Harnessing Open Sound Control for Networked Music in VST systems

October 9, 2009

A dissertation submitted to the Department of Computer Science,

Faculty of Science at the University of Cape Town

in fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

Information Technology

JASON HECTOR [B.SC ELEC ENG]

Supervisor

PROFESSOR KEN MACGREGOR

UNIVERSITY OF CAPE TOWN

IYUNIVESITHI YASEKAPA • UNIVERSITEIT VAN KAAPSTAD

"If I were not a physicist, I would probably be a musician. I often think in music. I live my dreams in music. I see my life in terms of music.... I get most joy in life out of music."

- Albert Einstein

Acknowledgements

....to my wife Stacy with love....

I would like to express my heartfelt gratitude for your unending patience and understanding. I appreciate your many sacrifices.

Thank you to my family for the support and encouragement over the years, and for helping me stay focussed.

Thank you to my supervisor, Ken MacGregor, for his advice and guidance given to improve on this work.

I would also like to acknowledge and thank the following people:

- Adrian Freed, Research Director at CNMAT at Berkeley University for the support and direction provided while developing the research ideas of this work.
- Daniel Martin, the developer of jVSTwRapper, who has provided me with assistance on numerous occasions troubleshooting errors encountered in my dissertation prototype work.
- Hanns Holger Rutz for the direction and feedback given while developing the OSC audio data type within the NetUtil library.

Abstract

Professional audio equipment is migrating towards general purpose computers running professional audio software systems such as Virtual Studio Technology [VST] and VST plugins that have been adopted as an informal standard for audio and MIDI plugins[26]. The proliferation of computer networks has facilitated sharing and distributing musical content over networked technology and has benefits of allowing multiple, simultaneous, remote access to audio resources, it allows processing and computer hardware resources to be distributed amongst many computers and therefore has potential for VST processing farms, and facilitates location independent musical collaborations. Open Sound Control is an open high-level application protocol developed to facilitate modern networked communications amongst audio processing units like VST plugins. OSC has been suggested as a successor to MIDI addressing its shortcomings in its design[11].

This dissertation presents a prototype VST plugin called OscVstBridge that bridges the network isolated VST plugin and VST host to the OSC network domain. The prototype facilitates modern, high speed, networked, musical control communications amongst VST audio processes. OSC is an open protocol allowing OscVstBridge to communicate between VST systems from different vendors furthering inter-operability of audio and musical system and promotes standardisation.

This work presents a novel mapping process to facilitate the construction of a rich, hierarchical OSC namespace using the VSTXML definition.

The bridging between VST and OSC required data type implementations including parameter, audio, synchronisation and transport. The audio data type required development in order to transport PCM uncompressed chunks of audio over the network. This new float vector 'v' type is an array of 32bit floats developed within the NetUtil OSC library for Java.

The OscVstBridge required the development of audio receiving control mechanisms that would remove network jitter and preserve packet order when receiving network audio data. The first control mechanism is the JitterBuffer and associated algorithm to order packets. The second uses OSC's intrinsic temporal semantics through the OSC timestamp and OSC Scheduling mechanisms.

This dissertation also present 7 experiments evaluating the OscVstBridge prototype. The first four experiments evaluate the parameter data type. These experiments evaluate against benchmarks of 10ms for both mean latency and peak jitter and show results that the parameter data type performance meets or exceeds these benchmarks. The last three experiments evaluate the audio data type and compares the audio receiving control mechanisms. The audio experiments have shown that the JitterBuffer is the only control mechanism that can transport raw audio data over a jitter-ridden[1] network without degrading the signal. The OSC Scheduling control mechanism does reduce jitter and improve on packet order but does preserve the audio signal when transported over this

[1] with latency of 2ms and jitter between 1 and 100ms

network.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Context

General purpose computers have advanced over the past decade with processing, memory and storage capacity exponentially increasing[8]. The cost of these have however been steady and therefore for the same monetary value today one could obtain an increase in these resources when compared to yesteryear. This therefore facilitated the penetration of general purpose computers into the music and audio industries and has paved the way for the recent shift of professional audio equipment to general purpose computers running professional audio software applications. Such applications absorb most of the functionality found in previous generation hardware-based professional audio systems such as music control and audio routing, mixing desks, synthesizers, effects processing, etc.

Audio software vendors began to harness and design systems that would utilise the increased computer resources available. Certain audio system software development philosophies evolved to give birth to Virtual Studio Technology [VST]. A VST plugin is an audio processing unit contained within a VST host application. The plugin applies its algorithm to the audio received from the VST host before returning it back to the host. The internal operations of the plugin are hidden from the host and therefore appears as a 'black-box' to the host. A VST system consists of VST hosts and VST plugins and has been adopted as an informal standard for audio and MIDI plugins[25].

Audio software developers could then develop VST plugins focussing on the algorithm needed to be implemented within the plugin. They would then simply just need to interface to the VST host and not be concerned with any VST host functions. Similarly VST host developers would focus on the VST host functions and not develop around the wide array of VST plugin processes. VST therefore

allows compartmentalisation of the functions and processes involved in audio software development.

VST plugins are instantiated by a VST host creating a direct link between the two. It can been infered that the underlying hardware platform and associated computer resources are shared between the VST host and all the plugins it instantiates.

Coupled with the advancement of general purpose computers has come the proliferation of computer networks. Computer networks are widely available with general purpose computer hardware standardising on Network Interface Cards [NICS] running the TCP/IP protocol stack. The typical maximum throughput bandwidth is 100Mbits per second and can be found in most Local Area Networks [LANs]. Hardware vendors now however supply Gigabit [1Gbit/sec] bandwidth devices. If the lesser of the above bandwidths are adopted, this bandwidth surpasses the bandwidth requirements for musical control by approximately a thousand times[1].

Sharing and distributing musical content over networked technology has benefits such as allowing multiple, simultaneous and possibly remote access to audio resources, its allows processing and computer hardware resources to be distributed amongst many computers, and facilitates location independent musical collaborations known as Networked Musical Performance [NMP]. These benefits have led to various research into transporting MIDI over Ethernet since MIDI is the defacto standard for musical control[24]. A new MIDI specification called HD-MIDI is capable of transporting MIDI over networked technology. Another protocol called Open Sound Control [OSC] also facilitates the transport of musical data across computer networks, but has in its a design no prescription to any set of messages and therefore has a large variety of application areas.

OSC is an open message-based high-level application protocol developed to facilitate modern networked communications amongst audio processing units. Such audio processing units could include VST plugins. OSC is capable of transporting musical control and audio data and is transport independent. Implementers have developed it using the TCP/IP protocol suite over LAN networks. OSC has been suggested as a successor to MIDI addressing its shortcomings in its design[11].

A VST host contains the VST plugins it instantiates. Communication between the host and the plugin is done via host procedure calls and the two share the same hardware resources. VST plugins that require communication to other audio processes must do this via the VST host application since there is no mechanism that allows them to do this directly. The plugin data such as parameter control, synchronisation, transport and audio data is therefore network

[1] assumption of LAN network at a conservative 31Mbit/s and the MIDI musical control bandwidth at 31Kbit/s

isolated to the VST host that instantiated it. Certain VST host applications do provide networked communication services but uses closed systems excluding networked communication between VST hosts from different vendors.

Commercial VST plugins have become advanced and resource hungry, and with VST hosts capable of hosting more than a hundred plugins simultaneously, the resource demand placed on the underlying computer hardware can quickly escalate, restricting the usability of the VST system.

This dissertation introduces the development of a VST plugin called OscVstBridge that bridges the islanded VST plugin and host to the OSC networked domain. The plugin provides a gateway that shares musical content from different VST hosts running on separate general purpose computers that are networked together. The audio resources shared are parameter musical control data, audio data, synchronisation and transport data. The OscVstBridge plugin provides an open communication platform using OSC between VST hosts and plugins from different vendors.

The requirement to network musical systems for collaboration purposes was a natural development[4]. Audio software vendors therefore enhanced their products to include networking services in and amongst their own product lines. They developed closed networked system such as Steinberg's[2] VST System Link. There however is no open communication platform that facilitates inter vendor communication for musical content. This dissertation works towards establishing such a open platform using the open protocol definition OSC in the widely used audio processing systems of VST. NMP can be deployed on this open framework.

Closed network systems enforce intra-vendor product locking. While this might tie in well with the vendor business philosophy, it limits inter-operability of audio and musical systems. Establishing an open platform contributes towards standardisation and widens the competitive playing field amongst vendors.

Users of such networked systems that are deployed on open frameworks are free to choose amongst the VST hosts they network which could include open source VST hosts. Users within the closed networked systems are forced into a specific vendor's product with licensing required for each software product running on each computer. The open framework therefore could become a cost effective way of networking musical devices.

1.2 Motivation

[2]Steinberg is the founding company of VST

The MIDI protocol standard was first published in 1983 and is presently the most used music protocol worldwide. Computer networking bandwidth has proliferated into the 100Mbit/sec maximum throughput compared to MIDI's 31.25Kbit/sec. This enormous bandwidth difference between presently available computer networks and MIDI has facilitated research into other music and audio protocols that use the high speed network technology available today. Such a protocol is the OSC open protocol definition. OSC therefore has been compared to MIDI[13] and proposed as its successor. In order to achieve this, OSC would have to establish itself within audio processing systems established in industry. Such a processing system is VST which has been adopted as an informal standard for audio and MIDI plugin processing. Therefore the OscVstBridge plugin provides this critical link assisting OSC to establish itself amongst VST and the processing systems used in industry. This plugin development has provided high speed musical networking using OSC to established industry audio systems.

Sharing of audio resources in the OscVstBridge plugin enables computer hardware resources to be distributed amongst networked computers. VST hosts can therefore control VST plugins that reside on separate hardware by transporting the data that usually flows between VST host and VST plugin over a computer network. This therefore delinks the tightly coupled VST host and plugin and decreases the hardware resource requirements placed on individual computers. Since the computer network and the number of computers on that network is scalable, a VST plugin processing farm becomes possible.

All the interfaces in the VST host are available to OscVstBridge and hence the OSC environment through the flexible routing schemes found in a VST host. A MIDI controller interfaced to the VST host can be transformed into an OSC controller by routing MIDI messages within the host to the OscVstBridge plugin. This therefore provides a MIDI to OSC conversion. Similarly OSC to MIDI conversion is possible by receiving OSC messages in the OscVstBridge plugin and sending it to the VST host. The host could then use these MIDI messages to control audio synthesis processes.

Audio inputs and outputs interfaced to the VST host are also capable of being routed within the host to the OscVstBridge plugin. Audio data can therefore be transported between VST hosts using OSC. This audio transportation allows all sources of audio data to be transported to a central VST host whose audio interface output is used to playback the audio data. This can be a cost saving exercise in that a single audio interface is needed and shared amongst all network computers. Synchronisation and transport data can also be transported between VST hosts on different computers allowing musical synchronisation of the networked VST hosts. The transport data allows a single VST host to control the playback position of the audio markers used within all the VST hosts that are networked. These networked VST hosts can be location separated facilitating NMP.

Zbyszynski and Freed[41] address the control of VST plugins using OSC by suggesting that the integration of OSC and VST technologies would improve and further the field of musical networking. Added functionality would be afforded

to VST plugins such as enhanced control structure.

1.3 Objectives of this Dissertation

Zbyszynski and Freed[41] address the flat control structure used within VST plugins stating that the VST parameter namespace is unwieldy. The paper introduced OSC as a means to address the flat namespace creating a hierarchical, meaningful namespace that will aid users and plugin designers. The release of VST SDK 2.4 by Steinberg in 2004 also addresses the unsorted, flat namespace of VST parameters with the introduction of the VST parameter structure XML definition (VSTXML). VST hosts that support this definition will construct a hierarchical namespace based on the VSTXML definition supplied with the plugin. The definition is backward compatible with pre-SDK2.4 VST plugins[36].

The OscVstBridge plugin therefore aims to construct a rich, hierarchical OSC namespace using the VSTXML definition.

Zbyszynski and Freed affirm the research ideas and intention of this dissertation work by stating the following:

> "If OSC support was integrated into VST plug-ins, much of this namespace could be built automatically"
>
> "Products by Native Instruments,Reaktor and Intakt, already support OSC in their standalone applications, but it is not possible to address their plug-ins directly using OSC."
>
> "..control can be further improved if the plug-in could be directly controlled using OSC"

It became apparent when conducting the literature review that an AudioReceiver process would need to be developed as part of the functionality of the OscVstBridge prototype plugin. Other work such as Voice over IP [VoIP] require a buffer between the asynchronous receiver of network packets and the synchronous audio playback. Such a buffer is called a JitterBuffer. The OscVstBridge therefore developed a JitterBuffer in the AudioReceiver process to remove network jitter introduced into the audio stream. The OSC protocol also contains a jitter reduction scheme called OSC Scheduling. This scheme is also developed and implemented in the OscVstBridge prototype plugin..

The primary objective of this dissertation was to create a VST interface that bridges VST data to the OSC networked domain. It therefore followed that a prototype VST plugin be developed interfacing VST and OSC technologies. The essential VST data types are the parameter and audio data types and therefore the prototype plugin required such data types to be developed as OSC types. The parameter data type could be accommodated in the standard OSC types

but the audio data type required the development of a new type containing an array of float types.

A secondary objective of this dissertation work was to construct a rich, hierarchical OSC namespace using the VSTXML definition. The design of the OSC namespace was done through this construction process by mapping VSTXML data to an OSC namespace within the development of the prototype plugin.

Upon completion of the prototype, experiments were conducted on the OscVstBridge prototype plugin in order to evaluate its performance. The experiments evaluate each of the parameter and audio data types. The parameter experimental results were compared against industry benchmarks. The audio experiment results have evaluated and compared the AudioReceiver control mechanisms deployed in the OscVstBridge plugin.

From the above objectives the research questions below are outlined:

1. *Can we use Open Sound Control [OSC] as an open protocol interface to implement networked communications between VST systems and OSC environments facilitating distributed audio processing?*
2. *Can we build a rich OSC namespace using the VstXml definition?*
3. *How can we measure and evaluate system performance based on QOS and industry accepted benchmarks for control [parameter] data?*
4. *To determine and compare the system performance in applying different QOS control mechanisms on audio data?*

1.4 Design and Methodology

The methodologies adopted in this dissertation consist of the design and development of the OscVstBridge prototype plugin followed by experiments evaluating the prototype.

The essential core features of the prototype to be developed are:

- a VST Functional Wrapper constituting a VST Host Spoofer,
- a data bridge between OSC and VST domains,
- OSC Data Type development including OSC Control parameter, VST Time Info and developing an OSC audio type,
- OSC Environment development including OSC client and OSC servers including VSTXML definition,
- Audio Receiver development implementing a JitterBuffer and OSC Scheduler to accommodate network jitter.

The prototype development aims to answer research questions 1 and 2.

The experiments were sectioned into parameter and audio experiments. The parameter experiments have metrics of mean latency, peak jitter, scheduling and packet loss. Upper and lower bounds were determined for each metric and the upper bound was compared to industry benchmarks.

The audio experiments metrics are scheduling, latency, jitter and temporal fidelity. The AudioReceiver control mechanisms were evaluated for each metric in order to compare them against one another.

The experiments aimed to answer research questions 3 and 4 by evaluating the system performance of the prototype.

1.5 Structure of this Dissertation

- **This chapter** introduces the topic giving background and context. Chapter 1 also presents the aim, motivation, objectives and design methodologies of the research.
- **Chapter 2** provides an overview of related research covering Distributed Audio Systems and Networked Music, Open Sound Control, VST systems, Networked Communications and Quality of Service.
- **Chapter 3** describes the development process of the OscVstBridge prototype plugin developed. Descriptions of the core components comprising the OscVstBridge plugin are presented such as main classes, VST host spoofer, OSC libraries, OSC data type design, data flows, AudioReceiver, OSC environment, user interaction and threading.
- **Chapter 4** presents the evaluations and results of experiments conducted on the developed OscVstBridge prototype. The first 4 experiments evaluate the parameter data type and the last 3 the audio data type.
- **Chapter 5** concludes the dissertation by discussing the results obtained in Chapter 4 and provides a summary of this dissertation. Future directions in this research area are also recommended.

Chapter 2

Literature Review

2.1 Introduction

Musical control and networking has been using the MIDI protocol as a standard since its inception in the early 1980s. MIDI still has concrete roots in today's musical networks but network technological advancements are tipping the scale towards establishing a new musical protocol standard with improved resolution and bandwidth.

OSC is such a protocol that was developed to facilitate high speed modern networked communications amongst audio processing units. Amongst the most common audio processing units is the VST audio and MIDI plugin. The integration of OSC and VST systems therefore seems a natural development.

The Literature Review contains the following sections:

- **Distributed Audio and Networked Music** describes networking musical systems and how this is achieved through the use of protocols. The MIDI protocol is also overviewed and a Network Musical Performance case is presented.
- **Open Sound Control** gives a detailed overview of the protocol, dissecting its elements. Recent research into embedded devices for OSC, uOSC, is outlined together with serial transfer using RFC1055. The rapid prototyping and development tools in Max/MSP are presented. Related research work such as the Kroonde, MATRIX and Circular Optical Object Locator are presented together with some OSC implementations.
- **VST Systems** is explained giving a conceptual understanding of this system. Control of VST plugins using OSC[42] is also discussed.
- The **Network Communications** section compares circuit and packet switched networks, explains the OSI model framework, the IP protocol

suite, common network architectures and a brief history of the Internet. Musical data in packet switched networks is explored and includes real-time audio. Real time Transport Protocol is outlined as related work.

- **Quality of Service** explains its metrics in the context of Distributed Audio. The metrics discussed are latency, jitter, packet loss, scheduling and temporal fidelity. Control requirements in audio systems are explored and the control mechanisms of OSC timestamping and jitter buffers are explained.

2.2 Distributed Audio and Networked Music

The area of professional audio covers music recording and editing, sound for motion picture and radio production that has been implemented, amongst others, on special-purpose systems called digital audio workstations [DAWs][1]. However, technological advancement has paved the way for the recent shift of professional audio equipment to general-purpose computers running professional audio software applications. Such applications absorb most of the functionality found in previous generation hardware-based professional audio systems such as control and audio data routing, mixing desks, synthesizers, effects processing, etc. However the "software studio" coupled with its intensive processing requirements is most often still localised to a single computer.

> "Many areas of professional audio involve several users sharing audio resources" Anderson et al[1]

Such resources are audio data, control data, computer processing, shared memory, etc. This sharing of resources can be categorised broadly as Distributed Audio and is achieved through communication networks. A simple demonstration of a professional studio with basic point to point communication can be seen below in Figure 2.1.

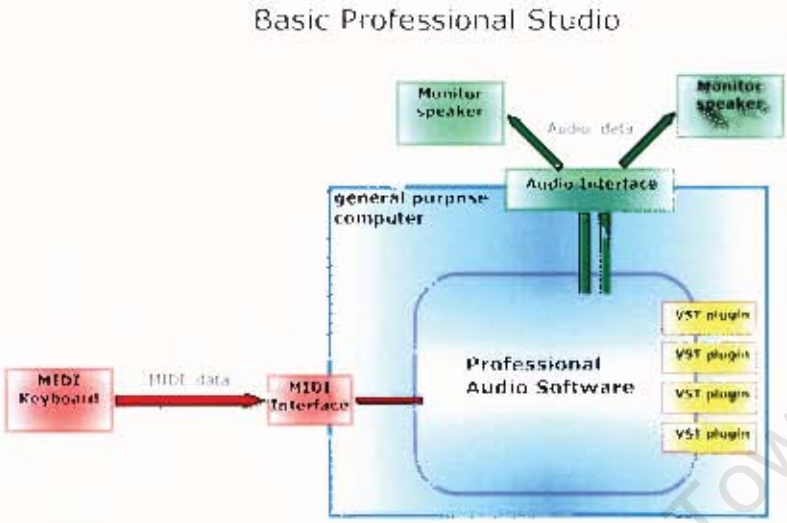

Figure 2.1: Basic professional studio with point to point communication

The general purpose computer running professional audio software will have an audio interface outputting an audio analog signal destined for monitor speakers. A Musical Instrument Digital Interface [MIDI] controller output is also connected to a MIDI input of the computer that is sourced into the audio software. This is a basic point to point network carrying MIDI protocol to transport control messages between the computer and MIDI controller. MIDI has a physical transmission length limitation of less than fifteen meters[15] but does facilitate remote operations between the audio software and MIDI controller and could be considered a Distributed Audio system.

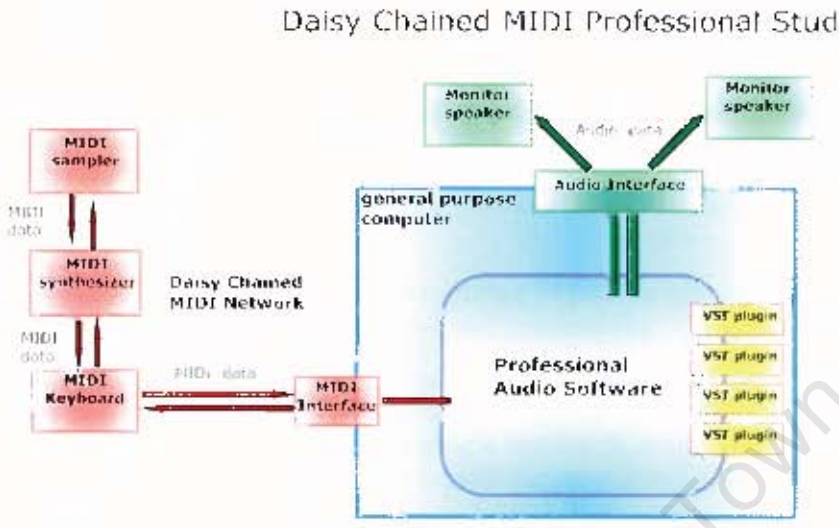

Figure 2.2: Professional Software Studio with Daisy Chained MIDI communication network

Point to point networks can be expanded to daisy-chain the MIDI communications to form the network as in Figure 2.2. This expanded system resembles a networked system but still contains its distance limitation of fifteen meters between MIDI devices. It also is restricted physically to exclusively using the MIDI protocol which was designed for use in local communication networks.

To achieve a true Distributed Audio System the distance limitation should be removed but still function as if all the resources were available locally. The system should leverage open established protocol standards used in network technologies. Such networks include the Internet that uses the IP Protocol Suite which allows for remote users of the system to be scattered globally. General purpose computers can accommodate NICs that implement the IP protocol stack and allow for connectivity to Ethernet networks. They thereby allow for Distributed Audio to be practiced on and amongst "software studios" using standard open protocol technologies up until the application layer defined in the 7 Layer Open System Interconnection [OSI] model. There, however, has been limited contributions made to establishing a open protocol at the application layer for control and audio. The communication model is therefore incomplete when it comes to Distributed Audio.

The benefits of providing an open application layer inclusive protocol definition that can be used over Ethernet networks in a Distributed Audio fashion are numerous. These include multiple and possibly simultaneous access to audio resources, and remote access to resources that allow users to be distributed throughout the network. This applies globally with reference to the Internet. It allows for audio processing to be shared and distributed amongst many computers. This therefore increases the overall processing capability of the system

and reduces the computational load placed on a single computer[6]. This is of incredible benefit to the "software studio" whose processing requirements are high when based on general purpose computers.

Other benefits provide location independent musical collaborations such as NMP[21] to take place provided that the connecting network surpasses the minimum Quality of Service [QOS] requirements set out for the collaboration. Musicians interact over a network that provides transparency rendering performance similar to a local interaction.

