a MIDI message to a port.

The class itself has only a single data member, a pointer for storing the location of the related LocalMIDINet unit so that the port may be associated with a specific MIDINet unit. Apart from the class's constructors and destructor and excluding any inherited member functions, the class has only a single function for returning the ID of the associated MIDINet unit.

RemoteInputPort

The RemoteInputPort class inherits the standard port access methods from the Port class. In many ways the RemoteInputPort class is similar the LocalInputPort class, providing much the same function for Getting outputs, Connecting and Disconnecting from ports.

Unlike the LocalInputPort, the RemoteInputPort does not require any functions for processing MIDI messages, since an instance of the class actually just represents a LocalInputPort attached to the MIDINet unit hosting the specified port. MIDI message handling will be done on the part of the LocalInputPort.

RemoteOutputPort

RemoteOutputPorts use only the Port class as the base class from which they inherit the bulk of their data members and functions. The constructor is used to set the ID and Name of the Port as well as which RemoteMIDINetUnit it belongs to.

Once again few functions are needed as the RemoteOutputPort is really just an identifier for the LocalOutputPort (which takes care of all the processing) found attached to the correct MIDINetUnit elsewhere on the network.

5.0.8 System Start-up and Operation

To comprehensively understand the way a system functions, its start-up and operational procedures must be examined. Gaining insight into the way a system operates will allow for the identification of any shortcomings. These observations will aid in the rectification of these flaws, thereby improving the system.

Start-up procedure

When a user starts the MIDINet application, a new LocalMIDINetUnit is created. This MIDINetUnit will represent the host machine. When the LocalMIDINet class is instantiated, MIDI input and output devices attached to the host are scanned and the respective local input and output port objects are created and stored.

The application then creates a new network object which immediately sets itself up for socket communication by binding to its own IP address and the chosen multicast port. The class also opens a port for sending UPD packets to a multi-cast address and then requests to join the multicast group. The networking class then starts the MIDINet set-up by sending a "Who's there" message to the multicast group. The group then forwards the message to all of its members.

Next, the application creates the Connection Display window followed by the creation of the Identification window. The user then chooses a name for the MIDINetUnit, entering it in the Identification Window's text input box. The application then fetches the name from the Identification Window and checks the network to ascertain if the name already exists. If the name is unique then the application sets the name of the LocalMIDINetUnit and retrieves the unit's unique ID.

Should there be other machines running the MIDINet application on the network that have already gone through the set-up procedure, on receiving the "Who's there" message the other application(s) would reply with an "Im here" message, thus initiating the remote MIDINet unit's port discovery.

Discovery

When a new MIDINet unit is set-up on a network, in order for the unit to learn about other units on the network, the application sends out (via multicast) a "Who's there" message. If there is another MIDINet unit on the same network and it is already configured, it will respond with an "I'm here" message. The LocalMIDINet unit subsequently responds with "Configuration Request". Upon receiving the Configuration Request, the RemteMIDINetUnit begins transmitting its information. After getting its LocalMNUnit object, it transmits the object to the requesting machine. The same procedure is then followed for the unit's InputPorts, OutputPorts, Connections and if present, other known RemoteMIDINet units.

Operation

When a MIDI message is sent from a MIDI device to a computers MIDI interface, it is handles by the MaxMidi library. The library's OnMidiData function fetches the MIDI messge from the device and handles it internally. MIDINet's LocalInputPort inherits funtions contained within the MaxMidiIn library, and overrides it's ProcessMidiData function for getting MIDI messages, which it then repackages as a MIDINet message. The new MIDINet message is then sent to LocalOutputPorts before being transmitted to the network.

When a MIDINet message is received from the network, the networking class then checks the RemoteMIDINet unit for the port. The MIDINet message is then sent to the LocalOutputPort which handles the message. This is done using MaxMidi's MaxMidiOut library to output the message from the device using the Put function.

5.0.9 Limitation of the MIDINet system

The MIDINet system was originally implemented in Visual C++, making extensive use of Microsoft's Foundation Class (MFC). While this MFC implementation is not necessarily problematic, it can only be deployed on machines running Microsoft's Windows platform which results in a severe limitation of the application's target base.

The system was also designed to run on a wired LAN (100mb/s Ethernet network) and had not been intended for wireless networks. This can limit the use of the network as the ability of the system to run over wireless networks would enable the use of laptops and smaller devices such as tablets to run the system. MIDINet messages are sent using the User Datagram Protocol (UDP) which provides a faster communication, yet is connectionless and provides no assurances as to delivery. While UDP is not necessarily a limitation, its use in industry is declining and research into other protocols for MIDINet's communication should be conducted in the interests of an efficient, updated system.

Lastly, the MIDINet system's user interface was designed using Microsoft's forms, which provide a simple and usable application interface. The limitation to this is that the application was developed for desktop computers, therefore proving difficult to use on touch screen computers. As the screen size decreases, interaction with the system's interface becomes more troublesome and the application usability decreases.

Chapter 6

Cross-platform MIDINet: Design and Implementation

6.1 Languages and libraries

6.1.1 Language of choice: C++

The original version of MIDINet that this implementation is based on was initially developed using Visual C++. C++ offers the ability to deal with memory more directly over higher level languages. This is of great value to any application or system that may need to use pointers.

Visual C++ is Microsoft's variant of the language and comes with a range of Microsoft's many built in datatypes and libraries. The language is comprehensive and was a perfect choice for the original MIDINet system as the target systems were Microsoft Windows machines. The primary downfall of Visual C++ is that it cannot be compiled for other operating systems. Because the target device for this project is the Apple iPad, Visual C++ would need to be substituted with another compatible language.

Apple's proprietary language is Objective-C. Apple's Integrated Development Environment (IDE) is XCode, which is capable of creating and compiling both C as well as C++ projects. For this reason C++ was selected as the language in which the new MIDINet implementation would be developed.

6.1.2 JUCE

This project aimed to place the MIDINet system on wireless tablet computers, specifically the Apple iPad. But to port the system to Objective-C would mean that the system would once again be constrained to a specific platform. Instead, a cross-platform implementation has been developed using a toolkit known as JUCE.

JUCE (Jules's Utility Class Extension) (Storer, n.d.b) is a collection of libraries that was designed to make development simpler when creating cross-platform projects. JUCE provides an application called the Introjucer through which both C++ and header files can be created. The Introjucer then allows the developer to choose which JUCE libraries to include and then choose the target platforms to create projects for. In this way developers can write one set of code, and then have that code exported into Microsoft Visual Studio, XCode, CodeBlocks, Linux (through the creation of a Makefile) as well as Android projects.

This allows for these projects to be created and for a single set of source code to be used. JUCE also allows for Graphical User Interfaces (GUIs) to be created (from within the Introjucer) which use JUCE's own libraries, making it possible to create a consistent user interface across various platforms.

Another consideration when choosing JUCE is that the original implementation of MIDINet made use of Microsoft's own standard libraries. With the standard C++ language not including all the functions and data types that are needed for MIDINet, these would need to be replaced with other libraries. JUCE offers many libraries, including networking, audio and graphics, which are required for this cross-platform implementation.

6.2 Development Environments

6.2.1 Visual Studio

Visual Studio is Microsoft's integrated development environment (IDE). The IDE allows for the creation and management of projects developed using a variety of languages. The IDE provides Microsoft's own libraries and Microsoft's Foundation Class which aids in developing custom classes.

Visual Studio was used extensively in the development of the original MIDINet system

as as such has been used in the design and implementation of the cross-platform version of the application.

All testing of cross-platform application deployment targets aimed at the Microsoft system have been compiled and tested using Visual Studio.

6.2.2 XCode

XCode is Apple's IDE which can be used to the development of C, C++ and Objective-C applications. The IDE allows for the creation of project for both Apple's OS X and iOS operating systems. OS X application can be compiled and run natively on Apple OS X machines.

Applications targeted at iOS devices can be run using Apple's iOS simulator. The simulator allows for application to be tested and run on OS X machines without the need for the target device, either an iPad or iPhone. The simulator also allows for applications designed for mobile devices to be tested without the need for an Apple iOS developer licence. The simulator does have some drawbacks though, namely the lack of actual interfaces.

As the simulator is run on an OS X machine, applications can not be tested as they would on a physical device. This project deals with MIDI devices and interfaces, and discovering these in the simulator can be troublesome should applications be designed to use physical interfaces. Virtual MIDI interfaces can be used within the simulator by either opening ports within the application or by using a second application in the simulator at the same time.

For testing of applications targeted for deployment on the iPad or iPhone, testing should be done using the target device. This requires iOS developer licence as application can not be deployed to a device unless they have been signed.

6.3 An alternative to the MaxMidi library

In the original MIDINet implementation, handling of MIDI ports was achieved through the use of an external library called MaxMidi. The MaxMidi is a C library that allows for the discovery, access and control of MIDI ports available to a system. The MaxMidi library provides a general class for handling MIDI tasks, as well as classes for both MIDI input ports and Output ports.

The library, while providing a comprehensive array of MIDI device functions, is only available for use in the Microsoft systems environment. The MIDINet implementation being originally designed for such systems functioned adequately with the MaxMidi library and did not require any other external libraries to enable MIDI functionality. For a cross-platform implementation, an alternative to MaxMidi needed to be found.

6.3.1 MIDI capabilities of JUCE

JUCE is an extensive and comprehensive collection of libraries, and among its libraries geared towards audio are a several MIDI classes(Storer, n.d.a). JUCE provides classes for the the storage of MIDI messages, the management of MIDI devices and even the parsing of MIDI files.

MidiInput

MidiInput is a class that JUCE provides to enable the identification, creation, opening and handling of MIDI inputs. The class provides member functions that cater for all aspects of MIDI input port handling.

```
1  static StringArray findDevices (const bool forInput)
   {
3    const ItemCount num = forInput ? MIDIGetNumberOfSources()
       : MIDIGetNumberOfDestinations();
5    StringArray s;

7    for (ItemCount i = 0; i < num; ++i)
     {
9      MIDIEndpointRef dest = forInput ? MIDIGetSource (i)
         : MIDIGetDestination (i);
11     String name;

13     if (dest != 0) name = getConnectedEndpointName (dest);

15     if (name.isEmpty()) name = "<error>";

17     s.add (name);
     }
19
   return s;
21 }
```

Listing 6.1: findDevices: iOS callback for MidiInput's getDevices.