> "...we should use computer networking to provide new and unique forms of musical communication and collaboration." Wright[39]

The interaction of musicians over computer networks for distributed collaborations to produce a musical result seemed an inevitable step[4].

An NMP case of two pianists playing a four-hands composition over the Internet has been conducted[21]. One person was located at UC Berkeley campus while the other 64km away at Stanford campus. Each piano responded to both the local and remote players and remote musical control information was shared between the campuses over a CalREN2 data link. The data link had a round trip time of 4.2ms and was considered not sufficient for audio streaming due to bandwidth limitations and lack of QOS.

NMP clients were used by each pianist and a conference server co-ordinated communication between each client. Gestural control information was shared between clients using the IETF Real Time Protocol (RTP) with MIDI messages as its payload.

The NMP experiments evaluated latency and packet loss. Qualitative experiences found the system usable and playable as a musical instrument.There were however some negative effects due to network congestion. These were that depressed keys do not sound immediately and released keys do not stop sounding immediately.

2.2.1 The need for protocols

The underlying mechanism for achieving a successful Distributed Audio system is a protocol. A protocol is a set of rules defined for communication between nodes in the network that support the exchange of data[34]. The TCP/IP protocol suite is amongst the most important set of Internet-based protocols developed. It consists of the physical, network access, internet, transport and application layers which constitutes a subset of the OSI model[34]. The TCP/IP protocol suite is amongst the most widely used in the world.

2.2.1.1 MIDI

Within the audio and music industry the MIDI protocol has established itself as the de facto standard for musical control. MIDI is found in musical performances, synthesizers, audio interface cards and commercial audio production software[24]. MIDI has been instrumental in facilitating communications within

professional audio since its inception in the early 1980s. Prior to that there was no universal protocol for remote control of musical parameters[2].

A complete description of the MIDI specification can be located on the website *http://www.midi.org/about-midi/spechome.htm*. A brief overview of the specification is outlined within this document.

The 3 facets of MIDI are the communications protocol language, the distribution format for standard MIDI files and the hardware interface.

Serial protocol

MIDI is a serial protocol with a byte-sized command set. At its physical layer it communicates at a baud rate of 31250 baud asynchronous and therefore transmits each byte message in 320 microseconds. The recommended connectors are 5-pin DIN that is a 1.5mA current loop. MIDI devices contain either 2 or 3 ports. These are the IN, OUT and THRU ports. The IN port receives MIDI messages into the device and the THRU port replicates the IN port data as an output port for daisy chaining purposes. The OUT port is used for messages created within the MIDI device itself.

The MIDI Bus

The MIDI addressing scheme is achieved through the MIDI channels. 4 bits are allocated to channel addressing and therefore 16 channels are defined in the standard. Each MIDI message contains a destination channel and although all MIDI devices on the bus will receive this message, only the device which matches the channel address will process the MIDI message. This makes MIDI bus topology possible.

Message Format

The message length can vary depending on the type of message but usually is either two or three bytes long. System exclusive messages are an exception being possibly longer than two bytes since they are vendor specific messages. The first byte contains the status information in the first nibble and the channel number in the second. The first nibble values and their corresponding status information can be found in the MIDI spec. The second and third byte of information is dependent upon the type of message specified in the status nibble.

As an example a MIDI message that specifies for the middle C note to be set on channel 1 at full velocity will resemble the structure:

[(STATUS NIBBLE)(CHANNEL NIBBLE)][KEYNOTE NUMBER][VELOCITY]

and can be viewed on the wire as hexadecimal:

90 3C 80

The status nibble in the 90 hex first byte is 9 or 1000 in binary and indicates Note On message type. The channel nibble is 0 indicating user channel representation number of 1. The keynote number of 3C hex is a decimal note number of 60 or middle C note. The velocity in the last byte of 80 hex is 128 decimal indicating full velocity.

THE EVOLUTION OF MIDI

The transport of MIDI was specified as 5-pin DIN connector in the original specification. However technological advancement in communication media has made provision for MIDI to be utilised over transport mediums such as Firewire, USB and Ethernet. Commercially available products have Firewire and USB integrated into their MIDI products already.

HD-MIDI is a major extension of the original MIDI specification that provides greater resolution to data values as compared to the original 8 bit values, has an increased number of channels available for use and support for a new type of MIDI message. HD-MIDI is backward compatible with MIDI 1.0 messages.

There has been significant research contributions into transporting MIDI over Ethernet networks. This makes NMP possible with musical collaborations over the Internet[21].

OSC is an application layer protocol that has implementations in musical control. There a numerous applications that use Ethernet as the transport mechanism and OSC has been suggested as a successor to MIDI addressing its shortcomings in its design[11].

2.2.2 Data Types of Distributed Audio

Musical control data such as those transported by MIDI is the most important data type for use in Distributed Audio. The bandwidth requirements in transporting musical control is much lower than transporting audio data, even in a compressed format. MIDI musical control has a bandwidth of 31.25kbits/sec for 16 separate channels. For a typical MIDI instrument that accepts a single MIDI channel input and outputs a stereo 16 bit uncompressed PCM audio channel pair with a sample rate of 44100 samples per second, the bandwidth requirement of 16 stereo audio channels would be 22.6Mbits/sec [44100 sample rate x 16 bit x 32 channels] which is roughly 717 times more than the MIDI musical control bandwidth. Conversely we can say that MIDI control data is 717 times more efficient than transporting uncompressed audio provided that the destination device for MIDI messages is capable of processing them into audio data. MIDI and musical control data would therefore seem a likely choice for transporting musical information in bandwidth limited networks.

Audio data transportation opens up many system architectural avenues when transported on an open framework. Since audio itself is a core resource produced and consumed in musical performances and professional audio studios, there are benefits in transporting this resource over a computer network. Audio interfaces are no longer limited to a local machine and can be shared amongst networked devices.

Open Sound World (OSW) is a visual development and run-time environment[9]. Sound designers and musicians can process sound by using expressive real-time control widgets called transforms. OSW is similar to the Max/MSP and Pd visual programming environments in that they all allow for patching of visual components together. OSW supports OSC and has transforms that process OSC data. Of particular interest is the processing of uncompressed PCM audio

data by OSW using OSC as the transport protocol. OSW defines a new non-standard OSC type tag 'v' that has an array of floats as arguments. OSW is the only program to support this feature of transporting raw audio data using OSC. Figure 2.3 illustrates OSW.

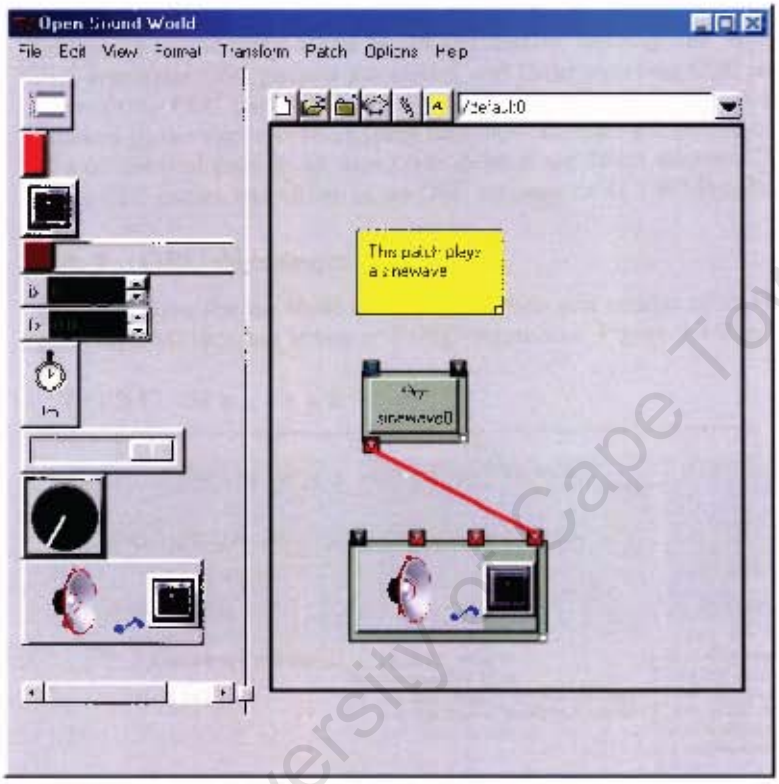

Figure 2.3: OSW with sinewave transform

2.3 Open Sound Control

2.3.1 Introduction

Open Sound Control is a message-based high-level application protocol developed to facilitate modern networked communication among audio processing units. These include computers, sound synthesizers, digital audio workstations, multimedia devices, etc[38]. OSC is an open protocol and does not prescribe any specific set of messages that must be implemented. It therefore has a large variety of application areas and can transport control data as well as audio data, and implementations of each are found in Lemur controller and OSW[9] respectively.

OSC is transport independent and has found implementers developing on

the TCP/IP protocol suite using mostly User Datagram Protocol [UDP] but also Transmission Control Protocol [TCP].

2.3.2 OSC Packets

OSC employs the client-server model architecture defining that any application that transmits OSC packets are clients and those receiving OSC messages are servers[40]. OSC packets are the OSC data units and consist of its contents followed by the size in bytes of those contents. The size will always be a multiple of 4 or 32 bit aligned as all data types defined are 32-bit aligned. The contents of an OSC packet can either be an OSC message or an OSC Bundle.

2.3.3 OSC Messages

OSC messages are the basic units of OSC data and consist of an OSC address pattern, OSC type tag string and OSC arguments. Figure 2.4 illustrates this.

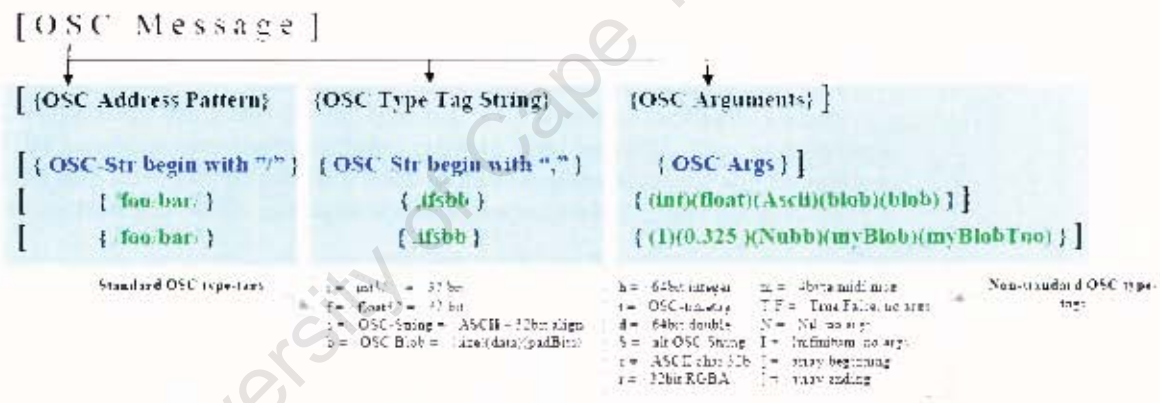

Figure 2.4: OSC Message Structure

2.3.3.1 OSC Address Pattern

The OSC address pattern uses an URL-type naming scheme, begins with a forward slash "/" and is used as the destination address marker for the OSC message. An OSC address part is the substrings between the adjacent pairs of "/" and after the last "/"[38]. The OSC address pattern is human-readable text lending indication to a description of the OSC message. For example an OSC address pattern of the following:

```
/OscVstBridge/Parameter/Filter/Cutoff
```

indicates that the message is destined for the device **OscVstBridge** (the prototype bridging OSC to VST), and is a **parameter** element of **filter** type controlling the **cutoff** of that filter.

OSC Address Patterns are used instead of OSC addresses in OSC messages to accommodate special characters for expression pattern matching in OSC servers. If an OSC address pattern of an OSC message matches an OSC server's address for a particular leaf node within the OSC address space, then the appropriate OSC method is invoked in the server. This allows for a single OSC message to dispatch multiple OSC server methods simultaneously.

2.3.3.2 OSC Type Tag Strings

The OSC type tag string begins with a comma (,) followed by a sequence of characters. Each character type tag gives the data type used in the argument of the OSC message. The OSC arguments sequence follows the type tag sequence exactly. The four official type tags defined are 32-bit integer numbers (defined by character "i"), 32-bit floating point numbers (defined by character "f"), ASCII strings (defined by character "s") and 'blobs' which are blocks of binary data (defined by character "b"). There are also non standard OSC type tags that accommodate data types such as MIDI messages, RGBA color, timetags, etc. A comprehensive list can be seen in Figure 2.4.

It is of particular interest that a OSC type tag is defined for the MIDI musical protocol. In a simplistic configuration MIDI data could be encapsulated within the OSC protocol using this defined type tag. This therefore facilitates transport of MIDI messages over the underlying network used by OSC. This underlying network uses the TCP/IP protocol stack in most implementations and therefore makes transport of MIDI messages across Ethernet and the Internet possible.

2.3.3.3 OSC Arguments

OSC arguments are a single contiguous block sequentially representing the data described in each of the data types defined in the OSC type tags. The range of arguments are from zero to many.

An example of a type tag and its corresponding arguments are shown below:

```
OSC type tag(char)   -> ,  f  m  ()
OSC type tag(hex)    -> 2C 66 6D 00
OSC arguments(hex)   -> 43 DC 00 00   90 3C 80 00
```

It can be seen that there is a 32-bit floating point number and a non-standard MIDI OSC type tag as found after the comma. There is also a zero pad byte to align the atomic OSC units to be 4 byte aligned. The OSC arguments are also 32-bit aligned with the floating point number receiving two pad bytes after its value. This is followed by the MIDI byte based message which is three bytes long followed by a single pad byte.

2.3.4 OSC Bundles

An OSC Bundle is a sequence of OSC messages or OSC bundles themselves allowing for nesting of OSC data. OSC bundles begin with the string "#bundle" to denote that the message is of this type. Following the bundle string is

a OSC timetag consisting of a 64-bit message that uses the same format as Internet Network Time Protocol (NTP)[23]. NTP provides up to 200 picosecond accuracy[32].

After the OSC timestamp the bundle contains OSC bundle elements which comprises of its size and contents which may either be OSC Messages or other OSC Bundles.

All messages within an OSC Bundle should be processed atomically. The synchronisation technique used between the client and server in the communication architecture falls outside of the responsibility of OSC protocol definition. OSC does however assume that the client and server within a communication link are synchronised. The OSC Bundle timestamp is an absolute future dated time giving notification to the OSC server of when to process the message. However if the timestamp has already elapsed then the message is to be processed immediately. This timestamp property is important for correct scheduling of packets in the server and provides an intrinsic jitter compensation mechanism.

Figure 2.5 illustrates OSC bundles.

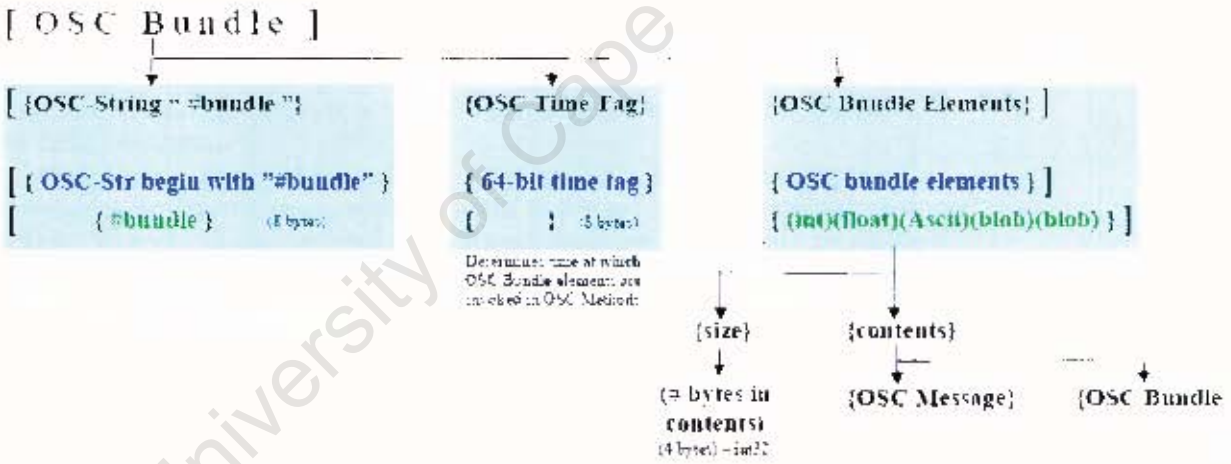

Figure 2.5: OSC Bundle Structure

2.3.5 OSC Address Spaces and OSC Addresses

The OSC server contains a set of OSC methods that are addressable by OSC messages. These OSC methods are arranged in a hierarchical tree-like structure called the OSC address space as seen below in Figure 2.6.

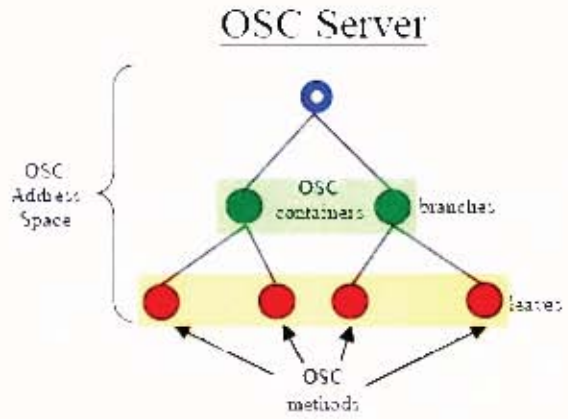

Figure 2.6: Hierarchical tree structure of the OSC address space

The tree structure comprises branches called OSC containers that are not directly addressable by OSC messages. These branches can be likened to the *parts* of the OSC address pattern in OSC messages. The leaves in the tree structure contain the methods of the OSC server which are the points of control for OSC messages.

OSC does not prescribe the architecture to be used in the OSC address space. Developers are free to design address spaces that meet their application needs without considering any protocol restrictions when using OSC. Other protocols such as MIDI and ZIPI do not afford this open architecture[40].

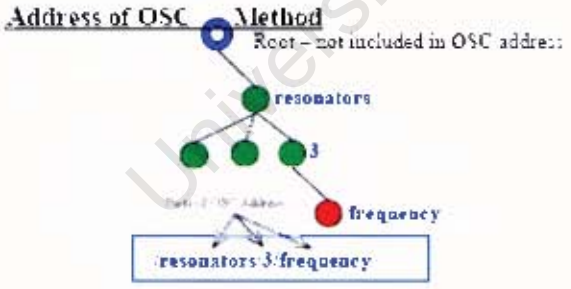

Figure 2.7: Example of an OSC address space

Figure 2.7 gives an example of a OSC address space. The OSC address is */resonators/3/frequency*. The first part 'resonators' is the first branch encountered when traversing the tree from the root. This node has 3 child nodes all of which are OSC containers. The '3' child node is of interest and it in turn contains a child leaf node called 'frequency'. Since it is a leaf node it has an associated method attached to it that can be invoked by using the OSC address specified above.

2.3.6 OSC Message Dispatching and Pattern Matching

OSC message dispatching is illustrated in Figure 2.8.

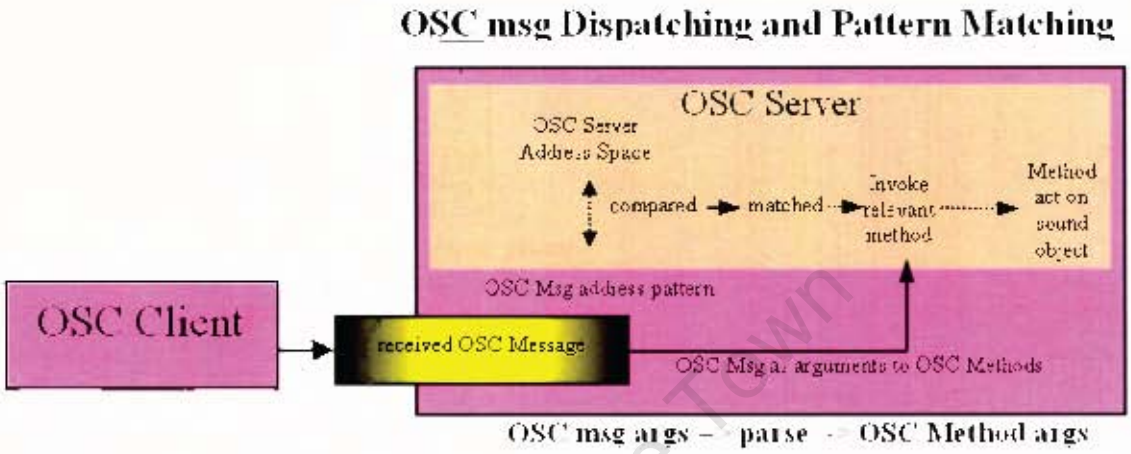

Figure 2.8: OSC Message dispatching

In a typical transmission of an OSC client sending OSC messages to an OSC server, the OSC message address pattern will be examined by the OSC server. If the OSC address pattern matches one or many of the OSC addresses defined in the address space, then those methods would be invoked[1]. When the methods are invoked the OSC arguments from the OSC messages are passed as arguments into the methods allowing the method to act on the appropriate sound object.

Pattern matching is matching an OSC address to the OSC address pattern. An OSC address pattern can contain special characters that conform to certain syntax rules defined within the OSC 1.0 specification. A summary of this syntax is found in Figure 2.9.

[1] this could be numerous methods

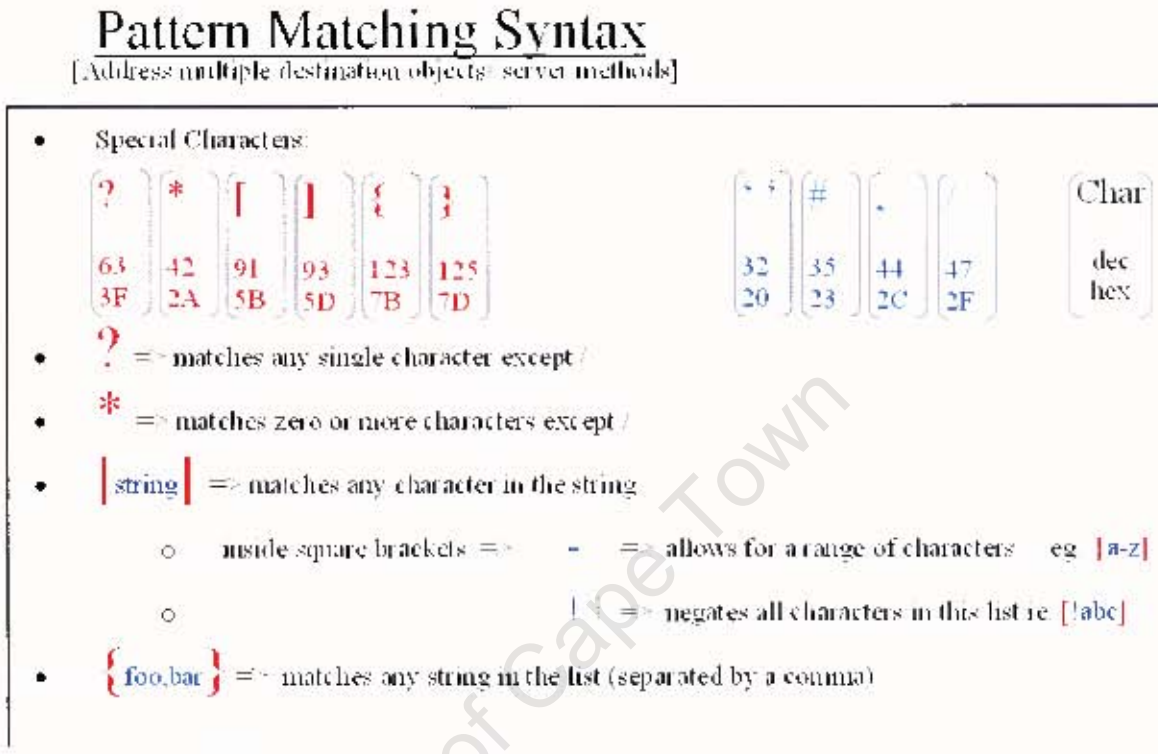

Figure 2.9: OSC Pattern Matching Syntax

2.3.7 OSC queries

OSC queries are client requests to the OSC server for certain aspects of its information. Such information could be exploring the address space, requesting messages type signatures, documentation on the server and current parameter value queries and can be seen in Figure 2.10.