Before a MIDI device can be opened or used for receiving messages, input devices need to be discovered and enumerated. JUCE accomplishes this by providing a getInputDevices function which returns a StringArray[1] containing the names of all the MIDI devices that were found. These devices can later be referred to using their index.

In order to use a MIDI input, the input needs to be opened. This can be achieved using the openDevice function which is passed the MIDI device's index, and a callback. The purpose of the callback is to provide instruction on how the input should respond to received MIDI messages.

The MidiInput class also provides methods for starting and stopping the MIDI device from receiving.The class also provides fairly standard functions such as the member for returning the name of the device.

MIDI input devices can also be created, although the feature is not available on Microsoft systems.

[1]A data structure provide by JUCE, which stores an array of JUCE strings, and contains various functions for managing the data structure

MidiOutput

The MidiOutput class allows for the connection and handling of output devices. In a similar fashion to the MidiInput class, the MidiOutput class provides a getOutputDevices function for listing the available output devices. The class also provides an openDevice method, which as expected required the index of the device, but this MidiOutput variant does not require a callback to be provided when opening a device.

MidiMessage

this could simply be replaced with JUCE's MIDIMessage data type. The two types differ only slightly in their implementation (as can be seen below).

MIDINet had originally used the LPMIDIEVENT type for the storage of MIDI messages. This type is defined in the MaxMidi header file, as a pointer to a MidiEvent. MidiEvent is another of Microsoft's custom data types and therefore needed replacing in order to become cross-platform.

```
1 typedef struct
  {
3         DWORD dwDeltaTime;
          DWORD dwStreamID;
5         DWORD dwEvent;
          DWORD dwParms[];
7 } MIDIEVENT;
```

Listing 6.2: Structure of the MIDIEVENT type.

Although not identical in structure, the MidiMessage class provides aa similar format to the above mentioned types. All other MIDI reelated work is done by implementing JUCE's oown MIDI functions and so no issues were encountered using the class as a replaccement.

MIDIView: Test application for JUCE's MIDI library

In order to test that JUCE's MIDI functions would provide adequate control over MIDI devices, a test application was developed. The application was developed to be cross-platform so that the JUCE functions could be tested on various platforms for compatibility.

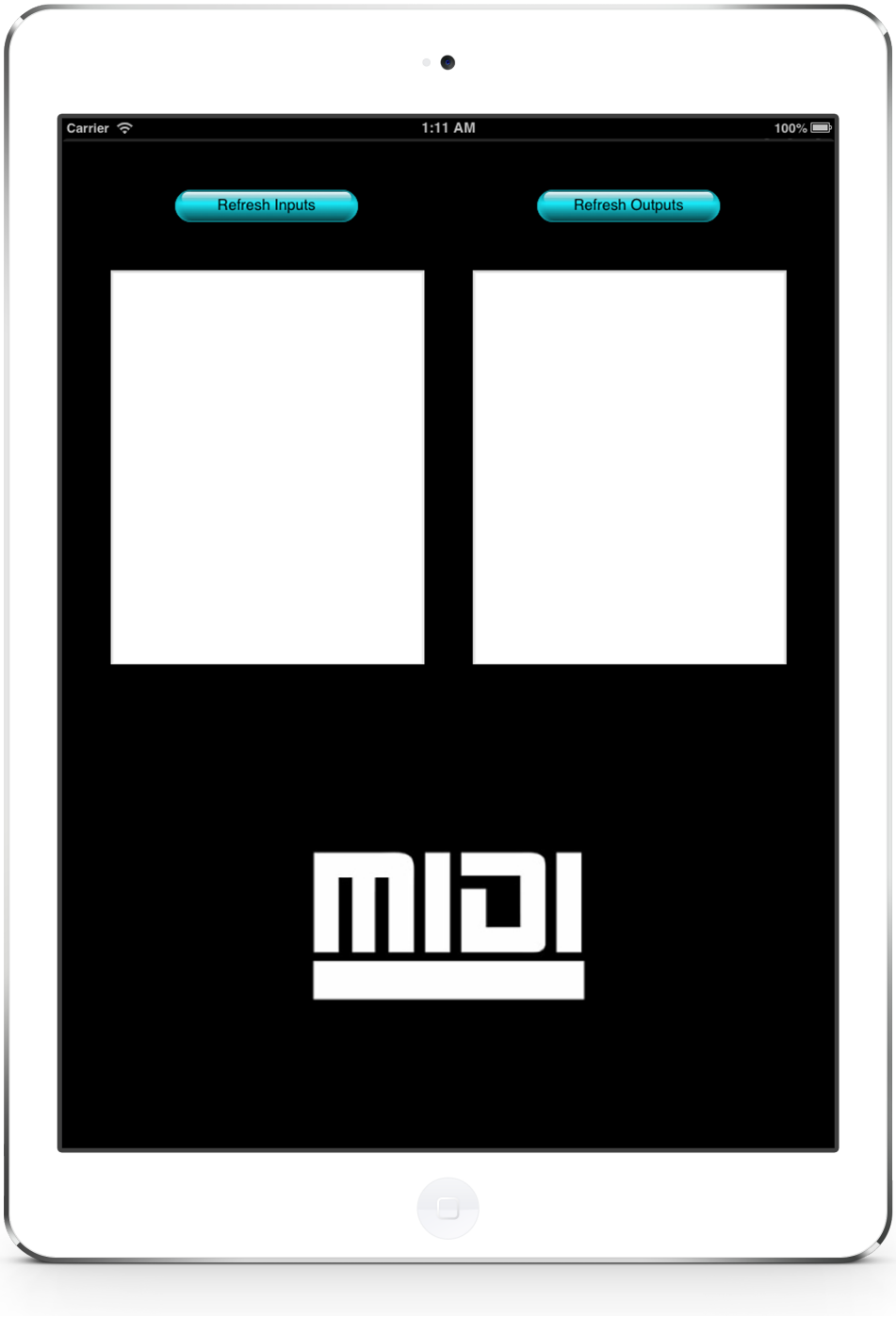

Figure 6.1: Screenshot of Connection Management window.

Figure 6.1 shows the application's user interface. There is a list box on the left of the window which will display all the MIDI input ports discovered on the device. The right of the window has a list box for displaying the MIDI outputs. Each list box has a refresh

button which, when pressed, causes the JUCE MidiInput or MidiOutput class to call the respective getDevices function.

The application was tested on Windows 8, Ubuntu Linux, OS X Lion and iOS 6. MIDI-View performed as expected, displaying all MIDI devices available to the target device. The application has proved that JUCE's MIDI library provides enough control over MIDI devices to be used in the cross-platform implementation of MIDINet.

6.3.2 Development of JMIDI: A JUCE based alternative to MaxMidi

The MaxMidi library consists of three main components, namely the MaxMidi header file, a MidiInput class and a MidiOutput class. As expected, the MaxMidiIn and MaxMidiOut classes handle all functions relating to Midi device inputs and outputs. The MaxMidi header itself mostly contains defines that are needed for the creation of MIDI messages and used in the MaxMidiIn and MaxMidiOut libraries.

Although all calls in the original MIDINet system that used members inherited from the MaxMidi could be replaced with function calls the JUCE library, this would require the modification on many lines of code over many different classes in the MIDINet system. The option to rather create a library that closely matches the MaxMidi library meant that a minimal number of changes would need to be made to the MIDINet system to implement JUCE's MIDI functions.

The new library to be created was named JMIDI[2] and aims to replace the platform specific MaxMidi library. The JMIDI library uses JUCE's basic_audio(Storer, n.d.a) modules to provide cross-platform utility functions for controlling MIDI devices. The new library is designed to simply be a cross-platform, JUCE based implementation of the MaxMidi library, and therefore the new library attempts to have as similar functions as possible in order to increase the compatibility with applications that already use the MaxMidi library.

6.3.3 JMidiIn: Input device control

The JMidiIn class provides a basic container for a MIDI input device and related functions used for controlling the device.

[2]JMIDI gets its name from the functions it performs in managing MIDI input and output devices and from JUCE, since the access control to MIDI devices is done through the use of the JUCE library.

The class has only two data members, namely a pointer to a MidiInput and a String that is used to store the device's name.

BOOL Open(UINT deviceNum)

The Open function is used to open a MIDI input device using the devices index. The function sets up the handler for incoming MIDI messages, storing received messages in a data member held within the JMidiIn class.

To ensure that the port was opened correctly, mDevice (the JUCE MidiInput object) is checked to verify that the pointer is not null. If mDevice contains a null pointer then the opening of the device has failed and the function returns false to indicate the failure. If the device opened successfully and the pointer is not null, the device index is used to retrieve the name of the device and store it in a JUCE String data member in the JMidiIn class.

void Close(void)

Simple function that sets mDevice to a null pointer to assign it.

void Start(void)

The Start function used JUCE's MIDIInput class to start mDevice receiving MIDI messages. The device will continue to read any incoming MIDI messages until it is stopped.

void Stop(void)

The Stop function stops mDevice listening to MIDI message.

String getDescription(void)

The getDescription function can called to return the name of the MIDI device opened by mDevice. This name is set by the device and can not be changed.

6.3.4 JMidiOut: Output device control

The JMidiOut class can be used to provide a container for MIDI output device attached to system.

Same as the JMidiIn class, JMidiOut has only two data members: a pointer to a MidiInput and a String storing the devices name.

BOOL Open(UINT deviceNum)

The Open function in JMidiOut operated in the same manner as the function in JMidiIn. When passed the index of a MIDI output device, the function sets the mDevice pointer to that device.

Should the pointer be a null pointer, the function will return false to signify that the opening of the device has failed.

void Close(void)

The mDevice pointer is unassigned in order to close the class.

bool Put(MidiMessage midiEvent)

The put function is used for sending MidiMessage event to the MIDI output device.

6.4 System Design

6.4.1 User Interface

In the interest of a solid cross-platform implementation of the system, the Introjucer was used to create a GUI for the application. The JUCE graphics library allows for developers to create user interfaces that are visually appealing, without losing their functionality.

The Introjucer allows developers to set a canvas size and whether it is to be of a static size or dynamically resizeable. Components can then be placed on the canvas and moved, resized and have all their attributes modified. A full complement of components is available, including TextBoxes, Buttons, Sliders, Rotary Pots and it even allows for custom created JUCE components to be placed within other components with ease. The Introjucer can also generate callbacks for the various components used to create the interface.