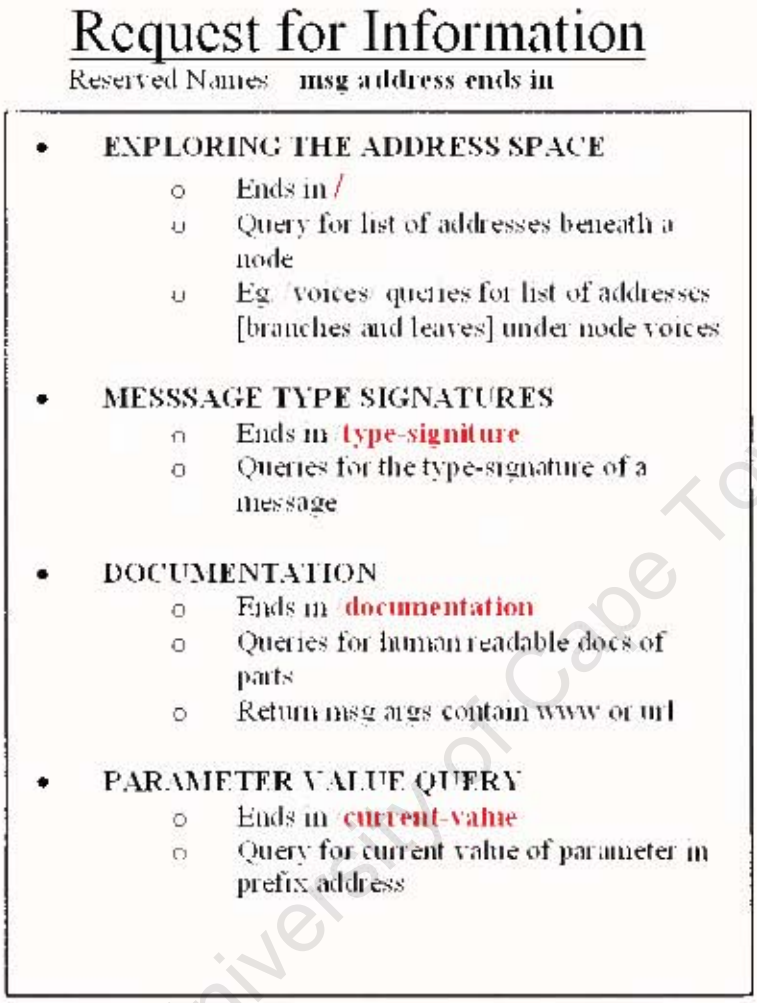

Figure 2.10: OSC Queries

2.3.8 The benefits of OSC

OSC is optimised for modern network technologies such as 100MBit/s Local Area Networks. For musical control this bandwidth allocation is generous and therefore affords OSC the opportunity to include in its definition 32-bit data allocation for numeric values. Its also uses picosecond accuracy on timestamping OSC bundles.

If OSC is deployed on Ethernet networks capable of broadcast services, then OSC packet duplication is afforded to OSC by this underlying network. Multiple receivers of OSC packets within the same network broadcast domain will receive this packet and process it. When the data payload is related to synchronisation

or transport data, a single message originating from a Master device can control numerous slave hosts.

The management of polyphony within a sound generating device can be achieved through the OSC pattern matching feature. A single OSC message can control numerous voices within the polyphonic arrangement simplifying the control parameter for the end user.

OSC has an intuitive naming convention with naming of parameters being a self descriptive URL-like style. This affords composers the opportunity to map meaningful names to events. Comparing this to MIDI, composers would have to interpret bytes within the MIDI message using lookup tables to determine a parameter mapping. This URL notation also creates an unlimited number of mapped points and maintains a tree-like hierarchical scheme for easy point navigation.

OSC has four standard data types and numerous non-standard types. Within the standard types the bit depth is 32-bit, enhancing resolution when compared to MIDI.

OSC Timetags are a powerful mechanism capable of removing jitter introduced by the network. By using a technique called forward synchronisation[32], the OSC client can timestamp messages for execution within the server at a future time based on the expected latency of the network. This creates an implicit buffer within the server as the messages are queued before being dispatched to OSC methods. The timestamping also provides scheduling services ordering packets in the correct sequence as transmitted.

2.3.8.1 Implementations

OSC has ever-increasing implementations across many application areas. Computer programming languages that now support OSC are C, Java, Javascript, Perl, PHP, Python, Ruby, Matlab and Smalltalk. Web graphic systems that support OSC are Macromedia's Flash and Director. Interactive processing languages that support OSC are Pd, SuperCollider, Bidule, CPS, Max/MSP, Open Sound World, Virtual Sound Server, Csound, Reaktor and Traktor.

The *Kroonde*[35] contains wireless sensors such as light, magnetic field, pressure and acceleration and converts the sensor data into OSC over UDP. The *MATRIX* ("Multipurpose Array of Tactile Rods for Interactive eXpression") interface by Dan Overholt[27] has a 12 x 12 array of spring mounted rods that are sampled at 30Hz transmitting the sensed data via OSC.

MIT Media Lab has a project[17] analysing real-time sound properties. The analysed data is converted into OSC control messages that are sent to another machine performing sound synthesis.

The *Circular Optical Object Locator[14]* at Stanford's CCRMA has a rotating platter with digital video cameras analysing the opaque objects placed on the platter. Image processing software analyses the objects and rotation feeding OSC messages to another machine that synthesizes sound.

UCSB's CREATE developed "high-performance distributed mutimedia" and "distributed sensing, computation and presentation" systems[30] that consist of

multiple sensors and trackers that process, interpret and render audio and video data. These systems communicate their data using the OSC protocol as well as CORBA.

Other software that also supports OSC are SonART, Picker, SpinOSC and EyesWeb.

2.3.8.2 Networked Music

> "The Open Sound Control protocol has facilitated dozens of such innovative networked music projects"[39]
>
> "OSC addresses our need for a network protocol usable for interactive computer music that could run over existing high speed network technologies such as Ethernet"[39]
>
> Categories of Networked Musical systems are:

- **Heterogeneous Distributed Multiprocessing on Local Area Networks** are machines in the same location used collectively to share processing and achieve a single goal or task.
- **Peer-to-peer LANs** are computers in the same location that operate independently. They are however networked and communicate with one another.
- **Wide-Area Networks [WANs]** are geographically separated and perform independently of one another. They are however networked to share information.
- **Single Computer** contains many processes and threads that are capable of communicating with one another using OSC

OSC is capable of transporting a range of standard data types and non-standard data types. This includes musical control data structured in a format of the designer's choice or the non standard type of MIDI. OSC is also capable of transporting audio data by developing a vector float data type in OSC. This therefore allows for any of the above networked musical architectures to be deployed using OSC.

Recently there has been developmental efforts into establishing a reference platform for embedded devices using OSC[33]. This freely available reference platform is called uOSC[2] and provides a performance focussed, cost effective implementation for hardware musical controllers. uOSC runs on the Microchip PIC family of microcontrollers that are USB 2.0 ready. The microcontroller of choice is the readily available PIC18F2455-based "bitwacker" that costs no more than R300. Other PIC controllers in the same family line can also be used if further IO or serial ports are needed. The firmware reference implementation is capable of scheduling and timestamping, a rarely implemented OSC feature.

[2] pronounced "micro-OSC"

The reference implementation would use USB2.0 in the transport and lower layers. This is unfamiliar as many OSC implementation have Ethernet and TCP/IP as the underlying transport mechanism. SLIP is used to provide the necessary framing needed to demarcate OSC Bundle boundaries within USB serial transport.

Max/MSP was the first programming environment to implement OSC[40]. This implementation was done by Matt Wright by creating the Max/MSP 'externals' called *OpenSoundControl* and *OSC-route*. The *OpenSoundControl* external is capable of transmitting and receiving OSC packets to and from Max/MSP data and buffers, but requires an *otudp* external if the OSC transport mechanism is UDP. The *OSC-route* external implements OSC's pattern matching feature by parsing OSC address patterns. OSC's bundle mechanism is also implemented in the *OpenSoundControl* external.

Two new Max/MSP externals implementing OSC timestamps are the *OSC-timetag* and *OSC-schedule*. *OSC-timetag* interfaces to the system clock providing NTP format timestamps. The *OSC-schedule* external uses an insertion-sorted priority queue to buffer OSC packets before they are to be scheduled to take effect.

This set of OSC externals simplifies OSC development and prototyping in the Max/MSP programming environment.

Serial transport of OSC data requires an additional layer to provide datagram framing[32]. The recommended layer protocol is SLIP (RFC1055) and has been implemented when TCP has been used as the transport layer. File I/O is a serial transport and also requires the use of SLIP as the framing protocol. OSC Scheduling is needed to recover from the high-jitter transport of file reading and can also be manipulated to enable time-machine operations such as variable rate playback.

2.4 VST Systems

A Virtual Studio Technology (VST) plugin is an audio processing unit contained within a VST host application. The host application is oblivious to the internal operation of the VST plugin with the interface between the VST plugin and VST host passing audio and control data. Generally speaking a VST plugin accepts an audio stream and applies its internal algorithm to that stream before passing it back to the host. Parameter data within the plugin is outside of the host's control and is maintained by the plugin.

The processing engine used by the plugin is usually the same as the VST host as well as all other plugins the VST host instantiates. A professional software studio package that is capable of hosting VST plugins, like Cubase SX from Steinberg technologies who developed VST, utilises a substantial amount of computer resources. Some commercial VST plugins are resource hungry as well, and with such VST hosts capable of hosting more than one hundred plugins, the resource demand placed on the computer can quickly escalate restricting

the usability of the VST system. A requirement therefore to distribute the processing load of such systems amongst many computers has developed within the audio industry despite the well documented prolific increase in processing power of computers over the past few years.

VST plugins have been adopted into industry as an informal standard for audio and MIDI plugins[25]. VST technology is found on PC and MAC platforms. Recent developments have seen commercial vendors package industrial computers in tidy 19 inch racks[3] solely for the purpose of processing VST plugins and moving processing away from the VST host processor. An example of such a system is the Receptor processing unit by Muse Research[31].

VST plugins can be categorised into VST audio effects, VST instruments and VST MIDI effects. VST audio effects process audio streams adding studio effects like reverb, delay, flanging, chorus, etc to the audio data. These types of audio effects can be chained by the VST host application. VST MIDI effects allow for processing of MIDI signals within the plugin before they have their effect on some sound object like a synthesizer. VST instruments model real world instruments like synthesizers, samplers, guitars, pianos, etc. They accept control inputs such as MIDI and perhaps audio data and process these inputs as a hardware device would before outputting the audio stream back to the VST host. Research into modeling of classical instruments and synthesizers has facilitated incredible advancement in the responsiveness and realism of such plugins to the extent that the music production software studio using VST plugins has steadily replaced the hardware studio.

Parameter data within VST plugins are control nodes that influence the audio processing algorithms. MIDI control data sent from the VST host to the plugin is destined for some internal parameter before being executed within the audio process. The GUI of the plugin will have controllable widgets that are linked to parameters. Parameters are 32-bit floating point numbers that fall within the range 0.0 to 1.0 inclusive.

[3] so as to resemble audio production hardware

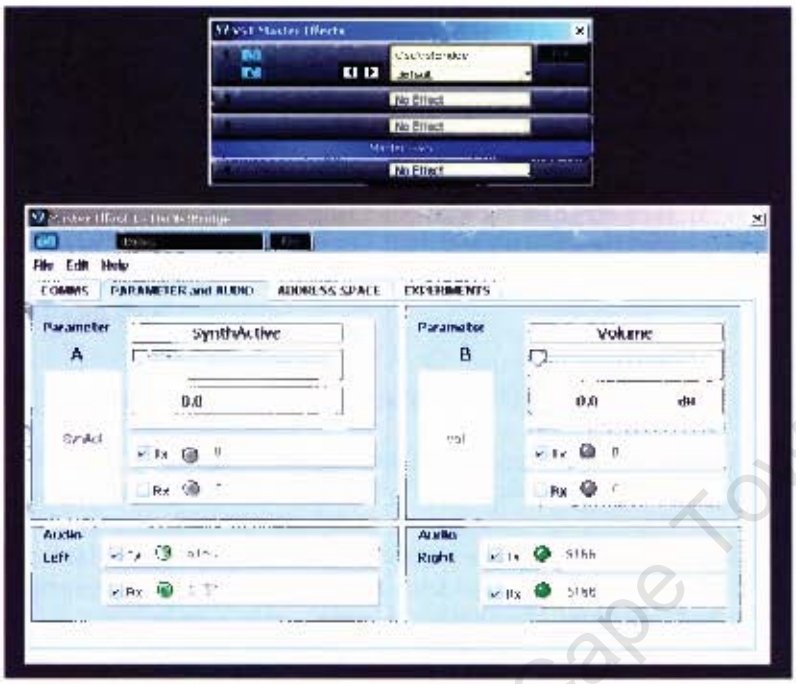

Figure 2.11: VStack VST host hosting OscVstBridge plugin

In Figure 2.11 the VST host called V-Stack by Steinberg Media Technologies is hosting a single master effect plugin. This plugin is called OscVstBridge and is the prototype developed for this dissertation work. The host provides a window for the GUI of the plugin in which users may interact and change parameter settings of the plugin in real-time.

Zbyszynski and Freed[41] propose the control of VST plugins using OSC to address the limited, flat namespace of VST plugins. Controlling VST plugins using OSC also allows higher level controls, parameter name and label aliases in order to improve usability.

OSC is proposed to create a rich intuitive hierarchical namespace in VST plugins as well as inherit pattern matching and simultaneous multiple parameter addressing functionality. Multiple parameters can therefore be controlled by a single OSC message using pattern matching.

> "OSC affords users of VST audio plugins simplified yet flexible control of the plugin parameters."[41]

The release of VST SDK 2.4 by Steinberg in 2004 addresses the unsorted, flat namespace of VST parameters with the introduction of the VST parameter structure XML definition (VSTXML). VST hosts that support this definition will construct a hierarchical namespace based on the VSTXML definition supplied with the plugin. The definition is backward compatible with pre-SDK2.4 VST plugins[36].

VST plugins are inextricably linked directly to the VST hosts that instantiated them. While VST host can run numerous VST plugins simultaneously, a single plugin instance can only be linked to one host. The host communicates to the plugin through exclusive procedure calls and does not employ any communication technologies. Therefore the only possibility of musical networking between different plugins is through the VST host application. Musical networking between VST hosts are only achieved through proprietary communication protocols that usually only constitute intra-vendor communications. Inter-vendor communication developed on an open musical protocol is yet to establish itself.

2.5 Network Communications

Communication networks link computers together in order to share information. The general communication model would have a data source with a transmitter in the source system. The data is then sent via a transmission system to the destination system which comprises of a receiver and a data sink. This model disguises many complex facets of a communication network such as interfacing, routing, signal generation, error detection and correction, flow control, synchronisation, message formatting, network management,etc. This area of work falls outside the scope of this project, but further reading material can be found in Stallings[34].

2.5.1 Circuit switched and Packet switched networks

Circuit switched networks involve a dedicated link being established between two end points. The link is temporarily exclusively reserved between the two end points with nodes in-between switching physical links to establish the circuit. The circuit is terminated when either of the two end terminals closes the link down. The resources required to establish the circuit are available irrespective of whether data is being transferred across the link. Computer applications typify bursty data transfer and hence such circuit networks are inefficient for computer application networking.

Packet switched networks are better suited for computer networked communications in that it improves the efficiency of bursts of data. A communication channel is shared amongst users instead of dedicating a link to a pair of end users. Messages are broken into a series of packets that are transmitted from node to node through the network. Each packet contains a source and destination address and a sequence number that are used at nodes of the network to determine the path the packet must traverse through the network to reach its destination. Messages could constitute multiple packets that could traverse the network taking different paths. When the packets reach the destination the protocols ensure that the packets are arranged in order based on the sequence number and that the message is reconstructed. The Internet is a global packet switched network.

2.5.2 The OSI model framework

The exchange of data between two devices can become rather involved. This task of exchanging data is broken up into several layers which define the protocol architecture. Each layer performs part of the function of the data communication and relies on the layer beneath it to perform more primitive functions. These functions are hidden from the layers above it.

The Open System Interconnection (OSI) reference model developed as a framework for developing protocol standards[34]. The seven layers defined in the OSI framework are:

- Application,
- Presentation,
- Session,
- Transport,
- Network,
- Data link, and
- Physical.

The **Application layer** provisions network services to the application programs. These services include clients and servers. The **Presentation layer** maps syntax into an external data form facilitating correct interpretation of data at the receiving end. This could include data encryption and compression. The **Session layer** controls the communication sessions between the users. It establishes and terminates connections. The **Transport layer** provides data transfer between the end devices in a reliable and transparent manner by utilising error recovery and flow control. The **Network layer** provides the switching technologies used and is responsible for the remote delivery of packets. This layer handles data routing, congestion and flow control as well as packet fragmentation. The **Data link layer** sends frames of data from one physical link to another. It provides synchronisation, error control and flow control. The **Physical layer** defines the bit stream transfer over the electrical or mechanical connections. It handles the mechanisms to control access to the physical medium.Figure 2.12 illustrates the OSI reference model framework sourced from Parziale[28].

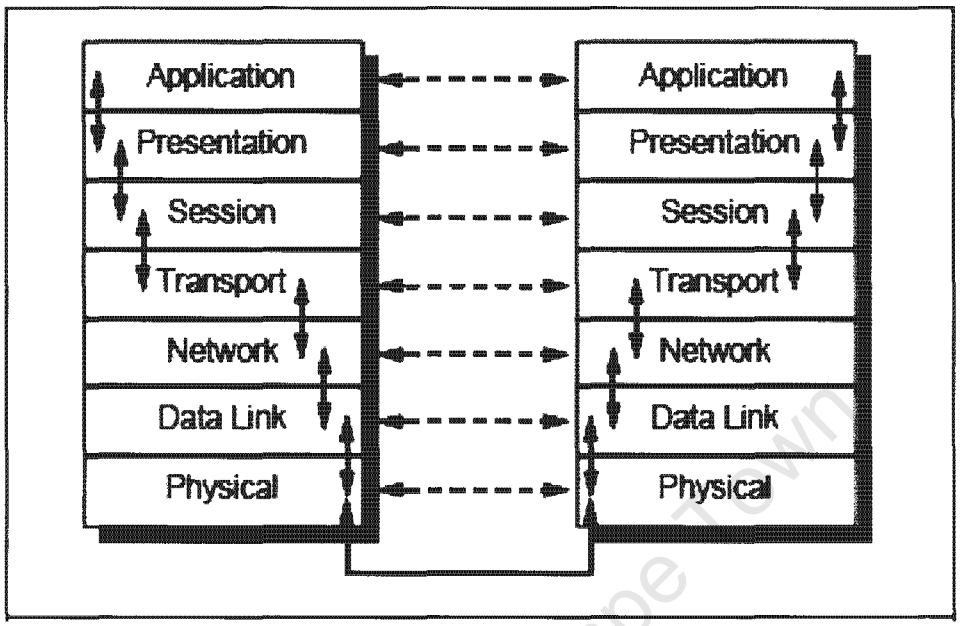

Figure 2.12: The OSI reference model framework

2.5.3 The IP protocol suite

The IP protocol suite is not just the TCP and UDP protocols but is rather a collection of protocols that have been issued as Internet standards. It consists of four layers defined within the OSI model framework and can be seen in Figure 2.13. It has a **Network Access layer** that covers the physical interfaces used between the device and the communication medium. It is implemented in the NICs and comprises the Physical and Data link layers in the OSI model. The NICs employ medium access methods such as CSMA/CA, token ring, etc. The protocols that apply to this layer include Ethernet, Asynchronous Transfer Mode (ATM), ARCNet, Frame relay and Fibre Distributed Data Interface (FDDI). Other protocols having effect within the Data Link layer are Serial Line Internet Protocol (SLIP) and Point-to-Point Protocol (PPP). The data unit defined here is the frame. The **Internet layer** is responsible for the routing of packets over the network using packets as the data unit. The Internet layer equivalent in the OSI model is the Network layer. The dominant protocol used here is Internet Protocol (IP) however other protocols such as Address Resolution Protocol (ARP), Reverse Address Resolution Protocol (RARP) and Internet Control Message Protocol (ICMP) are also used. The **Host-to-Host layer** equivalent in the OSI model is the Transport layer. The Host-to-Host layer is responsible for data integrity and packet sequencing and ensures the reliable delivery of data. The two most prominent protocols are UDP and TCP. UDP is connectionless and hence unreliable but incurs very little overhead. TCP on the

other hand is a connection oriented protocol with error control and improved protection against data loss. It therefore carries more overhead than UDP, is slower than UDP and not suited for transportation of real-time data[3]. The **Application layer** in the IP protocol suite combines the Application, Presentation and Session layers in the OSI reference model. This layer provides the interface between the user application and the IP protocol stack. Some common protocols include File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), TELNET, HTTP, NTP, etc.

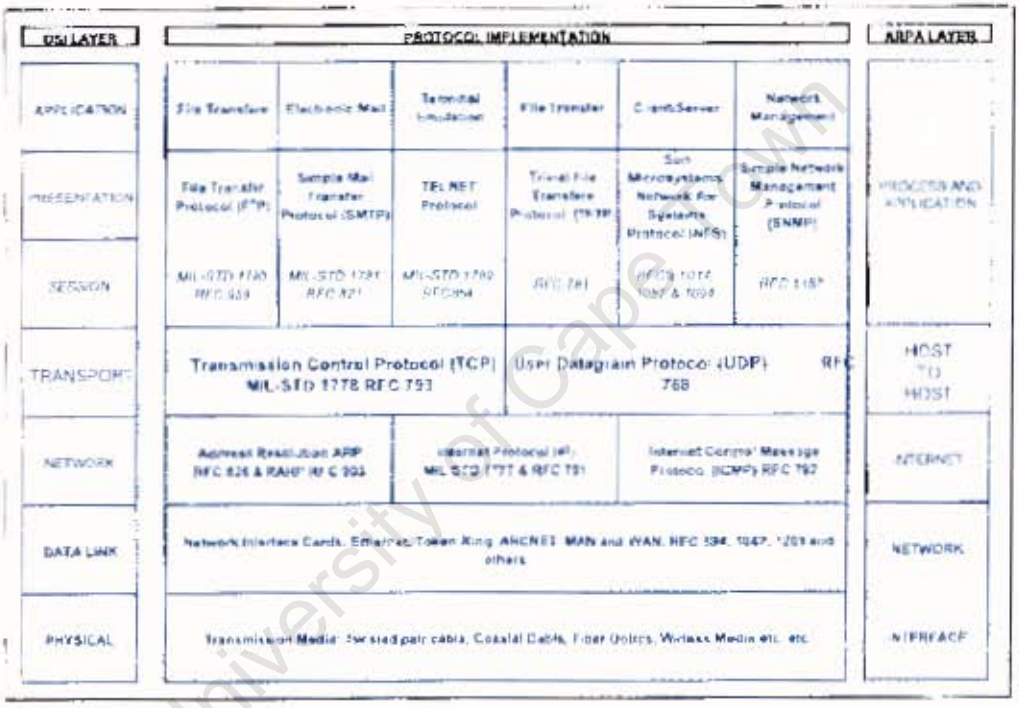

Figure 2.13: IP protocol suite within OSI framework

2.5.4 Network Architectures

The client/server model is amongst the most prevalent model architectures in use today within the Internet. Web browsing, email and file transfer all utilise this model in delivering their services to the user via a server. A server provides a service by replying to a client requesting such a service. The client aspect is usually fulfilled by the user as with using a Web browser or client to request a service (in this instance a Web page) from a Web server. Servers usually deploy their service on a specific IP port and clients therefore address their request to the IP address of that specific server coupled with the IP port that service is running on. This combination of the IP address and IP port is called a IP

socket.