The MIDINet system has only two graphical interface windows: The start-up window (where the MIDINet unit's name is selected), and the connection management window (where input and output ports are connected and current connections are displayed. These two windows did not require any specific redesign graphically, and so were laid out in the same format as the original system. The only real modification to the GUI was that the newer interface no longer consisted of two windows, but rather of a single window to which the two components (NamingComponent and the ConnectionComponent) were added. This was done as devices such as the iPad and other Android and iOS devices do not specifically use windows, but rather tend towards having single screen interfaces that modify contents as the application requires.

The NamingWindow is a simple design, only comprising of a single label, textbox and button. The window was designed with only a single callback for the button click. On the button click, the function was called to create a MIDINet Unit, with only the text contents of the text box in this window being required.

The second window to be created was the ConnectionManagementWindow. This window comprises three main sections. A list of available MIDI inputs is displayed in the textbox located on the left of this window, with a list of MIDI outputs being displayed in a textbox on the right of the window. The window also holds a textbox containing the list of active connections for the MIDINet system. The window only has two buttons which are used to create and delete connections. Callbacks for click events on these buttons are the only

two callbacks that are required.

The Introjucer automatically generated a start-up class for the project. This class is used to make the necessary calls to create the NamingWindow and set up the MIDINet network.

6.4.2 Global modifications

The majority of design changes made to MIDINet are due to the lack of compatible C standard libraries. It became necessary to change Microsoft's own variant on the string data type, CString, to a more common type. JUCE's libraries provide a String data type that is uncomplicated and provides a collection of sensible functions to accompany it. JUCE's String implementation also provides superior compatibility with other JUCE libraries and their functions. For this reason JUCE's String class was chosen for use over the standard C++ string type.

Visual C++ included functions for the creation of message boxes that are used to provide feedback and error information to the user. These messages boxes were originally created using Microsoft's AfxMessageBox function, which is part of the AfxStd library. With this being a Microsoft library (dependant on Microsoft's GUI libraries) it would not be available to be used in standard C++ and would not work on platforms such as Android, iOS, OS X, or Linux.

The AfxMessageBox function calls were replaced with JUCE's own message box implementation.

```
1 static void JUCE_CALLTYPE AlertWindow::showMessageBoxAsync
  (
3   AlertIconType iconType,
    const String &  title,
5   const String &  message,
    const String &  buttonText = String::empty,
7   Component * associatedComponent = nullptr,
    ModalComponentManager::Callback * callback = nullptr
9 )
```

Listing 6.3: JUCE Alert Box function and parameter types

Another type difference that needed to be rectified was that of the lists used by the MIDINet system to store items such as MIDI input and output ports, Remote MIDINet units and connections. These objects were originally stored in Microsoft's CTypedPtrList

lists which form part of the AfxStd library. In the cross-platform implementation, the list types have been changed from CTypedPtrList types to C standard list types.

The problem with this modification is that unlike the CTypedPtrList type which inherits from CObList (a generic object list provides by Microsoft), standard lists in C and C++ do not have built in indexing and iterator functions for traversing the data structure. This meant that every function that uses a list needed to have an iterator that could access the different nodes.

```
1  RemoteInputPort* RemoteMIDINetUnit::FindInputPort(UINT PortNo)
   {
3    RemoteInputPort* ThisRemoteInputPort;
     std::list<RemoteInputPort*>::iterator it;
5    for(it = ThisRemoteInputPort.begin(); it != ThisRemoteInputPort.end(); it
       ++)
     {
7      UINT PortNumber = *it->GetNumber();
       cstring PortName = *it->GetName();
9      if (*it->GetNumber() == PortNo)
         return *it;
11   }
     return (RemoteInputPort*)NULL;
13 }
```

Listing 6.4: Example of a funtion where CTypedPtrList is replaced

Many small type changes needed to be made such as moving from Microsoft's UINT format to a 32-bit unsigned integer. This would not have been a complicated modification, but as the whole MIDINet system only uses one UINT format, a define in the CrossPlatfromDefinitions.h file meant that no text needed be changed. At compile time the UINT value tokens are replaced with the 32-bit unsigned integer's data type.

The CrossPlatformDefinitions.h file is populated with constants that appear throughout the source files. The header file has also been used to create shorter keywords for regularly used functions.

```
1 #define BOOL int

3 #undef FALSE
  #undef TRUE
5 #undef NULL

7 #define FALSE   0
  #define TRUE    1
9 #define NULL    0

11 #define UINT unsigned int
   #define BYTE unsigned char
```

Listing 6.5: Defines to ensure datatypes are correct for all platforms

6.4.3 Networking

The networking class in the MIDINet system is a large body of code, consisting of nearly 1000 lines of code and some dozen functions. As the class was initially only intended to be used on Windows based systems, it relies heavily on Microsoft specific functions and libraries. The WinSock2 library is Microsoft's primary socket communications library and needed to be replaced with a cross-platform alternative. JUCE provides a few networking libraries of its own, including Streaming Sockets, Datagram Sockets and Interprocess Connection Sockets.

The original MIDINet system used UDP datagrams to transmit data over the network. The system, on start-up, instructed the host machine to bind a specific socket to a Multicast address and then issued a multicast join so that the host's IP address could be added to the multicast group.

JUCE, although providing a set of networking related classes, does not however support multicast groups. This meant that the system would not be able to join a host to a group. Sending a message to a multicast group is trivial as the destination IP need only be set to the multicast address, but the inability of the library to provide a method of joining IP addresses to the multicast group indicated that a new design decision needed to be made.

Either a new library that supported multicast would need to be included, or an alternative to multicast would need to be found, either by unicasting iteratively to multiple addresses

or by broadcasting to a specific subnet.

As JUCE libraries were already being used in the project, the choice was made to continue using JUCE's own networking libraries, rather than import alternative networking libraries. The JUCE libraries are cross-platform and so lend themselves to a simple implementation.

The User Datagram Protocol (UDP) is connectionless and is not concerned with ensuring that messages arrive. This implies that UPD is better suited to real time applications than TCP (Transmission Control Protocol) which requires for connections to be made before sending any data.

Datagram Sockets in JUCE are used to send UDP socket messages over the network. The Datagram protocol provide a simple method for binding to ports and setting up sockets.

The networking class was modified to use the JUCE Datagram sockets, only requiring one change to the system. The JUCE Datagram class does not provide any method for determining the IP address or hostname of the sender. This created a small issue with the message handling since when a "Who's there" message is received, the system had no way to determine where to direct the corresponding "I'm here" to.

To compensate for the lack of an adequate receive or read function, a class was created to allow for the sender's IP address and data to be included in the payload of the datagram. The MIDINetPacket class consists of two private variables, the first containing the sender's address (sendAddr) of the JUCE IPAddress type and the second containing a character (char) array in which the MIDINet messages are stored.

This enables the receiving system to direct messages to the correct MIDINet unit. The sending system however needs to be capable of providing its own IP address to be placed in the MIDINetPacket. The task of determining the correct IP Address of the host machine proves to be slightly complicated when considering a cross-platform implementation.

JUCE's IPAddress class provides a method for listing all IPAddresses connected to the localhost. The first IPAddress, should the system be enabled to use internal loopback, will be "127.0.0.1". The second IP address is then the primary IP address, or should loopback not be enabled on a system, the primary IP would be first in the list. This was the case in all tests performed on the target machines during the course of the the project. A new function has been added to the MIDINet networking class that returns the IP address of the primary NIC on the host machine.

6.4.4 DatagramSocket Testing

To ensure that the DatagramSocket class from JUCE would perform as expected, a simple pair of application were developed: DS_Send and DS_Receive. As their names suggest the applications were specifically designed to test the sending and receiving processes. These were developed as cross-platform application so that the test could be run on various systems.

DS_Send

The sending application seen in Figure 6.2 allows the user to enter an IP address and a port that the UDP messages will be sent to. JUCE's DatagramSocket application then connects to IP address and port specified. When the send button is pressed, the message sent using DatagramSocket's write function.

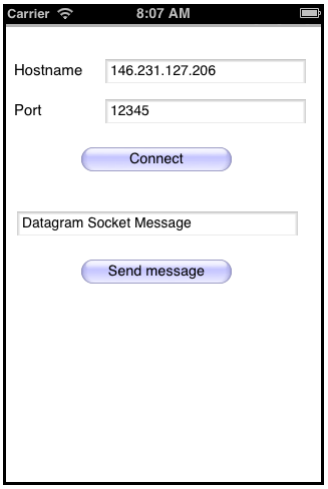

Figure 6.2: Screenshot of Naming window.

DS_Receive

Figure 6.3 show the receiving application. A text box is provided for the user to input a port to be listen to on the local host. When the connect button is pressed the application creates a DatagramSocket. On the create of the DatagramSocket, the port passed to the constructor is automatically bound to. JUCE class also provices a function for binding should one be needed. One the port has been bound to, all messages sent to the port are displayed in the text box of the application.

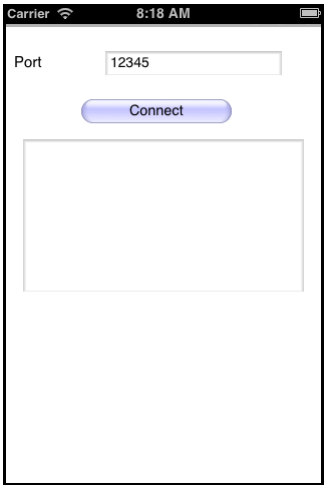

Figure 6.3: Screenshot of Naming window.

6.5 System development

The re-development of the MIDINet system was done by first testing the system in differing scenarios. Each scenario was set up to provide proof that the system could be modified in order to ultimately become a mobile control system for sending and receiving MIDI messages over a wireless network.

6.5.1 Stage 1

The first step taken was to set up a system for controlling the Yamaha 01v96 from a desktop computer. Although no actual development was needed in order to do this, this was an important step in verifying that the desk could be controlled remotely. A machine running Microsoft Windows 8 was set up with the Matrix Mixer application and equipped with an E-MU MIDI USB interface. The MIDI output from the interface was connected to the mixing console's MIDI input to allow the computer to send MIDI messages to the desk.

Using MIDI-Ox, it was possible to see MIDI messages being created by the Matrix Mixer and sent out to the desk. The Yamaha desk receives the MIDI messages and updates its local parameters appropriately. The desk is a good test subject for remote control as its motorised faders react to changes made on the computer by moving the physical fader up or down.