In music collaborations the architectural models used are the Centralised Network Model, the Distributed Peer-to-Peer Network Model and a combination of the two[3]. The Centralised Network has servers maintaining the state of the system while the clients handle processing. The Distributed Peer-to-Peer Network is server-less making no distinction between servers and clients. This network is a point to point distributed system with each user handling its own state of the system.

2.5.5 The Internet

The Internet was developed by the U.S. Department of Defence as the Advanced Research Project Agency (ARPA) and was called ARPANET back in 1969. It was the first fully operational packet switched network. The number of users in the Internet today are over a billion in 200 different countries[34]. The TCP and IP protocols provided the foundation for the Internet. The Internet connects host systems to LANs that connect through a router. Routers connect networks and forward packets on towards their destination. The destination address is an IP address which the router uses to route the packet.

2.5.6 Musical data in Packet Switched Networks

2.5.6.1 Musical Control in Packet Switched Networks

There has been significant research and developmental efforts into integrating MIDI and computer networks in order to achieve musical control over packet switched networks[5]. Live MIDI broadcasts, remote control of MIDI-able musical instruments and distributed processing of music software applications are a few of the immediate benefits of such research. The quest to transport MIDI over packet switched networks has culminated in commercial products like NetMIDI and MIDIOverLAN+. By modern networking standards MIDI utilises very little bandwidth that would not exceed the serial data transmission rate of 31.250 kbps. This, coupled with cheap network technology, makes transporting MIDI over computer networks very attractive. There are numerous protocols that support MIDI over IP such as MIDI Wire Packetization Protocol (MWPP). MWPP uses RTP and by design is meant to transport real-time audio and video data. RTP handles low latency interactive applications as well as content delivery applications. RTP however does not provide any Quality of Service (QOS) to the application since it uses the connection-less UDP protocol to send datagrams. A payload format within RTP has been designed to transport MIDI data[22]. This design encodes all MIDI messages found in the MIDI 1.0 specification and provides services to recover from dropped packets lost to the network. The packet loss recovery service is necessary since MIDI cannot tolerate missing messages.

Another such protocol is Remote Music Control Protocol (RMCP) that also uses UDP as the transport mechanism for message delivery. RMCP supports

time scheduling by timestamping outgoing packets and providing scheduling services in the receiver. Since it is UDP-based message broadcasting can be deployed as an efficient means of information sharing without multiple transmissions.

A client/server architecture is used in the MIDI over IP networks. Clients would typically be considered as the MIDI OUT devices and servers as MIDI IN devices.

OSC can also contain MIDI messages as the payload in its messages. This is achieved through the non-standard 'm' data type. This MIDI type defines a 4 byte MIDI message encapsulating the MIDI data in an OSC message. The OSC address space is limitless and open ended for the designer to implement an address space of choice. This improves on the 16 channels available for conventional serial MIDI. There has however been some efforts into developing a standard address space for OSC MIDI messages by Fabian Ehrentraud called the 'SYN' namespace[10].

MIDI was designed to be used in local communications and not in remotely distributed computer networks. While efforts to transport MIDI over computer networks have seen relative success, the fundamental design of MIDI to be used in local networks impedes optimising this distributed remote transport. While OSC supports the encapsulation of MIDI messages in its MIDI data type, it also provides a framework for a new protocol definition to transport musical control data over computer networks in an efficient manner.

2.5.6.2 Packet switching real-time audio

> "There is great interest in transmitting continuous media on packet switched networks"[12]

Multimedia data such as audio are one of the strongest forms of communications. Therefore to transport this form of communication in computer networks creates a flexible distributed system for this core communication. High speed Internet has facilitated multi-channel high quality uncompressed PCM audio data transmission over packet switched networks and therefore allows for both interactive and non-interactive audio applications to be deployed.

Audio data has properties of being large and chunky in size, continuous, isochronous and has real-time requirements. These properties place great demand on packet switched networks that were meant to handle text data.

Non-interactive applications such as Internet radio broadcasting are less latency dependent as the sending transmission time has no relevance to the end listener or user. Therefore jitter buffers can accept relatively large latencies to filter jitter. Interactive applications on the other hand place requirements of having a minimal end to end delay. IP telephony can sacrifice quality to gain a reduction in latency. Distributed musical performance too requires minimal latency.

The open ended design space of OSC can facilitate transportation of multi-channel real-time uncompressed high quality audio data over packet switched

networks. This has been achieved in the Open Sound World environment[9]. The prototype developed in this dissertation work defines a new OSC data type, the float vector 'v' that is used to transport audio data as the payload in OSC messages.

Real Time Protocol is a real-time transport layer protocol capable of transmitting audio and video data. The Internet version of RTP uses UDP which is connectionless and does not guarantee delivery[5]. Lost UDP packets imply lost RTP packets and unordered UDP packets imply unordered RTP packets. The RTP header includes a 16-bit sequence number and a 32-bit timestamp. The application layer can therefore use the RTP header information to sequence real-time packets or perform other real-time processing. The RTP header also contains a marker bit (M) that can be used to indicate special packet handling. RTP does not provide any Quality of Service assurance. RTP has an associated control protocol called Real Time Control Protocol (RTCP) that codes synchronisation information. RTCP reduces its traffic as RTP session traffic increases.

2.6 Quality of Service (QOS)

Distributed Audio applications such as NMP and collaborative efforts are highly interactive systems placing real-time demands on the underlying networks. The requirements placed upon these networks are significantly higher than the traditional data applications such as database queries and Web browsing. These requirements are called QOS guarantees that the network must fulfill in order to service these systems.

Distributed musical performances essentially contain 2 classes of data. The first class is the multimedia data class consisting of high resolution isochronous data streams of audio data. Video data also falls into this class. The other class is control data for musical performance control such as the established MIDI and OSC when developed as a musical control protocol.

The OSI 7 layer reference model was developed in a data environment using low speed networks[16]. The QOS support defined in this model is limited and does not cater for multimedia data transfer over high speed networks and extends to distributed musical performances. The OSI upper layers are not QOS oriented.

The IP network layer protocol does not support any QOS guarantees over the Internet with its best effort packet delivery services. The Internet is therefore not well suited for real time data transmission.

TCP at the transport layer is connection-oriented but not suited for real time transmission because of the high latencies introduced when packets are lost and retransmission efforts are made by this layer.

UDP is connectionless and does not cater for error detection, packet scheduling or network jitter.

RTP however is used over Internet for real time data delivery and multimedia streaming. It's an application layer protocol using UDP as the underlying

transport layer. MWPP transports MIDI control data using RTP.

The primary concern of this work is the QOS metrics associated to the application layer since QOS in Distributed systems are fundamentally end-to-end[16]. OSC is defined as an application layer protocol and therefore applies in an application to application manner. It uses application layer defined QOS metrics.

Systems could be categorised into real-time and non-real time systems having different QOS metrics. Non real-time systems have QOS indicators such as bandwidth, error rate and packet loss and apply to the general data IP networks such as http, databases, etc.

An audio data delivery system is an example of a real-time system in which real-time data is delivered over high bandwidth networks. These types of networks have QOS indicators such as end-to-end delay, jitter and packet loss. Audio data delivery is a time dependent service requiring data to be delivered in a continuous, isochronous and real-time manner[2].

QOS Categories are groups of QOS that have common aspects of QOS. These are defined in Table 2.1.

QOS Category	QOS metrics in this category
Reliability	dropped packets, error rate
Timelines	end-to-end delay, latency, jitter
Volumes	mean throughput, peak throughput, bandwidth
Quality of Perception	Temporal Fidelity, scheduling

Table 2.1: QOS metrics per category

2.6.1 QOS metrics

The QOS metrics in the context of Distributed Audio applications apply in an end-to-end manner and relate to the application level.

2.6.1.1 Latency

Latency is defined in this context as the end to end delay from the time of generation of data to consumption of that data. It is defined in milliseconds. Acoustic latency is the time taken from the production of the sound waves within the instrument to the time to reach the listeners ear. In a full sized symphony orchestra the acoustic delay could reach up to 46ms from the conductor to a trumpet player sitting in the back row[20]. This is determined using the speed of sound and the distance between the two entities.

NMP latency consists of local latency plus network latency. The local latency consists of instrument interfaces, acoustics latencies, audio interface and audio codec latencies. Network latency consists of transmission delay, propagation delay and inter-networking device delay[20]. Transmission delay is the packet size divided by the communication speed. Propagation delay is the time for

the messages to travel to its destination and is dependent on physical distance. Device delay is the store-and-forward services found in routers and switches, the buffering and queuing in devices and packet processing.

Acceptable latencies in musical control data varies in the application, tempo and genre of music. Audio interface design guidelines stipulate under 10ms latencies as acceptable. Sound events to the human ear are indistinguishable from one another when they are closer than 25ms. Musical performers have managed to perform with acoustic latencies reaching 46 ms in a symphony orchestra. Up to 60ms can be tolerated for continuous sounding instruments with slow attack times.

2.6.1.2 Jitter

Jitter is defined as the variation in latency experienced by packets traveling from the same source to same destination. These packets may experience variations in queuing in store-and-forward devices and may traverse different paths on route to their destination. Jitter variation of 1 ms in streams of audio data are audibly observable and excessive jitter can compromise the integrity of musical and rhythmic content[37]. This 1ms jitter can affect the timbre of musical content[39]. Excessive jitter may cause 'clicks' and 'pops' in audio communications[7].

There are numerous methodologies used to measure jitter. Peak jitter is defined as the difference in the maximum and minimum latency measured over a period of time. Peak jitter is easy to determine and generates acceptable results except for spike data. Other methodologies include measuring the difference in arrival time of consecutive packets, difference in latency of consecutive packets, and difference in current and average latency.

2.6.1.3 Packet Loss

Packet loss is defined as a ratio of packets lost to packets transmitted between a source and its intended destination. In the musical control context this is an important metric if there is no mechanism in place for message retries. If a packet that contains a MIDI message 'note off' is lost then the note will continue indefinitely disrupting the musical content. Packets lost in audio data are noticeably audible through 'clicks' and 'pops' being observed in the data.

2.6.1.4 Scheduling

Scheduling is the preservation of the order of packets between the destination and the source. This metric is important in musical control as it can ruin a musical piece. In MIDI if the original order is 'note on' followed 0.5 seconds later by 'note off' but the receiver receives and processes 'note off' first (does nothing since the note is already off) thereafter receiving and processing 'note on', the note will continue indefinitely. This again disrupts the musical intention.

Audio data is also sensitive to scheduling and ordering mechanisms should be in place in the receiver.

2.6.1.5 Temporal Fidelity [voice quality metrics]

Temporal Fidelity is a subjective audio quality metric that measures the preservation of rhythmic integrity in musical data. It is affected by latency but more so by jitter as jitter can degrade the rhythmic integrity of musical data. Temporal fidelity is a system level property.

2.6.2 Control requirements in audio systems

The performance requirements placed on musical control data relate to the latency, jitter, scheduling or order preservation and packet loss. These metrics each have certain upper bounds to which they have to adhere in order for the musical system to be usable.

The latency upper bound is dependent upon the application, the tempo and the genre of the musical piece. Musical composition and performance requires a latency upper bound of 10ms[37]. Isolated sounds are indistinguishable from one another if their difference is less than 25ms. Acoustic latencies can vary up to 46ms in a symphony orchestra. Others argue that latencies for continuous sounding instruments can be slacked to 60ms[20].

Jitter affects temporal fidelity and can degrade rhythmic content. Peak jitter bounds for MIDI interfaces vary from 3.1ms to 10.5ms as surveyed by Brandt and Wright[37].

Scheduling and packet loss have an absolute upper bound requiring that packet order is preserved and no packets are lost in transmission. Slack on either of these metrics can destroy musical intent.

2.6.3 Audio requirements

Uncompressed 16-bit PCM audio sampled at 44100Hz requires a bandwidth of 44100 x 16 = 705,6kbits per second per channel with a guaranteed throughput to prevent audio dropouts in the receiver. Control mechanisms are usually deployed in the receiver to order packets and to buffer incoming audio packets sacrificing latency to eliminate jitter caused by the network. Packet switched networks were not designed for real-time services like audio and other multimedia applications and therefore do not make provision for these services on the network. TCP at the transport layer will retry sending lost packets with no regard for the latency introduced for the upper layers.

Audio data latency upper bound is dependent on the application context. For example, in Internet radio a substantial latency can be tolerated as the end user does not care about the latency offset. In real time networked musical performances latencies comparable to acoustic latencies have an effect and upper bound similar to that expressed in 2.6.2.

Jitter has a noticeable effect on audio streams and control mechanisms to remove this jitter are necessary. Such mechanisms are jitter buffers that buffer the incoming packets from the network and synchronously stream out the audio data to the application preventing audio dropouts from occurring. This

mechanism introduces an added latency into the system.

There are also control mechanisms available to reduce the effects of packet loss and non-ordered packets at the receiver[29]. These mechanisms are however beyond the scope of this work.

Temporal fidelity can be affected by introduced latency into an audio stream system. Latencies at 11ms between separate channels of the same source are heard as a chorus effect. Latencies of 69ms are observed as slapback echo and latencies beyond 103ms are considered annoying[19, 20].

2.6.4 QOS Control Mechanisms

2.6.4.1 OSC timestamping for Forward Synchronisation

The most important mechanisms to manage control of temporal data in OSC is the timestamp which has not been adopted readily into implementations of OSC[32]. OSC timetags specify the absolute time at which the messages are to take effect in the server or destination device. OSC however does not provide any sort of synchronisation techniques between client and server and relies on outside applications such as NTP to provide this.

OSC timestamps conform to the NTP standard[23] having a 64-bit number representing time since January 1st 1900 and has accuracy within 200 picoseconds. The first 32-bits indicate the number of seconds since January 1st 1900, and last 32-bits specify the fractional parts of a second. There is also an immediate timestamp that contains the hex sequence 0x00 0x00 0x00 0x01.

Through an analysis of the jitter variation generated from a network one can extract the peak jitter value. This peak jitter together with a reasonable safety margin can give an indication of backward synchronisation. This backward synchronisation is then used to determine the forward synchronisation which schedules outgoing OSC messages to be executed at a future dated time. This technique is called ad-hoc Backward/Forward synchronisation[32].

This technique can be used to eliminate network jitter introduced to the system and preserve scheduling by sacrificing some latency of the system. It therefore performs the functions of both a jitter buffer and a message ordering system.

2.6.4.2 Jitter buffers

A jitter buffer, also called a clawback buffer, is a buffering algorithm that remove the effects of jitter and drift that enables synchronised playback of audio data[18]. The source of this jitter or drift is mainly caused by network communications. For each audio channel used in synchronous playback a separate jitter buffer is needed. Jitter buffers trade an increase in latency for the removal of jitter for the destination temporal processes. The trade off between latency and jitter needs accurate measurement as sacrificing too little latency and having the buffer size be too small to accommodate the jitter range empties the buffer hindering synchronous playback of audio data. If the buffer size is too large an unnecessary latency is introduced into the system.

If a jitter buffer were to empty and requests from the synchronous playback destination were made to the buffer for data, then the jitter buffer supplies a stream of zero amplitude samples inserting silence into the destination audio stream . This occurs when the jitter buffer is not sized correctly.

2.7 Summary

OSC has attained critical mass with many music research project implementing OSC. VST plugins, the informal standard, have no implementation of OSC for networked communications.

There has been a single implementation of OSC transporting audio data in the OSW project. Transporting this critical resource in VST systems using OSC could open up many opportunities for developers and users of VST plugins as well as OSC researchers and implementers.

OSC Scheduling and timestamping implementations are few despite temporal semantics' importance in musical interfaces. An implementation and evaluation of OSC Scheduling and timestamping on real time audio data could contribute to the OSC body of knowledge and inspire further research.

Chapter 3

Prototype Development

3.1 Prototype Description

3.1.1 Problem Statement

An OSC VST Bridge system was proposed to be developed and implemented that created a communication bridge between VST systems and the OSC networked environment. The OscVstBridge prototype plugin contains OSC server and client modules amongst others that are capable of interfacing OSC data to an Ethernet network. External OSC devices are then able to communicate and exchange data with this system. The system spoofs[1] the VST host by supplying it with all the procedure calls required in the dll file of a VST plugin. The data traveling through the OscVstBridge plugin is available to the OSC domain granting external OSC devices direct access to the VST data. This data includes parameter control data, synchronisation and transport data as well as audio input and output data. The OscVstBridge plugin facilitates retrofitting of non-OSC VST hosts to the open OSC network domain. It also permits Distributed Audio processing by allowing VST hosts to be synchronised on different computers.

The proposed system has also provided a mapping of the VST control parameters from the VSTXML hierarchical namespace to the OSC namespace where the VST host supports VSTXML and the plugin provides its own VSTXML file.

Essential core features:

- a VST Functional Wrapper constituting a VST Host Spoofer,
- a data bridge between OSC and VST domains,
- OSC Data Type development including OSC control parameter, VST Time Info and developing an OSC audio type,

[1] Spoofing in this context is defined as the prototype plugin masquerading as a VST plugin to the VST host

- OSC environment development including OSC client and OSC server including VSTXML definition,
- Audio Receiver development implementing a JitterBuffer and OSC Scheduling to accommodate network jitter.

Research questions to be answered by OscVstBridge prototype:

- *Can we use Open Sound Control as an open protocol interface to implement networked communications between VST systems and other OSC-able environments thereby providing the capability for open distributed audio processing?*
- *Can we automatically build a rich OSC namespace using the VSTXML definition released in VST SDK v2.4?*

3.1.1.1 System Processes

1. *loading of VST plugin*
2. *configure OSC settings*
3. *configure VST-OSC routing*
4. *route VST audio from VST host to external OSC server*
5. *route VST control [parameter, sync, transport] from VST host to external OSC server*
6. *route OSC audio from external OSC client to VST host*
7. *route OSC control [parameter, sync, transport] from VST host to external OSC server*
8. *control VST parameters*
9. *construct OSC address space and VST parameter objects from VSTXML*
10. *display routed data in real time in plugin UI*

3.1.1.2 Constraints

The system is limited to VST plugins. Other plugin formats are not supported. VST plugins are limited to use with VST hosts only as they are needed to instantiate the plugin. The wrapping service provided by the plugin has a boundary within the plugin environment. The VST host environment is oblivious of the wrapping and bridging functionality. OSC is implemented over standard Ethernet technology and other communication technology are not used, such as USB or Firewire. We are then limited to Ethernet as the communication medium.

3.1.1.3 Rules

The VST plugin uses JvstWrapper that allows the development of VST plugins on the Java platform. Seventeen parameters have been created in the plugin to demonstrate the hierarchical namespace addressing facility available in OSC. Within the OscVstBridge plugin the OSC client[2] is able to provide data to the OSC server[3] internally.

3.1.1.4 Performance

The parameters within the plugin are controllable through the OSC domain. This control should have performance measures of a similar standard to that of perceived direct VST control. If this is not achievable then the performance measure should exceed industry accepted standards for audio control. Audio latency should also measure well against that of audio industry standards. The system was initially designed to accommodate 17 parameter controls, 2 audio in channels, 2 audio out channels as well as transport and synchronisation data. All performance data is benchmarked against this design.

3.1.1.5 User Requirements

There are no user expectations on using the system due it being a novel prototype in a research capacity. However upon using the system when implemented users would require the VST host to load the plugin successfully each time without bugs or software glitches. The configuring of the user OSC settings should have excellent usability with clear and precise flow in configuring these settings. The real time display data should be as informative as possible without cluttering the UI.

3.1.1.6 Resource Requirements

The two main resources are VST data and OSC data. VST data is acquired through the VST wrapper from the VST host. This data constitutes audio data and control data. Control data is subdivided into parameter, transport and synchronisation data. The application context in which the VST data is applied will give indication as to which data resources are relied on. If the application requirement is to get VST audio data into the OSC domain then the audio data becomes relied on by the VST wrapper. If synchronisation is the application requirement then the synchronisation and transport control data is relied on. The restriction around the VST data is applied to VST audio data in that the VST host will dictate the audio block size supplied to the VST wrapper. The host can also change this block size at any time after plugin instantiation and will inform the plugin thereof.

[2]sends OSC data over networked technology to OSC servers

[3]receives OSC data from OSC client and processes it through its address space matching and OSC methods

OSC data is acquired through the OSC server but the OSC server can remain dormant throughout the lifetime of the plugin if no external OSC client directs OSC data to this server. When data is received by an OSC server it is passed by the bridge to the corresponding VST node which sends it to the VST host. This can either be audio or control data. The OSC address pattern in the OSC data will be matched to a corresponding OSC address space in the OSC server and the relevant OSC method invoked. The OSC method will then act on the sound object which in the VST wrapper in the VST domain. If no matching occurs then the OSC message is discarded by the OSC server.

3.1.1.7 Functionality Requirements

The system must:

- **deploy a functional VST wrapper** that is capable of spoofing a VST host by supplying it with the procedure call of a VST plugin. This is needed to create the system in which the data bridging occurs between OSC and VST environments.
- **receive VST data** from the VST host. This could be control or audio data. This is a receiving interface to the VST environment.
- **transmit VST data** to the VST host. This could be control or audio data. This is a transmitting interface to the VST environment.
- **receive OSC data** from an external OSC client. This is a receiving interface to the OSC environment.
- **transmit OSC data** to an external OSC server. This is a transmitting interface to the OSC environment.
- **bridge between OSC and VST data.** This is the essential core function of the system that bridges data between the two interfaced environments.
- **control VST parameters.** This is done via the plugin UI as performed in conventional VST plugins.
- **construct OSC address space from VSTXML.** This is done in order to match OSC address patterns to OSC address spaces for correct OSC server operations.
- **handle different data types** through method calls. The OSC environment and VST system will provide different data types to the VST wrapper and the system must be capable of handling this.