The test was successful, proving that it was possible to control the digital mixing console via MIDI from a desktop application. Level, panning, routing and most standard desk features were tested and all were fully controllable remotely. There were issues sending messages back to the Matrix Mixer application from the desk as the Matrix Mixer seemed to be constantly sending update messages to the desk, even while no parameters on the desktop application were being changed.

6.5.2 Stage 2

The second system that was created used the standard MIDINet system, compiled for Microsoft Windows. The machines used were running Windows XP and 7. The network architecture was that of a standard wired ethernet network.

The set-up was tested by running an instance of MIDINet on each machine. The first machine was loaded with Windows XP and had LoopB1 installed and running on it. LoopBe1 is an application that allows for the internal looping and routing of MIDI messages. The Matrix Mixer application was opened to provide MIDI messages to loop through to the MIDINet instance.

The second machine on the network was set up with Microsoft Windows 7. The machine was also equipped with an E-MU USB MIDI interface that allowed for MIDI messages to be sent in and out of the machine. Once again the USB MIDI interface on this machine was connected the the MIDI ports on the Yamaha 01v96. An instance of MIDINet was started on this machine too.

A connection was made using the MIDINet dialogue allowing the messages from the internal loop on the first machine to be connected to the MIDI out on the second. This meant that messages created whilst adjusting parameters on the Matrix Mixer were sent to the 01v96 over the MIDINet network. Changes to faders on the Matrix Mixer could be seen as the motorised faders of the 01v96 moved as the parameters were adjusted.

This proves that the MIDINet system, coupled with a control application such as the Matrix mixer is capable of providing control to mixing desks via the USB MIDI interface.

6.5.3 Stage 3

For this test a wireless network was created so that testing could be done in a secure and monitored environment. The network consisted of a wireless access point and target machines that were to be the MIDINet hosts. For each host, a set-up similar to stage 2 was implemented, where the first machine was running Microsoft Windows XP and MIDINet, with the Matrix Mixer as the control application.

The second machine was equipped with an E-MU MIDI USB interface for connection to the Yamaha 01v96. The machine was running Windows 7 and MIDINet and MIDI monitoring was done on the machine using MIDI-Ox.

monitored. With the current Windows implementation of MIDINet, messages are pushed out to a multicast address and so the toll on the network is low as packets are limited to a small domain.

Chapter 7

Possible future improvements

The MIDINet system can be a powerful tool when it comes to the control of digital mixing consoles and other studio devices. The movement of this system to a cross-platform implementation allows for the MIDINet application to be deployed on a larger range of devices. Running the application on devices such as tablets allows for the system to be used in a variety of new scenarios, which could benefit from further development of the system and research into other system related areas.

7.1 Smaller MIDINet units

The use of MIDINet to control digital mixing consoles requires a computer be located around 15m from the console itself, due to the MIDI specification regarding maximum cable length. While this may not be a problem in scenarios where the console is in an area that has adequate space for a computer, the need for a computer is a limitation on the system. Smaller devices such as laptops can be used in scenarios where space is less available, however they are still a significant overhead when their price is considered.

Newer computing devices such as the ARM powered Raspberry Pi offer a solution to this. The Raspberry Pi is a small scale (credit card-sized) computer that that can run on a Linux distribution. While the Raspberry Pi is not a very powerful device, running a small 1GHz ARM CPU, it comes equipped with a network interface, USB ports and requires only a 5v (500mA) power supply. The Raspberry Pi also comes equipped with 26 general purpose input and output (GPIO) pins which can be used for interfacing with hardware. This makes the device compact, highly power efficient and capable of being integrated

with equipped with a WiFi module and MIDI (either through the use of a USB MIDI interface, or through the development of a MIDI extension to the GPIO headers.

The Raspberry Pi is also a low cost device, costing only£28 for the version equipped with a NIC and £20 for one without, at the time this paper was written. The cost compared to a fully sized computer or laptop would make it a good candidate for the deployment of the MIDINet system. In the interest of a low cost, small form factor alternative to a computer, MIDINet units attached to console and other devices need not incorporate a display, as connections can be created from any other MIDINet device on the network. Only a small 16 LCD display capable of displaying the name or IP address of the MIDINet unit may be required in order for the device to be identifiable.

Other alternatives to the Raspberry Pi would include microcontroller based hardware such an the Missing Link. The Missing Link is already equipped with a WiFi module, USB and MIDI but lacks any form of display. Despite this, it too would be a good candidate for the MIDINet system. As the Missing Link does not run any operating system, the system would need to be redesigned for use on microcontrollers.

Other devices could also be designed and built to better support the MIDINet system, making this an area that could certainly benefit from further research.

Chapter 8

Conclusion

At the outset of this project, one of the goals was to assess the currently available digital mixing consoles and control applications for these consoles. This information was then used to attempt to understand in which direction the industry is moving in terms of digital consoles, control applications and control protocols.

Manufacturers in the audio industry have produced a range of highly flexible digital mixing consoles that come already integrated with many features. While it is clear that many of these consoles can be controlled remotely, there is no specific standard or protocol that has been accepted by the industry and manufacturers as an industry standard. This has led to many of these consoles using different methods to exert control over the internal parameters.