3.1.2 Use Case Diagrams

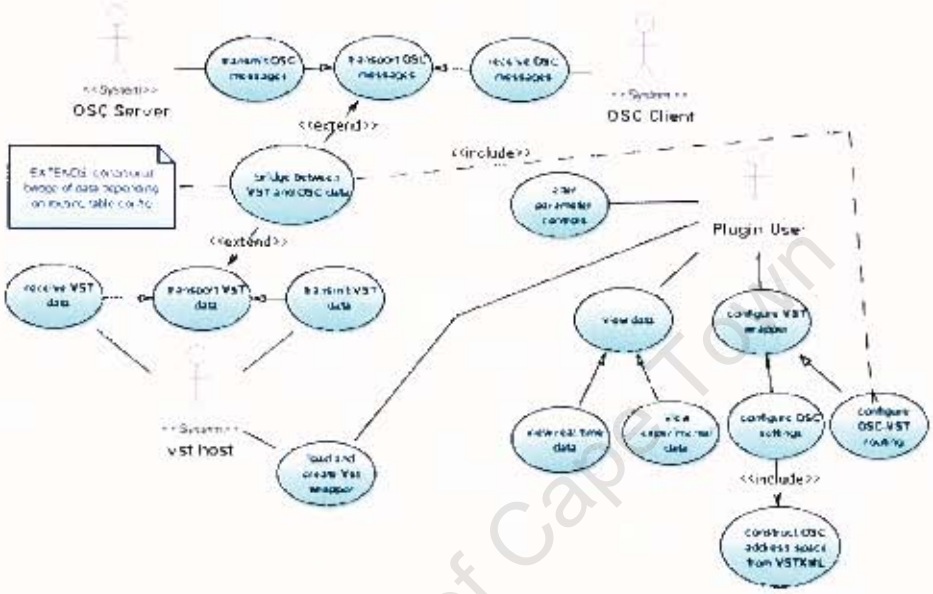

Figure 3.1: Prototype Use Case Diagram

Figure 3.1 illustrates the Use Cases [UC] used in the OscVstBridge prototype plugin. A decription of each UC is outlined:

3.1.2.1 Load and Create VST Wrapper

The UC is initiated by the plugin user loading the VST plugin from within the host environment. The host calls methods within the plugin dll in order to instantiate it.

3.1.2.2 Configure VST Wrapper

The default config settings are loaded upon plugin instantiation. The plugin user can change and apply configuration settings through the configuration UI. These configuration settings include OSC settings for the OSC server and the OSC client. The OSC server settings include the OSC server IP address and IP port. The OSC client settings include the OSC destination IP address and IP port as well as the OSC bundle delay.

Other settings include the JitterBuffer size used in receiving audio packets from the network.

The configuring of the OSC-VST routing is done for the parameters, audio and VST Time Info data types. This is achieved via enable/disable toggle buttons alongside the real-time information display.

The construction of the OSC address space and the parameter data types population is done through the VSTXML definition file supplied with the plugin. An XML parser is used to achieve this. All parameter types defined in the VSTXML file are mapped to parameter objects within the plugin.

3.1.2.3 Transport VST data

The UC provides the interface to the VST environment and consists of the "Transmit VST data" and "Receive VST data" UC's. It is also responsible for interfacing to the "Bridge" UC which lies at the heart of the prototype. The "Transmit VST data" UC receives data from the "Bridge" UC and transmits VST data to the VST host. The "Receive VST data" UC receives VST data from the VST host and sends it on to the "Bridge" UC.

3.1.2.4 Transport OSC Messages

The UC provides the interfacing to the OSC network domain by instantiating the OSC server and OSC client modules that are the responsibility of the "Receive OSC Messages" UC and the "Transmit OSC Messages" UC respectively. The "Transmit OSC Messages" UC received data from the "Bridge" UC packages it within OSC formatted packets, and transmits it using the OSC client into the network. The "Receive OSC Messages" UC receives OSC data from the network, decodes it using OSC protocol definition and matches the corresponding OSC methods in the OSC address space. All OSC methods eventually pass their data to the "Bridge" UC.

3.1.2.5 Bridge between VST and OSC data

The "Bridge" UC is responsible for routing between the VST and OSC environments. It resides at the core of the prototype. It has two branches. The first routes VST data to the OSC domain. The second routes the OSC traffic to the VST domain. The "Bridge" UC maintains a map of the objects it receives that can be routed. The map elements are configurable from the plugin UI.

3.1.2.6 View Data

The "View Data" UC allows the user to view and interact with the plugin UI. This UC contains the "View RealTime Data" UC and the "View Experimental Data" UC.

3.1.3 Workflow Diagrams

The workflow for the OscVstBridge prototype can be seen in Figure 3.2. It is divided into two parts:

3.1.3.1 Instantiation of the prototype

The constructor of the plugin loads the VST wrapper, configures the OSC settings, constructs the OSC address space and configures the OSC-VST routing. Thereafter a 'wait' state is assumed waiting for OSC messages or VST data to be received.

3.1.3.2 Bridging of Data

The bridging of data consists of two streams of data flow. The first stream flows from the OSC domain to the VST domain. Here OSC messages are received by the OSC server, transported through a thread in the Bridge class before transmitting it as VST data. The second stream flows from the VST domain to the OSC domain, in which VST data from the VST host is received by the prototype and bridged in its own thread to the OSC client which will transmit it as OSC data.

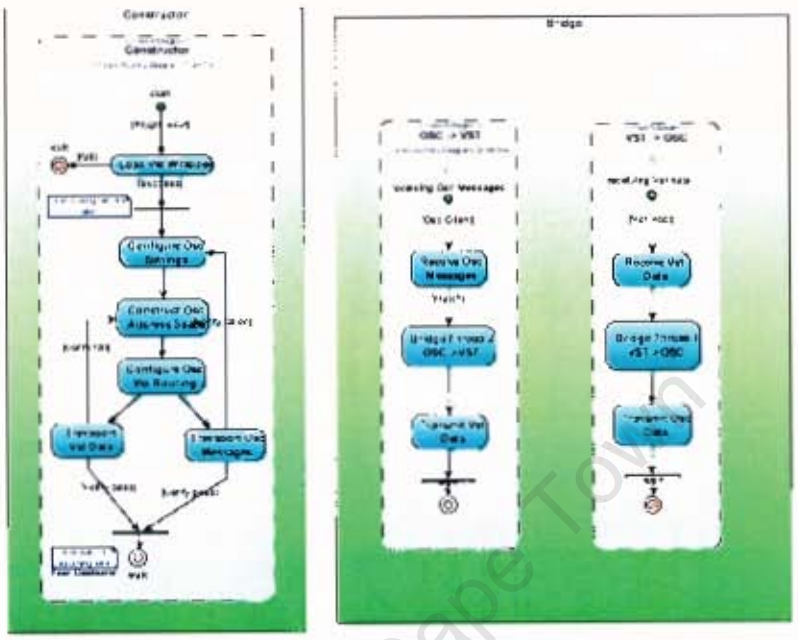

Figure 3.2: Prototype Workflow Diagram

3.1.4 Class Diagrams

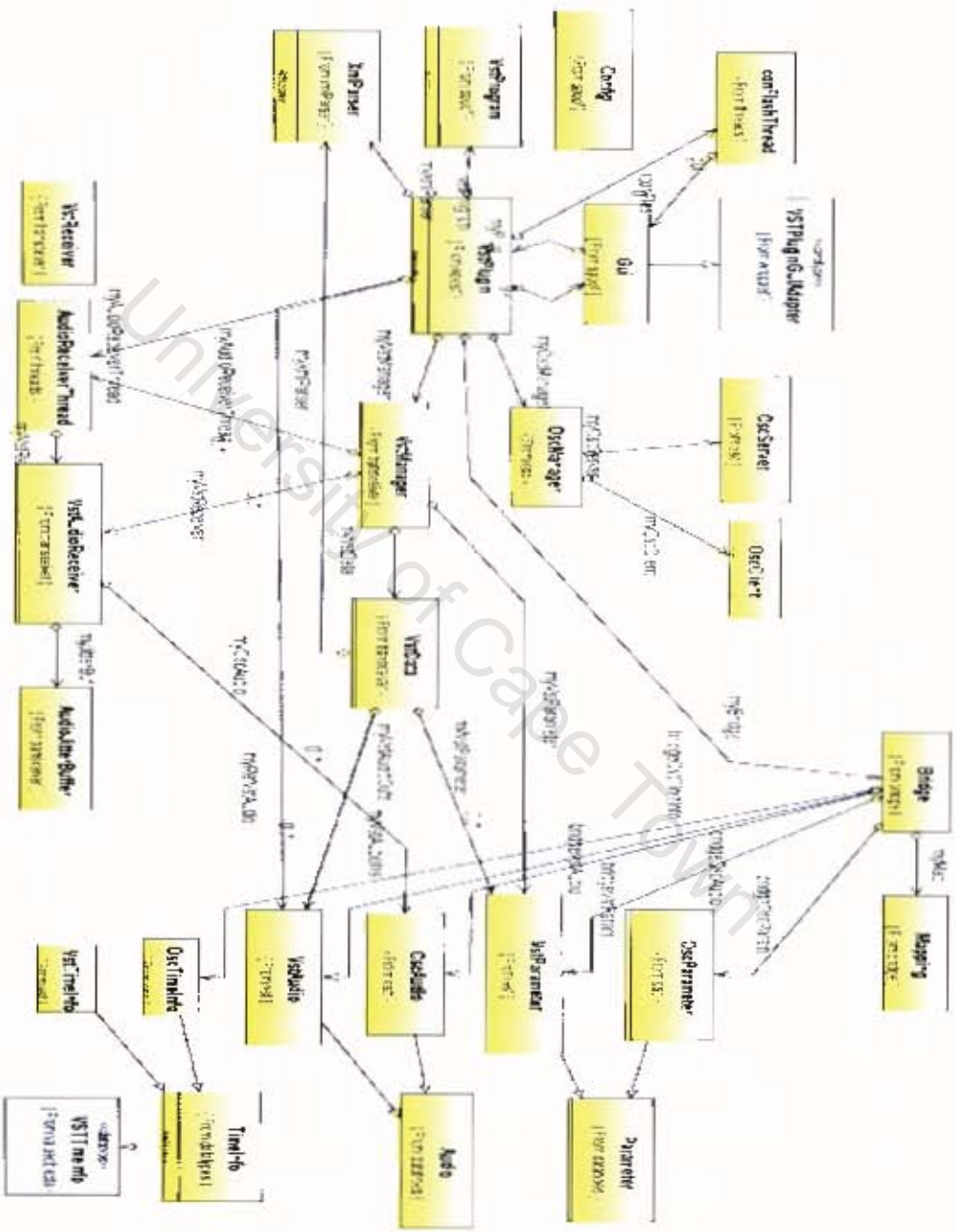

Figure 3.3: Prototype Class Diagram

The class diagram for the OscVstBridge prototype is found in Figure 3.3. Complete descriptions of the important and relevant classes can be found in Appendix A.

3.2 VST Host Spoofer

A VST plugin in Windows is a multi-threaded Dynamically Linked Library [dll] that contains methods that a VST host can call during runtime. These methods usually involve taking and returning streams of audio data between the VST plugin and the VST host. The VST plugin uses the same processor that the VST host uses. Audio processing done by the plugin prototype is achieved through the processReplacing() method. In this case audio processing is passing the audio streams to the OSC client for transmission onto the network, and receiving network traffic from the OSC server before returning it to the host. The method calls in initialising a plugin are the constructor AudioEffectX(), open(), setSampleRate() and setBlockSize().

In order to spoof a VST host a VST plugin framework needs to be utilised that provides methods as requested by the VST host.

3.2.1 jVSTwRapper review

jVSTwRapper is a Java VST wrapper that allows VST 2.4 plugins to be developed with the Java Programming Language. This is achieved by wrapping the native VST SDK C++ libraries into Java classes through the Java Native Interface (JNI). The native delegated calls from the VST host are translated by the JNI to Java method calls and vice versa. jVSTwRapper is a compiled binary on Windows with a dll extension and upon instantiation invokes a Java Virtual Machine (JVM) on which the plugin runs. This therefore allows platform independent development on either Windows, Mac or Linux that contain JVMs.

The same wrapping philosophy using the JNI to translate native calls to Java is adopted for the user interface of the plugin.

3.2.2 jVSTwRapper implementation

jVSTwRapper provides the VST framework necessary to spoof the VST host by providing the necessary translation from Java to native calls. The prototype is therefore developed on top of this framework in the Java Programming Language.

3.3 Open Sound Control Library - Netutil

Netutil is an Open Sound Control library developed in Java for Java. It support OSC servers and OSC clients, TCP and UDP transport, OSC bundles, and data types i (int32), b (blob), s (string) and f (float).

The OSC server groups transmitter and receiver. It can therefore not only receive OSC messages from the network but also respond to them without the need for an OSC client in the application. The OSC server requires an OSC listener to be added to it before it is able to receive messages. The OSC listener interface requires the messageReceived() method to be implemented which handles all incoming OSC messages.

The OSC client also allows for bidirectional communication. It not only can transmit OSC messages onto the Ethernet network but is also capable of receiving them by implementing the OSC listener interface with the messageReceived() method.

3.3.1 Netutil implementation

The implementation of the Netutil library within the prototype consists of a separately implemented OSC server and OSC client. The OSC server function is narrowed to receiving OSC messages from the network while the OSC client is limited to creating and transmitting OSC packets onto the network.

The standard data types however are not sufficient for transporting audio data as an array of floats as supplied by the VST host. Therefore a new data type needs to be developed and implemented within the prototype. This data type is the 'v' data type for float vector arrays and is covered in subsequent sections.

3.4 OSC Data type design

There are various data types that are associated with a VST plugin. The most important are VST parameter or control types, audio types, timing and synchronisation types.

The parameter types are used to influence the internals of the plugin processes by providing control data to it. The parameter data is also available to the user via the UI of the plugin which allows for audio manipulation.

The audio data types are essential as the plugin is an audio processing unit. The data contained within the audio type are the inputs and outputs of the plugin system.

Timing and synchronisation information is available through a VSTTimeInfo object in the plugin from which this type of information can be sourced from the VST host environment. The information contained within the VSTTimeInfo object includes sample position, sample rate, system time, musical position, tempo, bar start, cycle start and end, time signature, SMPTE and midi clock.

Each of the above types are required to be designed and implemented within the plugin when interfacing data to the OSC environment. Parameter, audio and TimeInfo data types are defined upon which VST and OSC versions are extended.

3.4.1 OSC Parameter /Control data type

When a parameter is defined in a plugin it contains the following variables:

index - provides an index to the parameter for referencing from the host and UI.

normValue - is the normalised/ actual value contained within the plugin. normValue's range is -1.0 to 1.0 inclusive.

pName - provides the name of the parameter to the host as well as the UI for displaying to the user.

dispValue - is the displayed value to the user in the UI and in the host. This usually involves calculating this value from the normValue internal.

label - is the label of the parameter or measurement unit.

The rendering of the parameter UI to the user will closely resemble Figure 3.17.

A Parameter class was designed and implemented within the plugin. The class contains the variables index, normValue, pName, dispValue and label. There are get() and set() methods for each variable. The VstParameter and OscParameter classes inherit from the Parameter class. The VstParameter is used to store VST parameter data while OscParameter used to store OSC parameter data.

3.4.2 OSC VST Time Info data type

The VST SDK2.4 defines the VSTTimeInfo class that contains information regarding timing and synchronisation about the VST host. It defines the following variables:

samplePos - current position in audio samples

sampleRate - current sample rate in Hz

nanoSeconds - System time in nanoseconds

ppqPos - Musical Position in Quarter Note

tempo - current tempo in Beats Per Minute (BPM)

barStartPos - last Bar Start Position in Quarter Note

cycleStartPos - Cycle start (left locator) in Quarter Note

cycleEndPos - Cycle End (right locator) in Quarter Note

timeSigNumerator - Time Signature Numerator

timeSigDenominator - Time Signature Denominator

smpteOffset - SMPTE offset (in SMPTE subframes)

smpteFrameRate

samplesToNextClock - Midi Clock Resolution (24 per Quarter Note)

flags

A new child class called TimeInfo extending VSTTimeInfo is defined with the same members as those defined above. There are also get() and set() methods for each member. The VstTimeInfo and OscTimeInfo classes are derived from the TimeInfo class. Each applies to the VST and OSC domains respectively.

While the framework for the bridging of timing and synchronisation data is provided by the prototype there is no application defined. Future work might include such an application.

3.4.3 OSC Audio data type

3.4.3.1 The need for a new type

The standard set of type tags defined by OSC include integers(i), floats(f), strings(s) and blobs(b) [chunks of arbitrary binary data][40]. The extended non-standard type tags include 64-bit integers (h), OSC-timetag (t), 64-bit double (d), alternate OSC String (S), 32-bit RGBA (r), 4-byte midi message (m), True/False (T/F), Nil (N), and Infinitum (I).

While this set of standard and non standard type tags are comprehensive there is no provision made for vectors of any types. The audio data chunks supplied by the VST host are vectors of floats and so the need to develop a new type has surfaced. There has been talk from the OSC developers mailing list of new vector types. One of such types is the float vector type (v) consisting of an array of 32-bit floats.

3.4.3.2 float vector [v-type] implementation

The Netutil library supports the integer (i), blob (b), String (s) and float (f) types. The new float vector (v) type required development before integrating into the prototype. This involved making changes to the OSCPacketCodec class in the Netutil library.

The OSCPacketCodec is responsible for encoding OSC messages into a bytebuffer for transmission onto the Ethernet network. Similarly it is also responsible for decoding bytebuffers into OSC messages.

Atom classes are defined for each of the data types above that implement an abstract class Atom. Therefore a new Atom class called floatVectorAtom was created, implementing the methods decodeAtom(), encodeAtom(), getAtomSize() and getTypeTag().

OSCPacketCodec methods

The encodeAtom() method has the float vector as input arguments. These arguments are passed to the OSC message upon instantiation of the OSC message. The method takes this argument array and packages it into a bytebuffer that is ready for transmission of the data onto the network.

The encodeAtom() method code is seen below:

```
public void encodeAtom( Object o, ByteBuffer tb, ByteBuffer db ) throws IOException
final float[] floatVector = (float[])o;
tb.put( (byte) 0x76 );                    // 'v'
final FloatBuffer fb;
fb= db.asFloatBuffer();
fb.put(floatVector);
db.position( db.position() + (floatVector.length<<2)) ;
}
```

The decodeAtom() method has the bytebuffer argument that contains the networked OSC data. The method extracts the network received OSC message argument data and places it into a float vector. When requests are done from the OSC message object for its arguments [when the OSC message type is of float vector] the float vector which contains the audio data chunks is returned.

The decodeAtom() method code is seen below:

```
public Object decodeAtom( byte typeTag, ByteBuffer b ) throws IOException{
final FloatBuffer fb = b.asFloatBuffer();
final int fvLength = fb.capacity();
final float[] fv =new float[fvLength];
fb.get( fv );
b.position(b.position()+(fv.length<<2));
return fv;
}
```

The getTypeTag() method simply returns the ASCII character 'v' - the type tag used for the float vector. The getAtomSize() method returns the length of the atom.

3.5 Data Flow

In order to bridge between VST and OSC data must flow from one domain to the other and vice-versa. Data flow is therefore a critical element in the prototype and is implemented by the use of the Observer and Observable classes. A class extending the Observable class can notify registered listeners (other classes implementing the Observer interface) that a change has occurred in a member of the extended Observable class. This is done with the following code:

```
setChanged();
notifyObservers(localMember);
```

The above code notifies the registered Observers and invokes their update() methods which is defined in the Observer interface and must be implemented. The update method() receives as arguments a reference to the localMember from the Observable class.

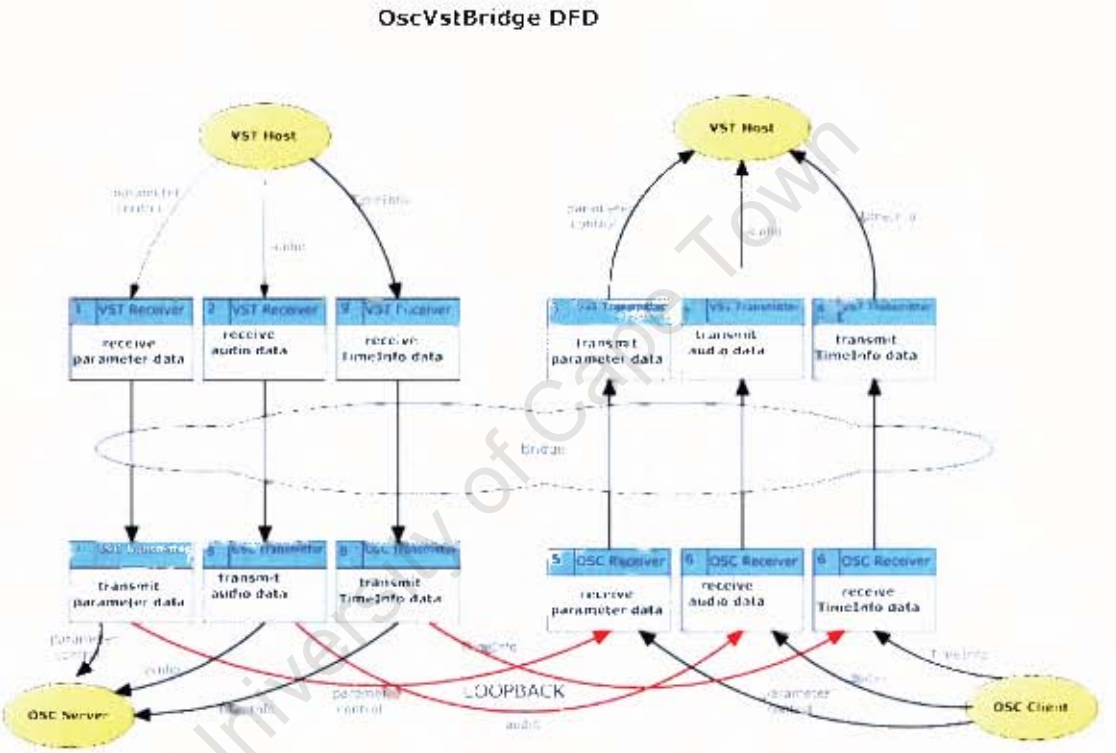

Figure 3.4: OscVstBridge Data Flow Diagram

The flow of data is divided into two main categories:

- VST to OSC
- OSC to VST

Within each of these categories there are subcategories carrying each of the data types implemented in the prototype. These are the parameter, audio and TimeInfo types.