Although MIDI may be an ageing protocol, it is still widely used in the audio industry. Its use in digital mixing consoles, synthesizers and digital effects units makes distinguishes it as a good choice as a standard for control. The limitations of MIDI lie mostly in the slightly outdated specification: low speed, small address space and restricted cable length. A renewal of the MIDI specification would allow for a wider range of control messages with finer parameter control.

The second major goal of this project was to determine the viability of MIDINet as a wireless cross-platform system capable of controlling multiple digital mixing console from multiple control devices. Research needed to be conducted into how the system operated in its current implementation as well as into its ability to be used for controlling digital mixing consoles.

The MIDINet application, although originally designed to provide a distributed approach

to MIDI, has been found to be capable of providing remote access to more than just musical synthesis devices. When implemented with the Matrix Mixer, the system has been shown to be capable of providing remote control over the Yamaha 01v96 digital mixing console. The system was also tested over a wireless network to ensure that a wireless improvement would be possible.

For the MIDINet systems to be viable for deployment over various target devices, integral parts of the system were required to be proven capable of cross-platform deployment.

The MIDINet system relies on the ability of the application to be capable of interfacing with MIDI devices attached to a system. JUCE's libraries were used as they provide a cross-platform method of accessing MIDI input and output devices on a system. The testing of this on an iPad using the cross-platform MIDIView application indicated that the application could not only access MIDI devices, but also be deployed on a number of systems.

A key component to the MIDINet system is its networking class. For the system to become cross-platform, the networking class was modified to use JUCE's DatagramSockets class for sending UDP messages. The development of the DS_Send and DS_Receive application shows the ability of JUCE's networking library in providing a method of sending UDP messages. The cross-platform applications also prove the ability of sending datagrams to applications running on various target operating systems.

This project indicates that it is possible to create a cross-platform MIDINet system that is capable of controlling multiple digital mixing consoles and that the application can be deployed on a wide-range of devices. It is therefore possible to create a generic, cross-platform MIDINet system with the aim of increasing the mobility and efficiency of sound engineers through the remote control of digital mixing desks.

References

Allen&Heath. GLD Remote. Online. Available from: `http://www.allen-heath.com`. Accessed on 27 May 2013.

Allen&Heath. iLive Digital Mixing System. Online. Available from: `http://www.allen-heath.com`. Accessed 20 May 2013.

Allen&Heath. iLive Editor. Online. Available from: `http://www.allen-heath.com`. Access on 28 October 2013.

Allen&Heath. iLive MixPad. Online. Available from: `http://www.allen-heath.com`. Accessed on 27 May 2013.

Ballou, G. 2008. *Handbook for Sound Engineers*. Handbook for Sound Engineers. Focal.

DeFuria, Steve, & Scacciaferro, Joe. 1990. *MIDI Programmer's Handbook*. Foster City, CA, USA: IDG Books Worldwide, Inc.

Eales, Andrew; Foss, Richard. 2012 (10). Service Discovery Using Open Sound Control. *In: Audio Engineering Society Convention 133*.

Foss, R. *Object Oriented Design Part 3: MIDINet complete design and implementation*. Rhodes University Computer Science III course notes.

Foulkes, Philip James. 2006. *A Grid Patch-Bay for Audio Mixers*.

Hexler. TouchOSC. Online. Available from: `http://hexler.net/software/touchosc`. Accessed on 25 June 2013.

Hosken, D. 2010. *An Introduction to Music Technology*. Taylor & Francis.

Huber, David Miles. 2012. *The MIDI manual: a practical guide to MIDI in the project studio*. Focal Press.

Izhaki, R. 2013. *Mixing Audio: Concepts, Practices and Tools*. Taylor & Francis.

Jacobs, Gabriel, & Georghiades, Panicos. 1991. *Music and new technology: the MIDI connection*. Sigma Pr.

Kirk, Ross, & Hunt, Andy. 1999. *Digital sound processing for music and multimedia*. Focal Pr.

Mosala, Thabo Jerry. 1995. *Routing MIDI messages in a shared music studio environment*.

Rothstein, Joseph. 1995. *MIDI: A comprehensive introduction*. Vol. 7. AR Editions, Inc.

Storer, J. JUCE API. Online. Available from: `http://www.juce.com/api/classes.html`. Accessed on 20 October 2013.

Storer, Julian. JUCE. Online. Available from: `http://juce.com/`. Accessed on 15 October 2013.

Wright, Matthew. 2002. Open Sound Control 1.0 Specification.

Wright, Matthew. 2005. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, **10**(2005/12/01), 193–200.

Wright, Matthew, & Freed, Adrian. 1997. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. *Pages 101–104 of: International Computer Music Conference*. Thessaloniki, Hellas: International Computer Music Association.

Yamaha. LS9 Stage Mix User Guide. http://download.yamaha.com/file/55009. Available from: `www.yamahaproaudio.com`. Accessed on 27 May 2013.

Yamaha. Yamaha 01v96 Digital Mixing Console owners manual. Online. Available from: `http://www2.yamaha.co.jp/manual/pdf/pa/english/mixers/01V96E1.pdf`. Accessed on 15 September 2013.

Yamaha. Yamaha Mixing Consoles. Online. Available from: `http://www.yamahaproaudio.com/global/en/products/mixers/`. Access 26 October 2013.