Therefore the category structure is:

- VST to OSC as seen in Figure 3.5

- Parameter
- Audio

  TimeInfo

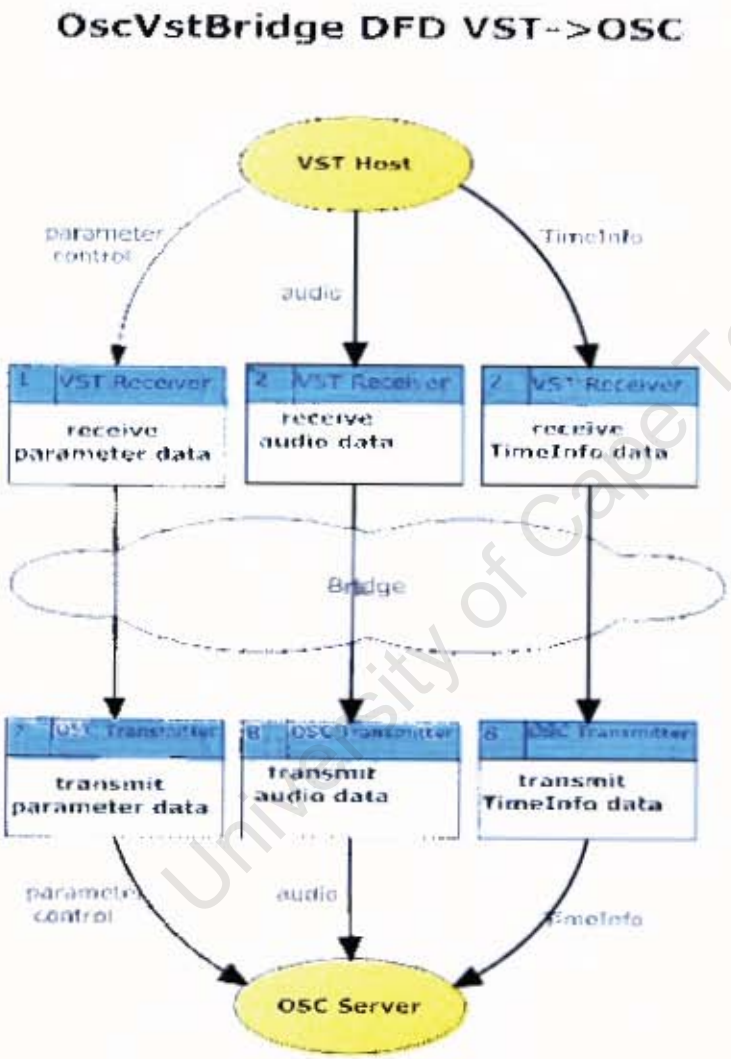

Figure 3.5: OscVstBridge Data Flow: VST to OSC

- OSC to VST as seen in Figure 3.6

  Parameter

  Audio

– TimeInfo

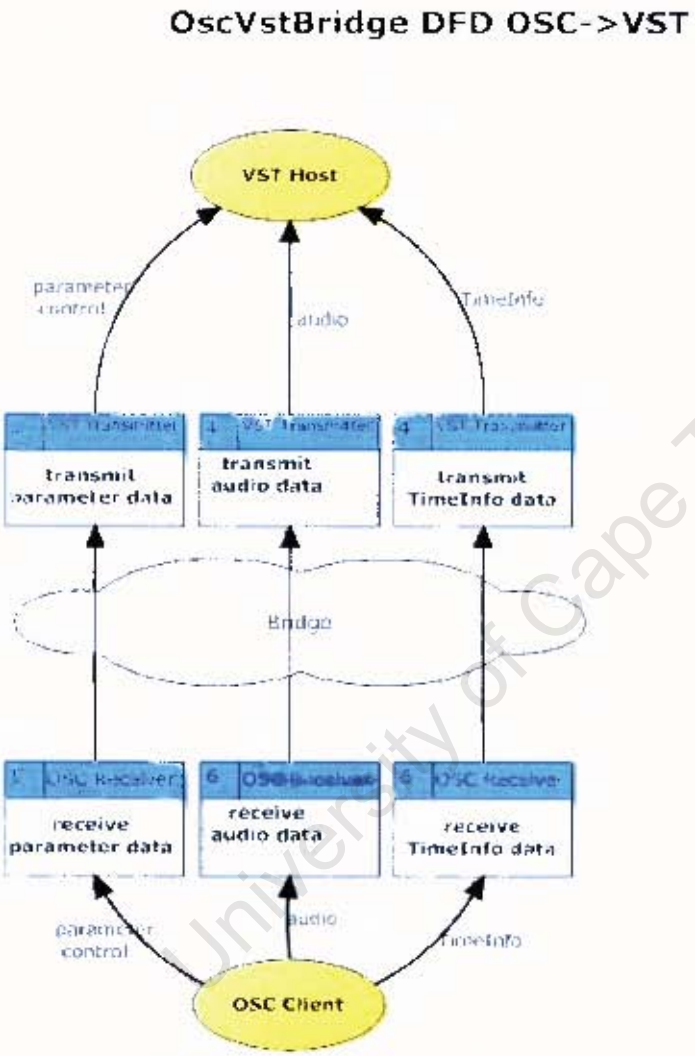

Figure 3.6: OscVstBridge Data Flow: OSC to VST

3.5.1 The Bridge Class

Every data flow process travels through the Bridge class. The purpose of the Bridge class is to selectively determine which processes should be further routed to their intended destination. This selective routing is achieved with the Mapping object which is instantiated by the Bridge class.

The Bridge class extends Observable and implements the Observer interface.

If objects are received into this class from an Observable class, the class checks the object in the Mapping object to determine if it may be further routed. If it can, then the Bridge class in turn notifies its Observers and passes the object on.

3.5.2 Parameter Data Flow

The Parameter data flow is illustrated below in Figure 3.7.

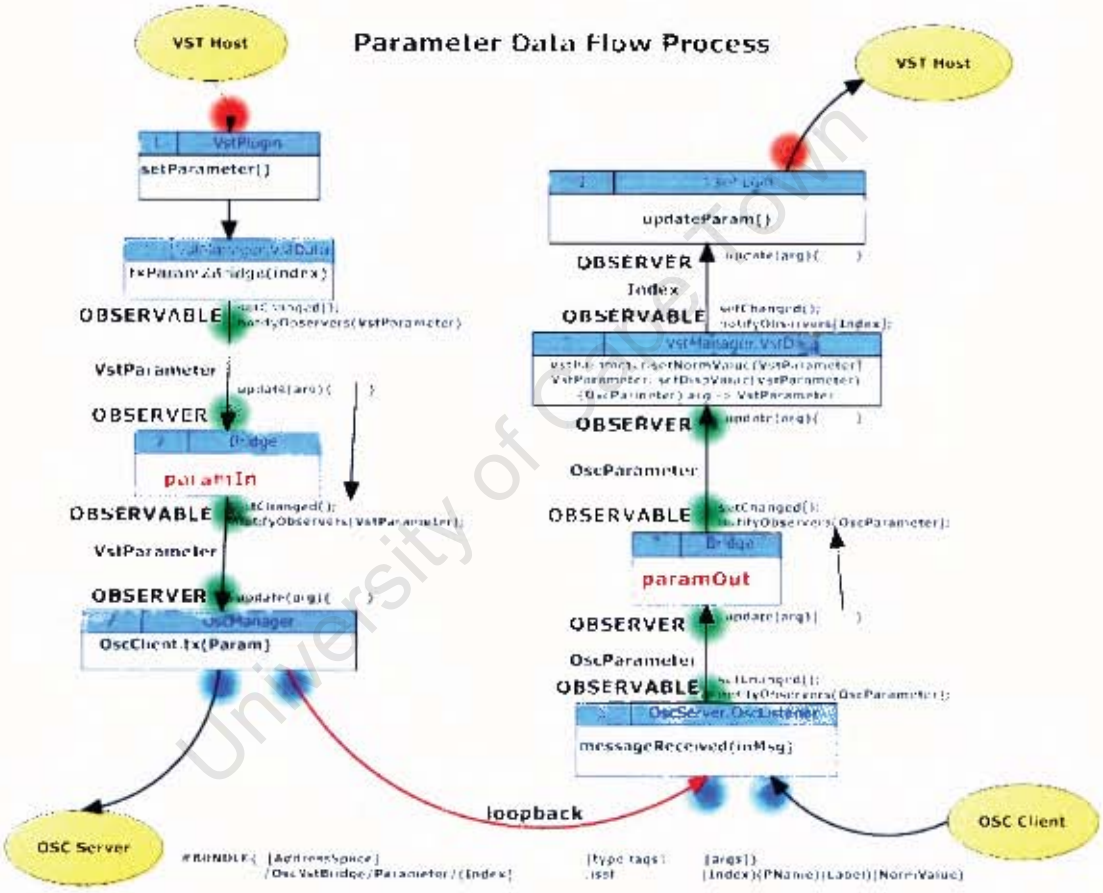

Figure 3.7: Parameter Data Flow Diagram

3.5.2.1 VST to OSC Parameter Data Flow

The Parameter data flow process from the VST to the OSC environment is initiated by the VST host calling the setParameter() method in the plugin. This method in turn invokes the txParam2Bridge() method but not before updating the plugin parameters contained within VstData object. The txParam2Bridge()

method notifies the registered Observers of this class and passes the VstParameter object reference to the Observer. In this case the registered Observer is the Bridge class. The update() method in the Bridge class receives the VstParameter object and if the mapping object allows for this VstParameter to be routed then the Bridge class notifies its registered Observers in turn and passes the VstParameter object as an argument. The relevant registered Observer in this case is the OscManager class. When the VstParameter object is received in the OscManager class the update() method invokes a method in the OscClient called tx() and passes the VstParameter object. The tx() method takes the VstParameter and constructs a new OSCBundle object from the Netutil library. An OSCMessage object is constructed from the VstParameter object received. The OSC message is then added into the OSCBundle object.

The OSC message constructed has the address space of

```
/OscVstBridge/Parameter/(Parameter Index)
```

The message contains four arguments of the following types : INTEGER (I), STRING (S), STRING (S), FLOAT (F). These arguments are:

- PARAMETER INDEX
- PARAMETER NAME
- PARAMETER LABEL
- PARAMETER NORMALISED VALUE

It should be evident that these arguments are similar to the members defined in the Parameter class.

Therefore a complete OSC message contained within an OSCBundle from this process would resemble:

[Address Space]	[type tags]	[arguments]
/OscVstBridge/Parameter/(Index)	,issf	(Index)(PName)(Label)(NormValue)

The OSC client determines the destination address and port for OSC messages. This could be an external OSC server or the OSC server module inside the prototype. The internal OSC server is addressed through the loopback network facility.

3.5.2.2 OSC to VST Parameter Data Flow

If the OSC client directs it packets to the internal OSC server via the loopback interface, the OSC Listener from the Netutil library invokes the messageReceived() method. The messageReceived() method looks at the OSC address space of the incoming message and executes its block of code related to that type. In this case it is a Parameter type and an OscParameter object is created

using the arguments from the received OSC message. The OSC server then notifies all registered Observers of this class and passes the OscParameter object to them.

The registered Observer is again the Bridge class whose update() method is invoked. The update() method receives the OscParameter object and if the mapping object allows for this OscParameter to be routed then the Bridge class notifies its registered Observers in turn and passes the OscParameter object as an argument. The relevant registered Observer in this case is the VstData class. When the update() method in VstData is invoked it modifies the plugin parameter with a matching index. These modifications include setting the normalised value and the displayed value of the parameter in question. The VstData class notifies its registered Observer and passes the index of the parameter to it. The relevant Observer is the VstPlugin class itself and its update() method calls the updateParam() method as well as the setParameterAutomated() method of the VstPlugin. This updateParam() method updates the plugin UI displaying this updated parameter information to the user. The setParameterAutomated() notifies the VST host of the parameter changes received from the network.

3.5.3 TimeInfo Data Flow

The TimeInfo data flow is illustrated below in Figure 3.8.

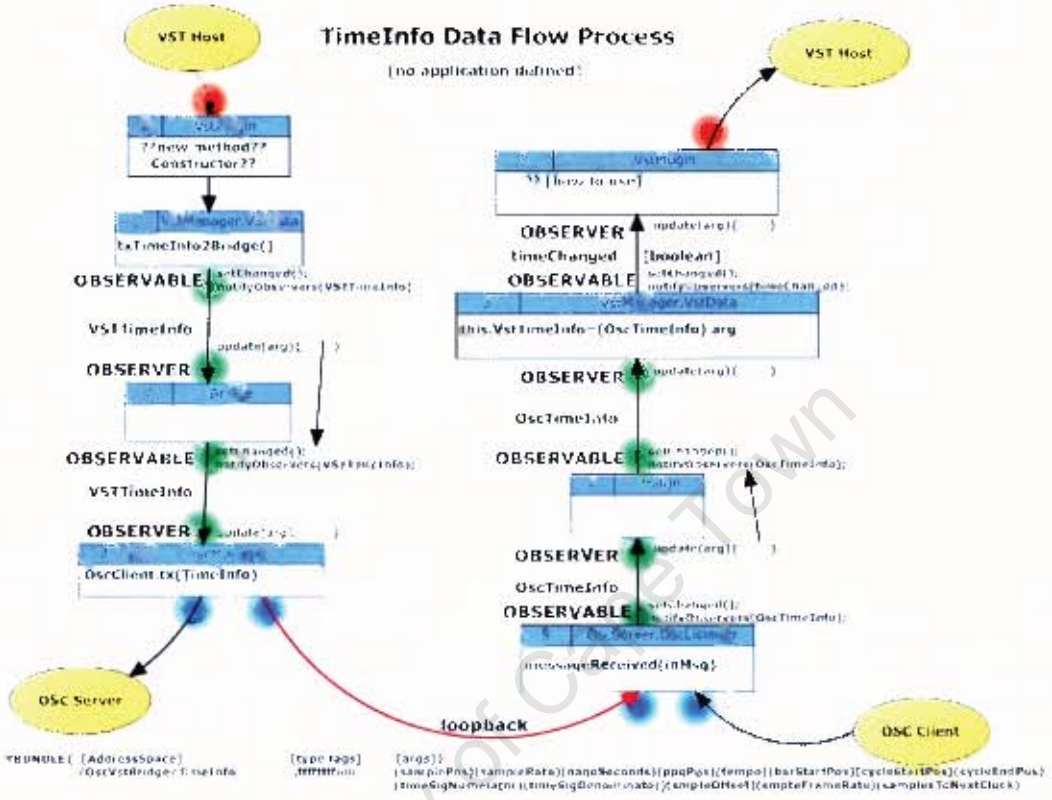

Figure 3.8: TimeInfo Data Flow Diagram

3.5.3.1 VST to OSC TimeInfo Data Flow

The TimeInfo data type provides the framework for application development. However within the prototype there is no application defined to use this type. If an application is to be defined then the implementing code is to be done within its own defined method in the VstPlugin class. This method would be required to call the method txTimeInfo2Bridge() that sets members within the VstTimeInfo object. The txTimeInfo2Bridge() method within the VstData class notifies the Observers of this class and passes the VstTimeInfo object. The registered Observers are the Bridge class that receives this VstTimeInfo object in the update() method. If the VstTimeInfo object is defined as routable by the Mapping class then the Bridge class in turn will notify its Observers and pass the VstTimeInfo object. The registered Observer is the OscManager class that invokes the tx() method in OscClient class when the VstTimeInfo object is received. The argument to this method is the VstTimeInfo object. The tx() method takes the VstTimeInfo object and constructs a new OSCBundle object from the Netutil library. An OSCMessage object is constructed from the VstTimeInfo object received. The OSC message is then added into the

OSCBundle object.

The OSC message constructed has the address space of

```
/OscVstBridge/TimeInfo
```

The message contains thirteen arguments of the following types : FLOAT (F), FLOAT (F), FLOAT (F), FLOAT (F), FLOAT (F), FLOAT (F), FLOAT (F), FLOAT (F), INTEGER (I), INTEGER (I), INTEGER (I), INTEGER (I), INTEGER(I). These arguments are:

- Sample Position
- Sample Rate
- nanoSeconds
- ppq Position
- Tempo
- Bar start position
- Cycle start position
- Cycle end position
- Time signature Numerator
- Time signature Denominator
- smpte offset
- smpte frame rate
- Samples to next clock

It should be evident that these arguments are exactly the members defined in the TimeInfo class.

Therefore a complete OSC message contained within an OSCBundle from this process would resemble:

```
[Address Space]            [type tags]       [arguments]
/OscVstBridge/TimeInfo     ,fffffffiiiii     args
args = (samplePos)(sampleRate)(nanoSeconds)(ppqPos)(tempo)
(barStartPos)(cycleStartPos)(cycleEndPos)(timeSigNumerator)
(timeSigDenominator)(smpteOffset)(smpteFrameRate)(samplesToNextClock)
```

The OSC client determines the destination address and port for OSC messages. This could be an external OSC server or the OSC server module inside the prototype. The internal OSC server is addressed through the loopback network facility.

3.5.3.2 OSC to VST TimeInfo Data Flow

If the OSC client directs its packets to the internal OSC server via the loopback interface, the OSC Listener from the Netutil library invokes the messageReceived() method. The messageReceived() method looks at the OSC address space of the incoming message and executes its block of code related to that type. In this case it is a TimeInfo type and an OscTimeInfo object is created using the arguments from the received OSC message. The OSC server then notifies all registered Observers of this class and passes the OscTimeInfo object to it.

The registered Observer is again the Bridge class whose update() method is invoked. The update() method receives the OscTimeInfo object and if the Mapping object allows for this OscTimeInfo object to be routed then the Bridge class notifies its registered Observers in turn and passes the OscTimeInfo object as an argument. The relevant registered Observer in this case is the VstData class. When the update() method in VstData is invoked it modifies the TimeInfo object within VstData. These modifications are application dependent. The VstData class notifies its registered Observer and passes a boolean reference to it. The relevant Observer is the VstPlugin class itself and its update() method should call the method of the intended application.

3.5.4 Audio Data Flow

The Audio data flow is illustrated below in Figure 3.9.

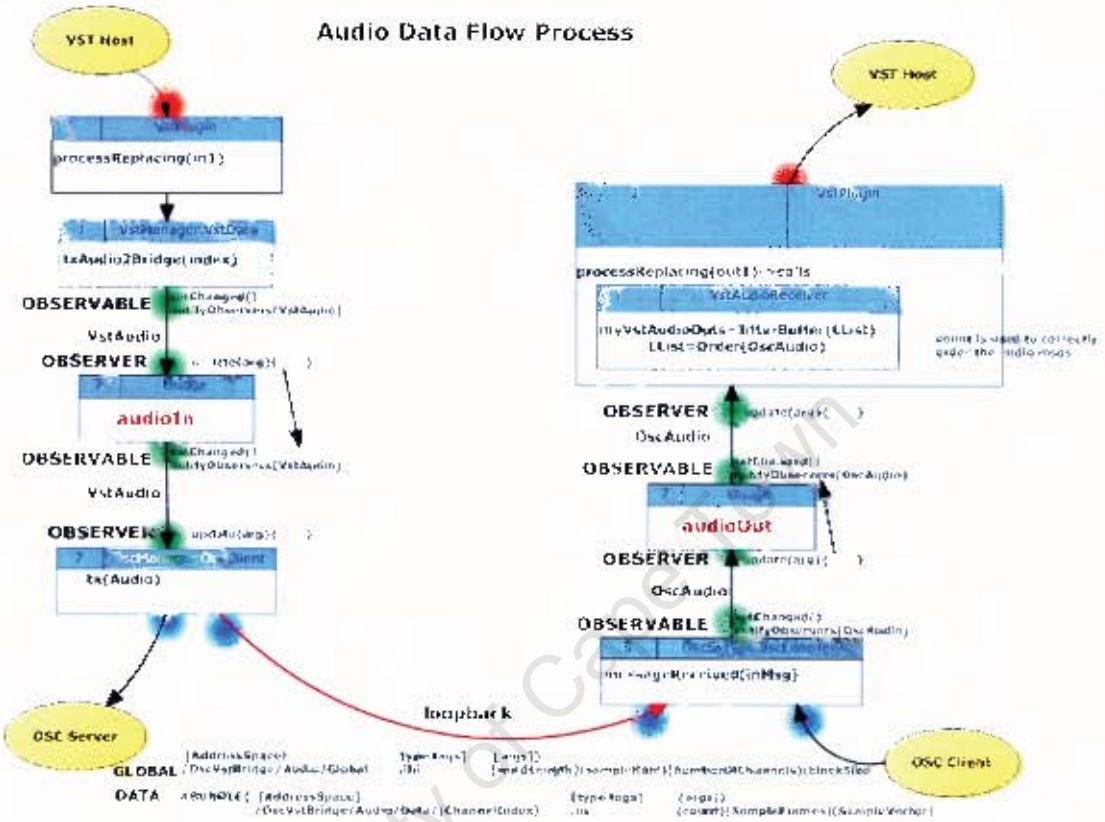

Figure 3.9: Audio Data Flow Diagram

3.5.4.1 VST to OSC Audio Data Flow

The Audio data flow process is the most processor intensive within the prototype. The VST host initiates the audio data flow with several calls per second to the processReplacing() method in VstPlugin. This method in turn invokes the txAudio2Bridge() method but not before updating the plugin audio objects contained within VstData object. The txAudio2Bridge() method notifies the registered Observers of this class and passes the VstAudio object reference to the Observer. In this case the registered Observer is the Bridge class. The update() method in the Bridge class receives the VstAudio object and if the Mapping object allows for this VstAudio object to be routed then the Bridge class notifies its registered Observers in turn and passes the VstAudio object as an argument. The relevant registered Observer in this case is the OscManager class. When the VstAudio object is received in the OscManager class the update() method invokes a method in the OscClient called tx() and passes the VstAudio object. The tx() method takes the VstAudio and constructs a new OSCBundle object from the Netutil library. An OSC message is constructed

from the VstAudio object received. The OSC message is then added into the OSCBundle object.

There are two type of OSC audio messages. The first is one that transmits **global** audio parameters to the destination OSC server.

The OSC message for **global** audio parameters constructed has the address space of

```
/OscVstBridge/Audio/Global
```

The message contains four arguments of the following types : INTEGER (I), FLOAT (F), INTEGER (I), INTEGER (I). These arguments are:

- WORDLENGTH - the bit depth of the samples
- SAMPLERATE - the number of samples per second
- NUMBER OF CHANNELS - the number of channels in the audio system
- BLOCKSIZE - the number of samples per block defined by VST host

A complete OSC message contained within an OSCBundle for the global audio parameters would resemble:

```
[Address Space]                    [type tags]   [arguments]
/OscVstBridge/Audio/Global         ,ifii         args
args = (wordLength)(sampleRate)(numberOfChannels)(blockSize)
```

The other type of OSC audio messages are the messages that contain the actual audio data.

The OSC messages for audio **data** constructed has the address space of

```
/OscVstBridge/Audio/Data/(Channel Index)
```

The message contains four arguments of the following types : INTEGER (I), INTEGER (I), FLOAT-VECTOR (V). These arguments are:

- COUNT - count of the OSC messages created for ordering OSC Audio in the receiver
- SAMPLEFRAMES - the number of samples per block processes
- SAMPLEVECTOR - the audio data as a vector

The arguments in the **global** and the **data** messages arguments are derived from the members in the Audio class.

Therefore a complete OSC message contained within an OSCBundle from this process would resemble:

```
[Address Space]                              [type tags]   [arguments]
/OscVstBridge/Audio/Data/(channelIndex)      ,iiv          args
args =  (count)(SampleFrames)(SampleVector)
```

The OSC client determines the destination address and port for OSC messages. This could be an external OSC server or the OSC server module inside the prototype. The internal OSC server is addressed through the loopback network facility.

3.5.4.2 OSC to VST Audio Data Flow

If the OSC client directs it packets to the internal OSC server via the loopback interface, the OSC Listener from the Netutil library invokes the messageReceived() method. The messageReceived() method looks at the OSC address space of the incoming message and executes its block of code related to that type. In this case it is an Audio type and an OscAudio object is created using the arguments from the received OSC message. The OSC server then notifies all registered Observers of this class and passes the OscAudio object to it.

The registered Observer is again the Bridge class whose update() method is invoked. The update() method receives the OscAudio object and if the Mapping object allows for this OscAudio object to be routed then the Bridge class notifies its registered Observers in turn and passes the OscAudio object as an argument. The relevant registered Observer in this case is the VstData class. When the update() method in VstData is invoked it modifies the plugin audio objects with a matching channel index. These modifications include setting the sample vector of the audio channel in question. The VstData class notifies its registered Observer and passes the OscAudio object to it. The relevant Observer is the VstPlugin class itself and the putAndProcess() method of the VstReceiver class is called. The plugin UI elements also are updated from the update() method. The putAndProcess() method takes the audio vector received from the network and orders the packets according to the count variable. The VstReceiver also serves as a buffer to compensate for jitter on the network.

The processReplacing() method is the receiver of isochronous audio from the VstReceiver class. The processReplacing() method requests data from the VstReceiver class which is received from the Ethernet network as OSC Audio messages.

3.5.5 Loopback data flows

Data flow can occur via the internal loopback interface for the different data types. For audio the audio data is received from the VST host and the data flow is initiated. The audio data is encapsulated within VstAudio objects passing through the Bridge to the OSC Client. The OSC client then transmits the audio data onto the internal loopback Ethernet interface. The OSC server within the prototype then receives this OSC audio data as OSC messages, decodes it and

passes it via the bridge to the VstReceiver. The VstReceiver then orders the packets and compensates for network jitter. The VstPlugin then pulls data from the VstReceiver and supplies it back to the VST host environment.

The same concept applies for the other two data types using the internal loopback interface.

3.6 Audio Receiver

Within the audio data flow process audio data is received from the Ethernet network from either the internal OSC client or some other external OSC client. The network introduces jitter due to the undetermined routing of packets over the network. Within the OSI framework the transport layer used for OSC packets within the prototype is UDP which provides no mechanism for the ordering of packets at the receiver as they were transmitted. It is therefore the responsibility of the prototype at the application layer to provide the functionality of ordering packets and providing a buffer to compensate for network jitter experienced.

The Audio Receiver class provides such mechanisms by firstly ordering the OSC audio objects based on the count arguments within the OSC audio object, and then provides a buffer that compensates for the jitter on the network.

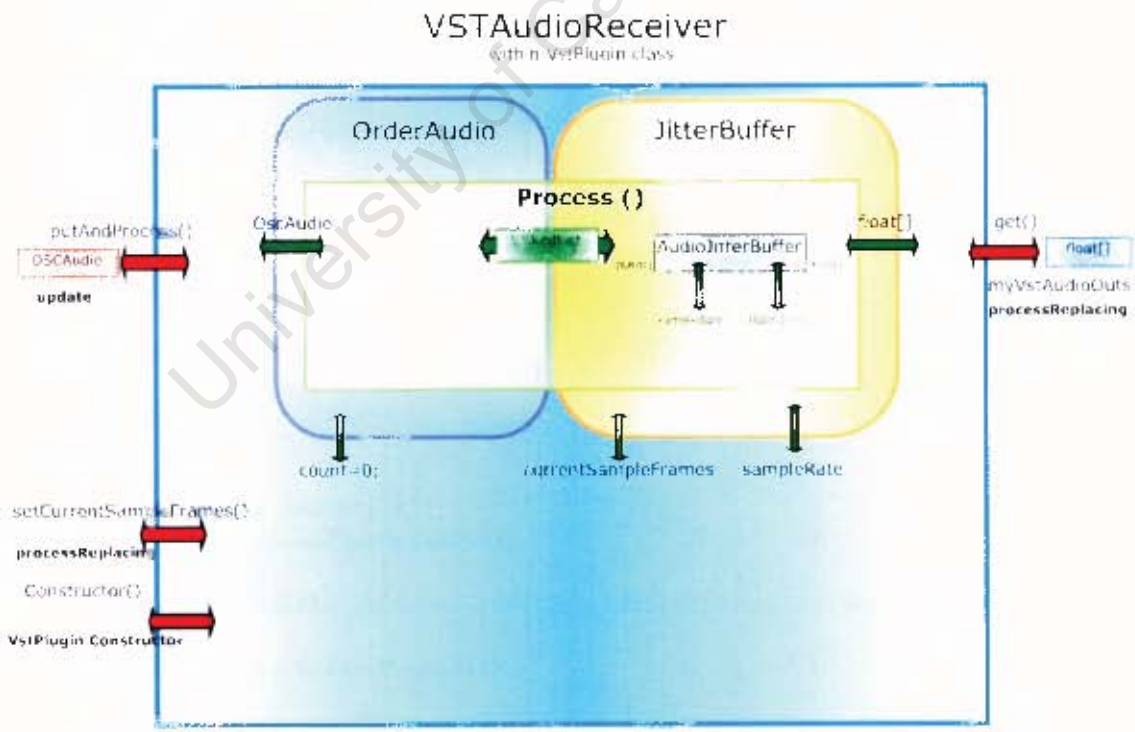

Figure 3.10: VstAudioReceiver diagram

In Figure 3.10 the OscAudio objects are received in the update() method of VstPlugin as explained in section 3.9. The OscAudio object is then passed as an argument in the putAndProcess() method to VstAudioReceiver. This method in turn calls the OrderAudio() method followed by the JitterBuffer() method.

3.6.1 OrderAudio

The orderAudio algorithm is contained within the VstAudioReceiver class. OrderAudio uses the count member to order the OscAudio objects received and places all the OscAudio object within a linked list.

The algorithm structure is:

- if linked list is empty, then add OscAudio object to linked list at first index,
- if linked list is not empty then compare the count value of the incoming OscAudio object to that of the maximum count value of all the OscAudio objects presently in the linked list,
- if received OscAudio object count value is greater than max of all OscAudio objects in linked list then add received OscAudio object to end of linked list,
- else find the index point in the linked list where received OscAudio object should be inserted based on its count value and insert it in that location.

The code algorithm for OrderAudio is:

```
public LinkedList orderAudio(int count,OscAudio myOscAudio){
boolean added;
int llsize;
if(!this.myList.isEmpty()){
    llsize=this.myList.size();
}else{
    llsize =0;
}
added=false;
if(this.myList.isEmpty()){
    this.myList.add(myOscAudio);
    count++;
}else if(myOscAudio.getCount()>=((OscAudio)this.myList.get(this.myList.size()-1))
                                                                    .getCount()){
    this.myList.add(myOscAudio);
    count++;
}else{
    for(int j=0;j<llsize;j++){
        if(!added){
```

```
                if(myOscAudio.getCount()>((OscAudio)this.myList.get(j)).getCount()&&
                 myOscAudio.getCount()<((OscAudio)this.myList.get(j+1)).getCount()){
                    this.myList.add(j+1, myOscAudio);
                    count++;
                    added=true;
                }else if(myOscAudio.getCount()<((OscAudio)this.myList.get(j))
                                                                .getCount()){
                    this.myList.add(j, myOscAudio);
                    count++;
                    added=true;
                }
            }
        }
    }
return this.myList;
}
```

3.6.2 AudioJitterBuffer

The AudioJitterBuffer class contains a float buffer of predefined size specified in the constructor by the jitter time period. This class has primarily two important methods that are used during its real time operations: push() and next(). The push() method inserts data into the buffer and has two arguments. The first is an integer that is used to determine whether the received packet is a duplicate or not. The second is the float array of audio data. Calls to this method are done in an asynchronous fashion. The next() method is synchronously controlled by the application invoking the method. The method removes data from the buffer. Each of the push() and next() methods have a synchronous block defined within them using its own object instance as the lock object in order to prevent data corruption.

3.6.2.1 The push() method

This method inserts audio data into the buffer in an asynchronous manner. It firstly checks the sequence number in the method argument so that packet duplication is prevented. Thereafter it proceeds to the synchronised block. Within this block the method checks to see if the spare capacity within the audio buffer is adequate for the incoming audio data. If it is not then the method simply returns without adding any data into the audio buffer. If there is capacity in the buffer then the incoming audio data is copied to the buffer. If the data contained within the audio buffer is above a threshold value[4] then waiting threads are woken up through the notify() method.

[4]defined as jitter size in bytes

3.6.2.2 The next() method

This method removes data from the audio buffer in a synchronised manner. The method is a synchronised block and blocks until the data quantity in the buffer surpasses the threshold. The method then checks to see if the requested data to be removed is less than the data contained within the audio buffer. If so then the data is removed. If not then then a zero filled float array is returned of the requested size.

3.7 Open Sound Control Environment

The architectural structure of OSC communication is typically one of a client-server model. The client initiates a connection request to a server which typically waits for such requests. When a request is received at the server, the server provides a service to the client depending on the application layer defined in the OSI model. In the case of OSC the server accepts incoming OSC messages from the client and therefore allows for a distributed system to be developed.

3.7.1 OSC Server

The OSC server deploys a service on the local IP address using a specified IP port. The transport layer in the Internet protocol suite is also defined here. In the prototype UDP is used. An OSC listener interface is implemented which is added to the OSC server. The interface contains a messageReceived() method which contains the constructed OSC address space. OSC messages destined for this IP socket on this server will be received as an argument in the messageReceived() method.

3.7.1.1 Mapping from VSTXML to OSC Address Space

The OSC server address space can be seen in Figure 3.11.

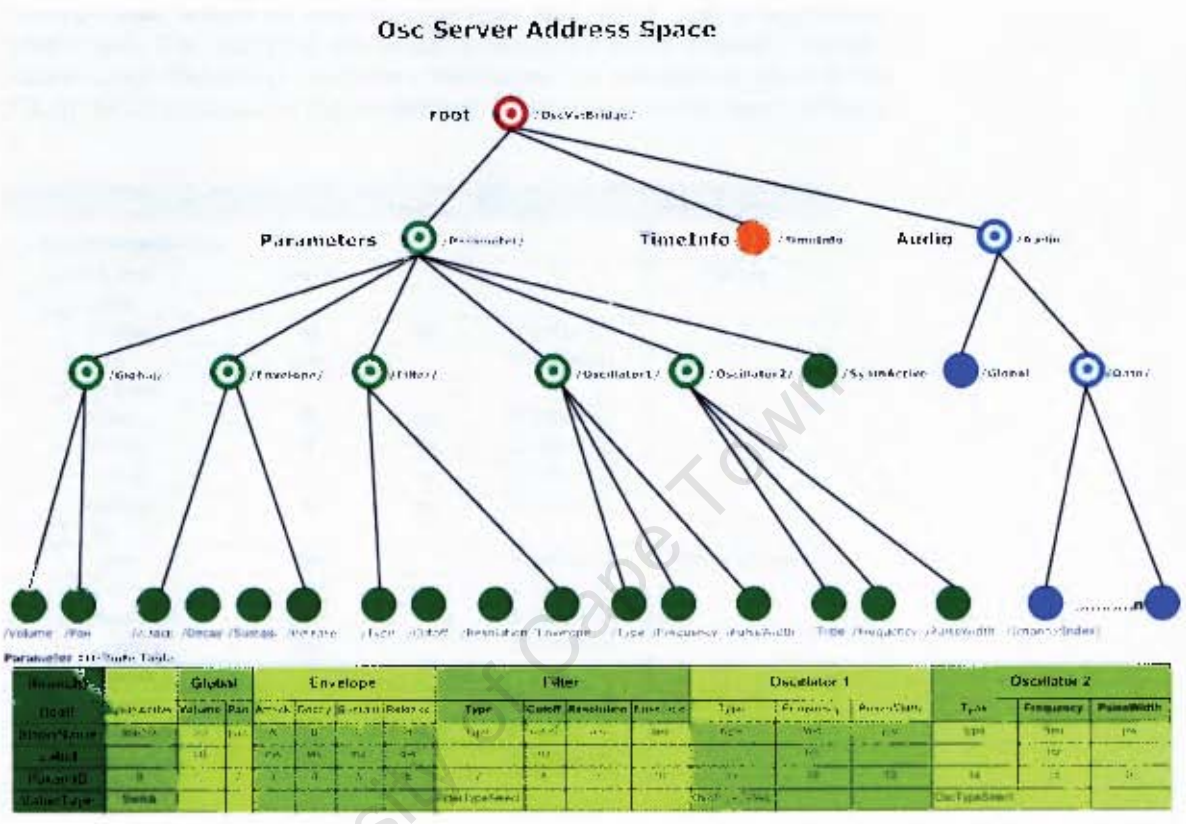

Figure 3.11: OSC server address space

The root of the OSC address space is */OscVstBridge/*. If the received OSC message does not contain this beginning segment in its OSC address pattern then the message is discarded by the OSC server. Only leaf nodes in the OSC address space have methods associated with them[38]. The OSC address space comprises of the three defined types: Parameter, TimeInfo and Audio. The Parameter node sub-tree is constructed from the VSTXML definition file parsing XML data. The */TimeInfo* node is a leaf node and is directly addressable.

The */Audio/* node contains a leaf node */Global* used to convey global audio parameters to the server application. The Audio node also contains another branch node */Data/* that has the same number of children as audio channels. These are labeled */(channelIndex)*.

The Parameter address space is constructed by importing a VSTXML file into an XML parser. The VSTXML file provides the hierarchical parameter structure of the plugin according the the VST SDK 2.4 definition. This definition contains 5 definition references. These are 'Param' describing a plugin parameter, 'Group' that is used to create hierarchical parameter structures and

contains 'Param' tags, 'Template' creates templates for parameter structures, 'ValueType' that defines an internal value type, and 'Entry' that is used inside a 'ValueType'. The VSTXML file found in Appendix B has 'Param', 'Group', 'Template', and 'ValueType' definition references. An hierarchical view of the VSTXML file as rendered by the application vstparamtool can be seen in Figure 3.12.

Name	Short Name	Label	Param ID	Value Type
OScVstBridgeRevTest				
Synth Active	synAct		0	Switch
Global				
Volume	vol	dB	10 (offset+0)	
Pan	pan		11 (offset+1)	
Envelope				
Attack	A	ms	20 (offset+0)	
Decay	D	ms	21 (offset+1)	
Sustain	S	ms	22 (offset+2)	
Release	R	ms	23 (offset+3)	
Filter				
Type	type		30 (offset+0)	FilterTypeSelect
Cutoff	cutoff	Hz	31 (offset+1)	
Resolution	res		32 (offset+2)	
Envelope	env		33 (offset+3)	
Oscillator 1				
Type	type		40 (offset+0)	OscTypeSelect
Frequency	freq	Hz	41 (offset+1)	
PulseWidth	pw		42 (offset+2)	
Oscillator 2				
Type	type		50 (offset+0)	OscTypeSelect
Frequency	freq	Hz	51 (offset+1)	
PulseWidth	pw		52 (offset+2)	

Figure 3.12: Hierarchical view of the VSTXML file

The XML parser uses the Java Document Object Model (JDOM) library which is used to map from the VSTXML file to the OSC namespace. The branch nodes of the Parameter sub-tree are either Param, Group or Template elements. The Param element is */SynthActive*. The Group elements are */Global/*, */Envelope/* and */Filter/*. The Template elements are */Oscillator1/* and */Oscillator2/*. The leaf nodes of each of the above Group and Template elements get their names from the attributes of the elements. These include ShortName,

Label, ParamID and ValueType.

The class used that contains the XML parser and maps from VSTXML to OSC namespace is the XmlParser class in the xmlParser package in the OscVstBridge prototype. The constructor of this class contains the algorithm that performs the mapping function from VSTXML to OSC namespace. The OSC namespace constructed can be seen as a treeView in Figure 3.16.

The children of the */Global/* node are */Volume* and */Pan*. The children of the */Envelope/* node are */Attack*, */Decay*, */Sustain* and */Release*. The children of the */Filter/* node are */Type*, */Decay*, */Sustain* and */Release*. The children of each of the */Oscillator1/* and */Oscillator2/* nodes are */Type*, */Frequency* and */PulseWidth*.

3.7.1.2 OSC Scheduling

OSC temporal semantics have the OSC timestamp as a core element. The OSC timestamp contained in the header of OSC bundles is a 64-bit high resolution timestamp conforming to NTP[23] definition. OSC messages travel between two host computers, and the protocol relies on external mechanisms to synchronise the host computers. The timestamp within an OSC bundle header gives the receiving OSC server the time the message is to be executed at the OSC server.

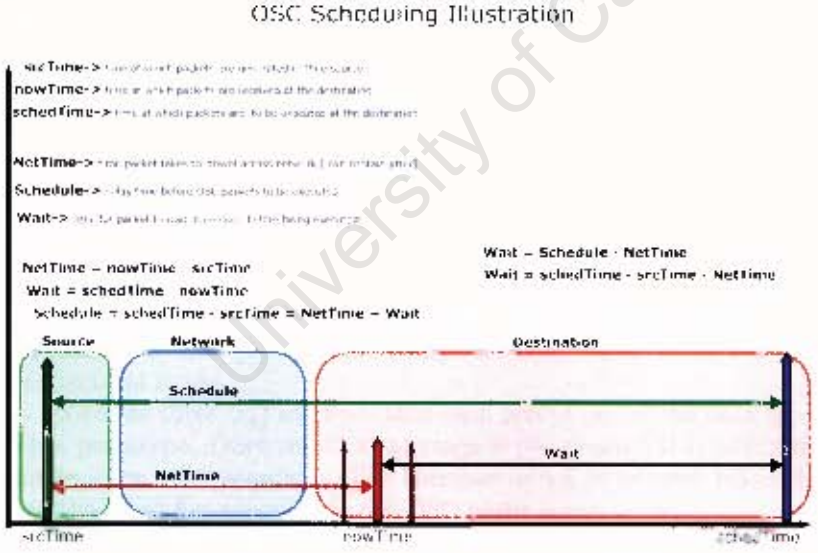

Figure 3.13: OSC Scheduling Illustration

Figure 3.13 illustrates the OSC Scheduling algorithm deployed in the OscVstBridge prototype. The scheduled execution time of the OSC message as transported in the OSC bundle header is *schedTime* and calculated using the delay *Schedule*. The algorithm requires the absolute time at the which the OSC

bundle or packet was generated in the source computer. This is *srcTime* and is transmitted as an OSC argument in the OSC message. When the OSC bundle is received at the destination computer in the OSC server, the time is logged as *nowTime*.

The time taken for the OSC message to traverse the computer network is therefore defined as:

$$NetTime = nowTime - srcTime$$

The time till the OSC packet is scheduled for execution is defined as:

$$Wait = schedTime - nowTime$$

or

$$Wait = Schedule - NetTime$$

The bridge class is core in the data flow for parameter and audio data types received from the computer network. The Bridge class therefore contains a ScheduledExecutor and ScheduledThreadPoolExecutor objects [5]that delay the data flow by breaking the original data flow and inserting a new data flow to the next data flow object. The next data flow object however only receives its data after the *Wait* time has elapsed by using the ScheduledThreadPoolExecutor's schedule() method. This method has the delay time and the executor object to run after the time has elapsed as arguments.

This mechanism therefore ensures that OSC messages are only executed at the time when they are scheduled to be.

3.7.2 OSC Client

The OSC client requires a destination IP socket for the transmission of OSC messages. The OSC client constructs the OSC message using the OSCPacketCodec in the Netutil library. Data received from the VST environment is packaged into OSC packets and sent to an OSC server using the client-server architectural model.

There are three tx() methods that each accept one of the data types defined in the prototype. Once the OSC message is constructed it is added to an OSC bundle. The OSC bundle is then encoded into a bytebuffer before being sent onto the IP socket specified in the OSC client constructor.

3.8 User Interaction

Users will interact with the system by using a VST host that loads the OscVstBridge VST wrapper prototype plugin. The prototype plugin will allow

[5]these classes are part of the JAVA API library

configuring of the OSC settings through a "COMMS" tab in order to communicate over Ethernet with external OSC devices. The OSC server as well as the OSC client will need an IP address and IP port. Routing UI elements are also needed to be configured to allow routing of data between the VST and OSC environments. The routing elements will allow for a route from the prototype's OSC client to its own OSC server facilitating internal or loopback communications. The plugin will contain 17 parameters that are controllable from both the VST host, OSC environment as well as from the plugin UI itself. Other datatypes such as TimeInfo and Audio will be displayed as real time data in the plugin UI.

3.8.1 UI Elements

The prototype user interface is a Java swing interface consisting of four tabs labeled COMMS, PARAMETER AND AUDIO, ADDRESS SPACE and EXPERIMENTS. The user interface is contained within a plugin window provided by the VST host.

3.8.1.1 COMMS Tab

Figure 3.14: COMMS tab

The Comms tab in Figure 3.14 consists of OSC server and OSC client communication settings as well as information displays on those settings. There are also JitterBuffer settings.

The OSC server settings consists of the OSC server IP address and IP port. When these values are applied they are displayed in the OSC server status info window.

The OSC client settings are similar with a destination IP address and IP port specified. Another parameter is the OSC bundle delay in milliseconds. When applied the OSC client status info window shows the stored settings.

The JitterBuffer setting can be changed and when applied displays to the user the current value used by the plugin.

3.8.1.2 PARAMETER and AUDIO tab

Figure 3.15: PARAMETER and AUDIO tab

The tab in Figure 3.15 consists of two parameters and the audio ins and outs.

There are 17 parameters in the prototype of which only two[6] are displayed to the user. The choice of the two parameters displayed to the user is done in the Address Space tab. In Figure 3.15 the two displayed parameters are the

[6]Parameter A and Parameter B

first two: SynthActive and Volume with indices 0 and 1. Each group displays the name of the parameter, the slider used to change the normalised value in the parameter and the displayed parameter value. There are also transmit and receive selection checkboxes for the parameter that enable transmitting and receiving of OSC messages respectively. For each parameter transmit and receive LEDs and counters are shown. Within the UI there is a built in mechanism that prevents the transmit and receive checkboxes from being simultaneously enabled if the OSC client's destination socket is the internal OSC server. This prevents infinite looping when parameter changes are done.

The audio display is segmented into left and right. The transmit and receive checkboxes as well as the LEDs and counter operate in a similar fashion to the parameter counterparts. However the transmit and receive checkboxes can be selected simultaneously as the audio ins and outs are independent of each other. The audio counters count the OscAudio objects processed which is typically related to the audio blocks received by the plugin from the VST host.

3.8.1.3 ADDRESS SPACE tab

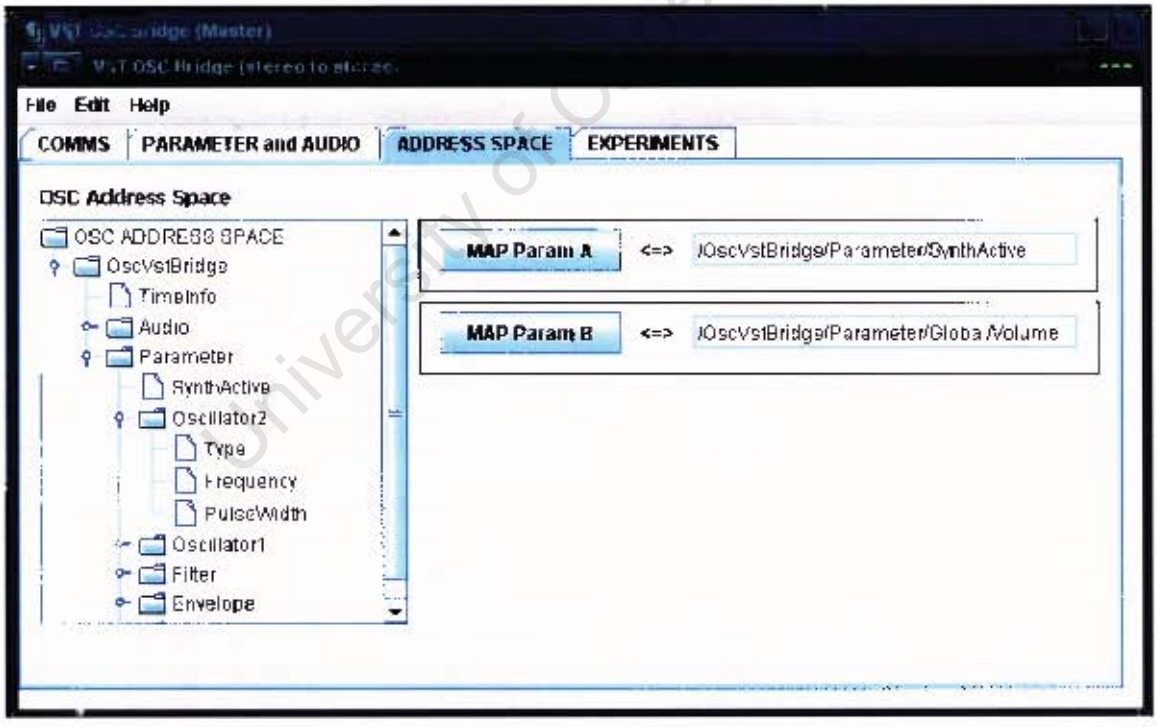

Figure 3.16: ADDRESS SPACE tab

The tab in Figure 3.16 contains the parameter address space displayed in tree form. This allows the user to navigate the parameter address space [7] and select the parameter wanting to be used in the Parameter and Audio tab. The user has the option of mapping the selected parameter to display Parameter A or B. In Figure 3.16 it can be seen that Parameter A is mapped to */OscVstBridge/Parameter/SynthActive* and Parameter B is mapped to */OscVstBridge/Parameter/Global/Volume*.

The EXPERIMENTS tab at the time of developing the prototype was left empty. Its intended purpose was to allow the user/experiment conductor to input experimental data as well as to display such data while conducting the experiments of this dissertation.

3.8.1.4 Parameter UI Element

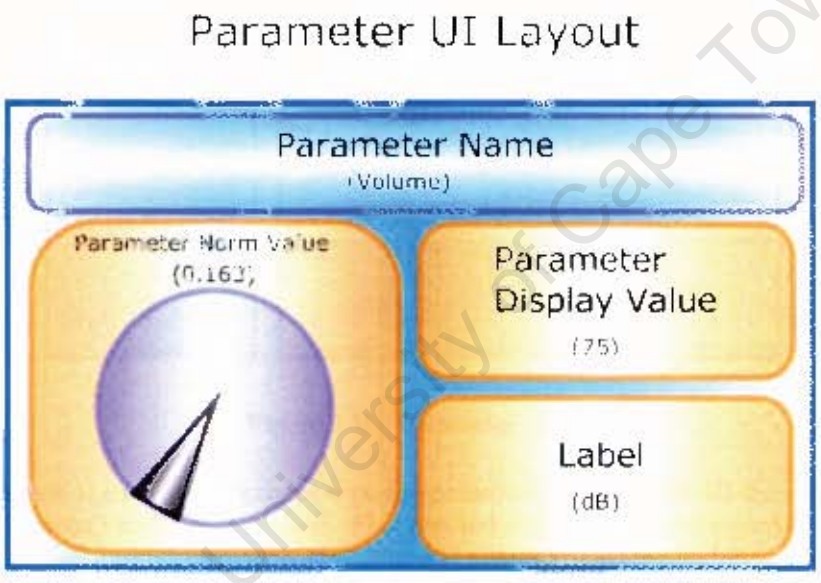

Figure 3.17: Parameter UI

The Parameter class members are index, pName, normValue, dispValue and label. These members are presented in the Parameter UI except the index. The pName is the Parameter Name. The normValue is the normalised value that indicates the position of the dial or slider to the user. This value is also used internally by the plugin. The dispValue is the displayed value to the user and is a result of a function with the normValue as an input to that function. The display value is an understandable value for the user. For example it would make no sense to a user to see that the Volume level is 0.163 when a dB value

[7] constructed from the VSTXML file

of 75 has more meaning. The label is the label attached to the displayed value to the user. In Figure 3.17 this dB indicates the volume to the user.

3.8.1.5 Updating Parameter UI

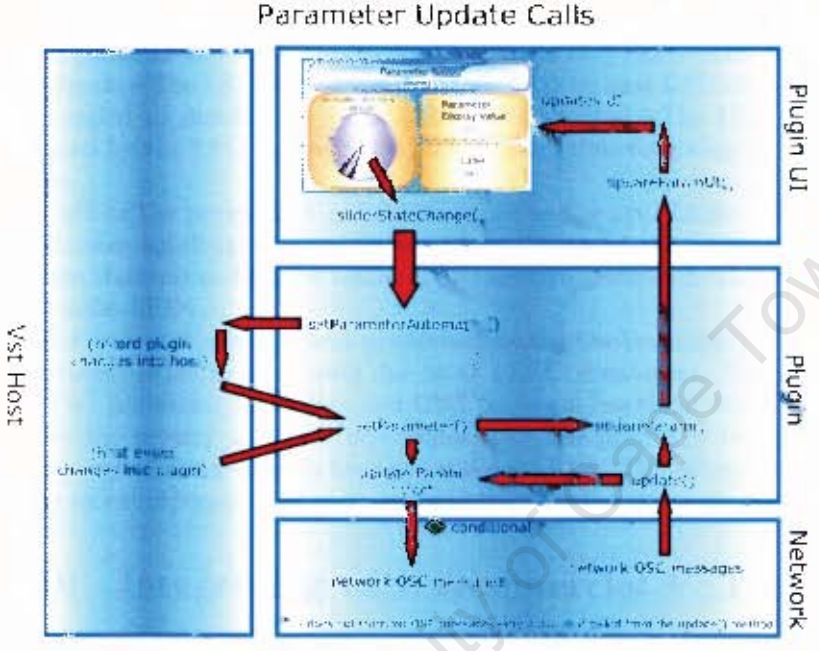

Figure 3.18: Parameter UI updating

When the user makes changes to the parameter slider in the UI the sliderStateChange() method is invoked. This method calls the setParameterAutomated() method [8] which calls the setParameter() method. This method then updates the parameter object in the plugin, transmits the changes to the OSC environment and calls the updateParam() method.

If the VST host sends its own event changes to the plugin, then the setParameter() method is called which again updates the param object, transmits OSC messages onto the network and calls the updateParam() method. The updateParam() method then calls the updateParamUI() method which updates the UI.

If the user enables the record facility within the VST host then all parameter slider changes are recorded into the host.

If OSC messages are received from the network the update() method in the Observer class is called. This method then updates the parameter object but does not transmit onto the OSC network. The update() method also calls

[8] which notifies the VST host of the changes for recording purposes

the updateParam() method which calls the updateParamUI() method which updates the UI with the changes.

3.9 Threading

3.9.0.6 Gui threading

The parameter and audio tab in the UI consists of 8 LEDs thats are simulated in software. The LED simulation contains LED on and LED off images which are toggled to simulate an LED 'pulse on' operation. The LED off image is replaced by the LED on image for 100ms, thereafter returning to the LED off image.

Each of the parameters contain transmit and receive LEDs. There are two parameters totalling four parameter LEDs. Each of the audio channels also contain transmit and receive LEDs. There are two mono audio channels totalling four audio LEDs.

For each OSC message sent or received by the OscVstBridge prototype there is a 100ms delay in processing due to the LED simulation. During this time no other processing is possible and OSC messages less than 100ms apart would render the prototype system non-responsive. The prototype therefore deploys separate processing threads for each of the 8 LEDs. This allows the prototype to be more responsive and meet the audio processing demands placed on it.

3.10 Integrating with commercial VST plugins

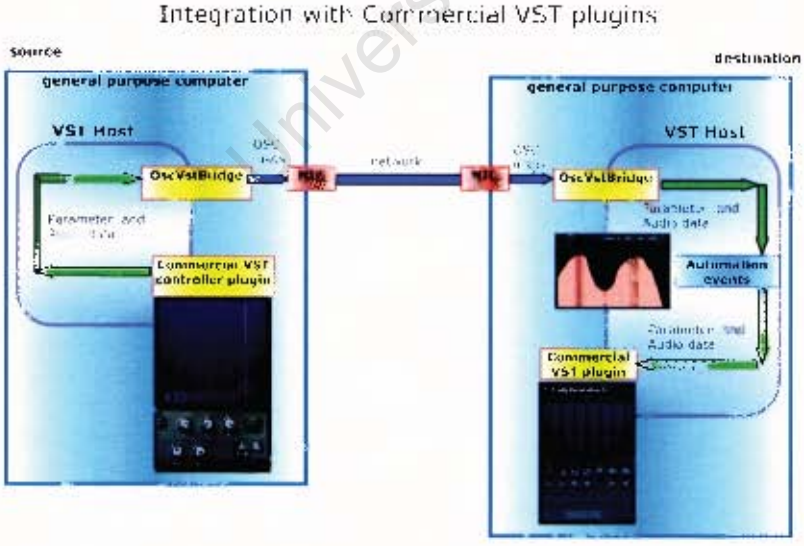

Figure 3.19: Integrating Commercial VST plugins with OscVstBridge

VST hosts support interfacing external MIDI controllers as well as VST plugin controllers to the VST effect and instrument plugin environments via MIDI CC[9]. Controller inputs can be freely mapped to any MIDI CC in the VST host. The same MIDI CC set can also be mapped to output parameters. This can include VST plugin effects or VST host controllable objects. In this way a dynamic and flexible mapping is achieved between the controllers and VST plugins.

It is also possible to record and automate parameter changes made within VST plugins to the VST host. Thes changes are recorded as automation events and can be dynamically appied to other parameter or controllable objects within the VST host.

In Figure 3.19 it can be seen that a commercial controller VST plugin can be linked via MIDI CC to the parameters of the OscVstBridge prototype. Changes made to the controller plugin are then bridged in the prototype to OSC messages that are transmitted onto a network. On a separate computer running a similar setup[10] the OscVstBridge plugin accepts the OSC messages from the network converting them to parameter data that can be recorded as automation events by the VST host. These automation events can then be the source of controller data to commercial VST plugins instantiated on this destination computer.

VST hosts have available virtual audio busses for dynamic routing of audio data. A source commercial VST plugin can route its audio data onto a virtual bus where the OscVstBridge prototype can pick it up and transmit it as OSC messages onto the network. In the destination computer the OscVstBridge prototype converts the OSC messages into audio data routing it onto a virtual bus configured on this VST host. Any commercial VST plugin that accepts audio data as an input can source the audio data from the virtual bus on the destination computer.

Figure 3.19 illustrates how the parameter and audio data resources are shared between VST hosts on separate computers. Sharing these critical parameter and audio resources paves the way for computation and memory resources to be distributed.

[9]MIDI continuous controller commands

[10]VST host with OscVstBridge and commercial VST plugins instantiated.

Chapter 4

Evaluation and Results

4.1 Introduction

The prototype evaluation categorises the evaluation in terms of the 2 data types used within the system. These are the parameter or control data type and the audio data type developed. For the parameter data type evaluation found in experiments 1 to 4, the aim was to measure the timeline category of QOS such as latency and jitter. In the Quality of Perception Category of QOS the scheduling or packet order preservation was measured. Packet loss was also measured. These metrics were compared to those presently found in the musical control industry[1] and a conservative benchmark was set against which to measure.

For the audio data type evaluation in experiments 5 to 7 the control mechanisms that remove jitter and ensure audio data sequence is preserved were compared against one another. These control mechanisms are the developed AudioReceiver containing jitter removal and order enforcement algorithms, and the OSC bundle scheduling. The metrics are scheduling, latency, jitter and temporal fidelity.

The computer networks used in experiments 1-4 were the internal loopback interface[2] as well as a simple configuration using a single network unmanaged switch with CAT5 UTP Ethernet cables no longer than 5m between the switch and the workstations each running instances of the prototype. The switch is a 100MBit Ethernet SMC switch.

The NICs are Broadcom NetXtreme Gigabit Ethernet Network adapters. The computers used are HP NC8230 notebooks, each with a 1.6Ghz Pentium 4 processor and 1gig of RAM.

The networks used in experiments 5-7 include two end computers running instances of the prototype. These computers were networked together using a third computer that has two NICs running router software called NetDisturb by ZTI. NetDisturb is capable of generating latency and jitter into the network.

[1] such as MIDI

[2] commonly known as 'localhost' or using IP address 127.0.0.1

4.2 Parameter Experiments

Parameter meeting QOS benchmarks for various performance indicators

4.2.1 Introduction

The metrics measured are latency, jitter, scheduling and packet loss. The performance indicators of the prototype processing were determined as the network effects can vary for different network topologies. Two network experiment architectures were tested that provided a maximum and minimum range in which the metrics fell. The application environment in which the prototype is most likely to be deployed would have network architecture similar to a LAN with a single broadcast domain. Although sending packets across routers is possible in the prototype it is highly unlikely and would occur in a specialised application environment.

The effects of the LAN network topology band can be seen in Figure 4.1 below:

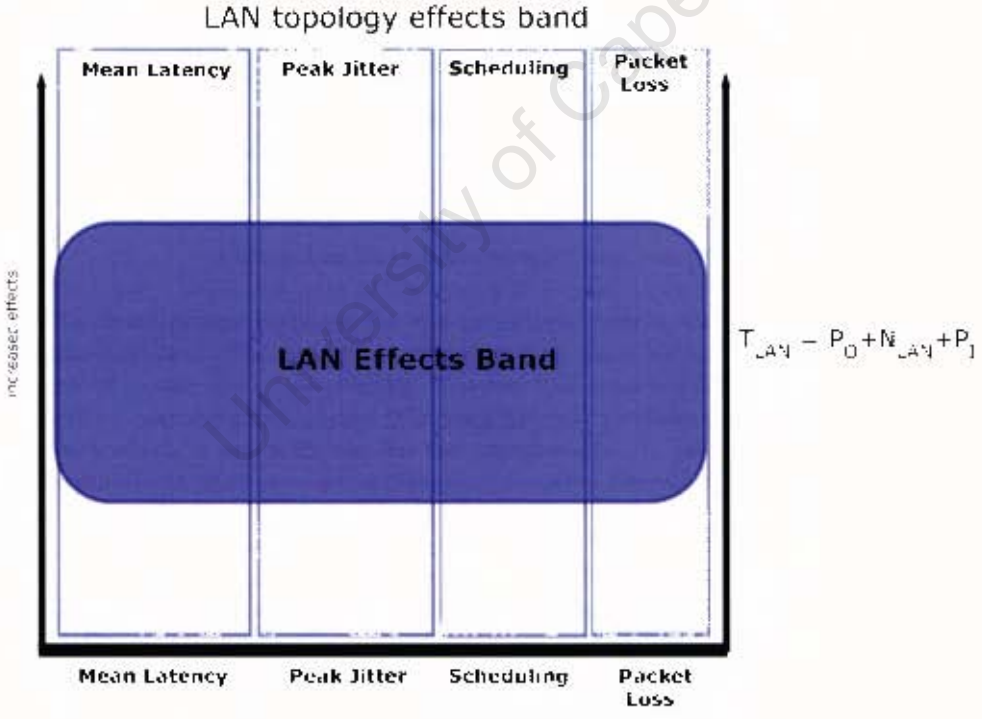

Figure 4.1: LAN topology effects band

The effects of the LAN network can be modeled by the equation:

$$T_{LAN} \cdot P_O + N_P - P_I$$

where
T_{LAN} is the effects of the LAN networked system,
P_O is the output processing effects of the prototype system, and
P_I is the input processing effects of the prototype system, and
N_P is the LAN network effects.

The system under test for the LAN network would resemble Figure 4.2.

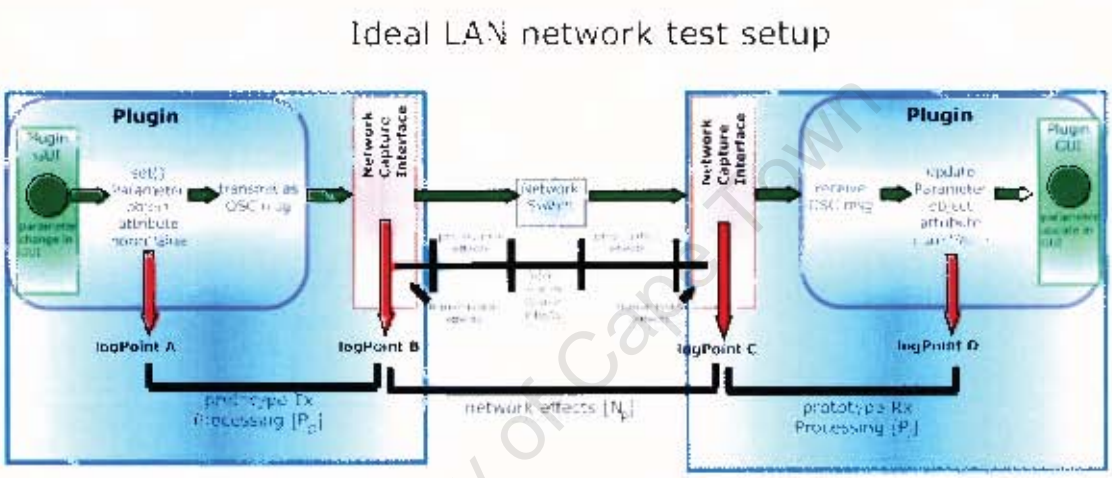

Figure 4.2: Ideal LAN network test setup

The development platform for the prototype system was a Windows XP operating system. The prototype is hosted by a Java Virtual Machine and is capable of picosecond timestamping. However this accuracy is dependent on the underlying operating system and Windows XP only provides tens of millisecond accuracy which is not sufficient for the experiments. A Java API called PreciseInternalDate that provides millisecond accurate timestamping within Windows XP on the host machine is used.

Attempts to accurately time synchronise two Windows XP host machines have been unsuccessful using NTP services. The NTP tool NetTime was used as well as another called Meinberg NTP but the offset between the two machines were still in the tens of milliseconds. This value was determined by developing a simple clock offset tool using Java, the PreciseInternalDate API and OSC. The principle of operation is similar to that described in the NTP specification[23]. It was uncovered that despite NTP synchronisation between the hosts, an offset between 80 and 110 milliseconds was observed. However the offset intervals were consistently above 8ms. It was therefore decided to discard using two host machines in the experimental setup due to the error rate introduced by the operating system.

Internal Loopback Interface Network

The first network setup used in the experiments used the internal loopback network interface provided on most network interface cards. This loopback interface has negligible latency and jitter as packets traverse only a portion of the TCP/IP protocol stack within the network interface and do not reach any physical network. The same instance of the prototype plugin was used as it is capable of both sending and receiving data simultaneously. This test set aimed to provide a minimum value for each of the metrics as any network effects introduced into the system will only be a measure of the prototype system effects and the internal negligible protocol stack effects.

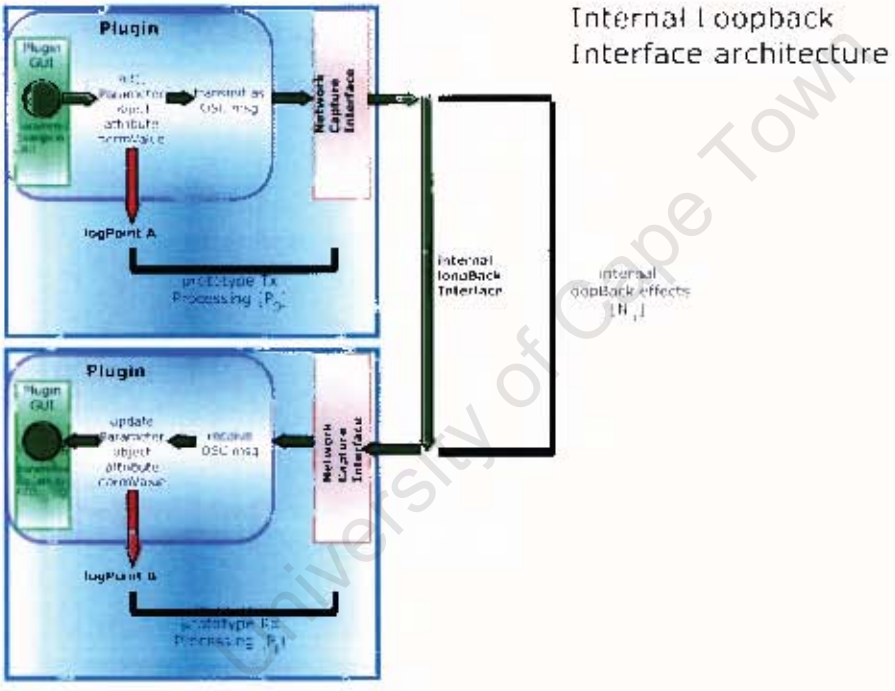

Figure 4.3: Internal loopback network architecture

The effects of the internal loopback network can be modeled by the equation:

$$T_{IL} = P_O + N_{IL} + P_I$$

where
T_{IL} is the effects of the internal loopback system,
P_O is the output processing effects of the prototype system, and
P_I is the input processing effects of the prototype system.
N_{IL} is the internal loopback network stack effects.

We can see that if $N_{IL} < N_P$ then $T_{IL} < T_{LAN}$.
We also know that

$$N_P = N_{STACK} + N_{TRANSMISSION} + N_{STACK}$$

and since the N_{IL} only traverses a portion of the protocol stack we can assume that

$$N_{IL} = N_{STACK}$$

We can therefore see that $N_{IL} < N_P$ and therefore $T_{IL} < T_{LAN}$.
From this the internal loopback band in Figure 4.4 is deduced.

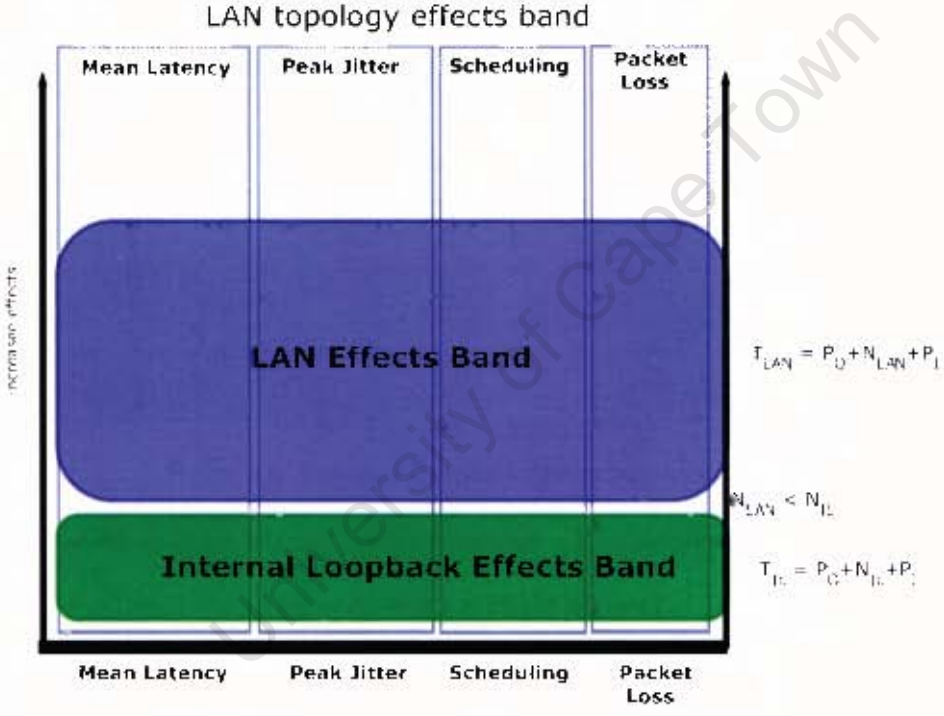

Figure 4.4: Internal loopback effects band

External Loopback Server over physical network

The second test involved a physical 100Mbit Ethernet network connected through a unmanaged switch. Two host computers were connected to the switch via 3m UTP patch leads. This gave an indication of the performance of the system that is inclusive of the network effects.

This test included an external loopback server located on a separate host machine. The server reflects OSC packets received back to the source of the

packets. The server timestamps incoming and outgoing packets and hence gives an indication of the processing time required by this server. This is illustrated in Figure 4.5.

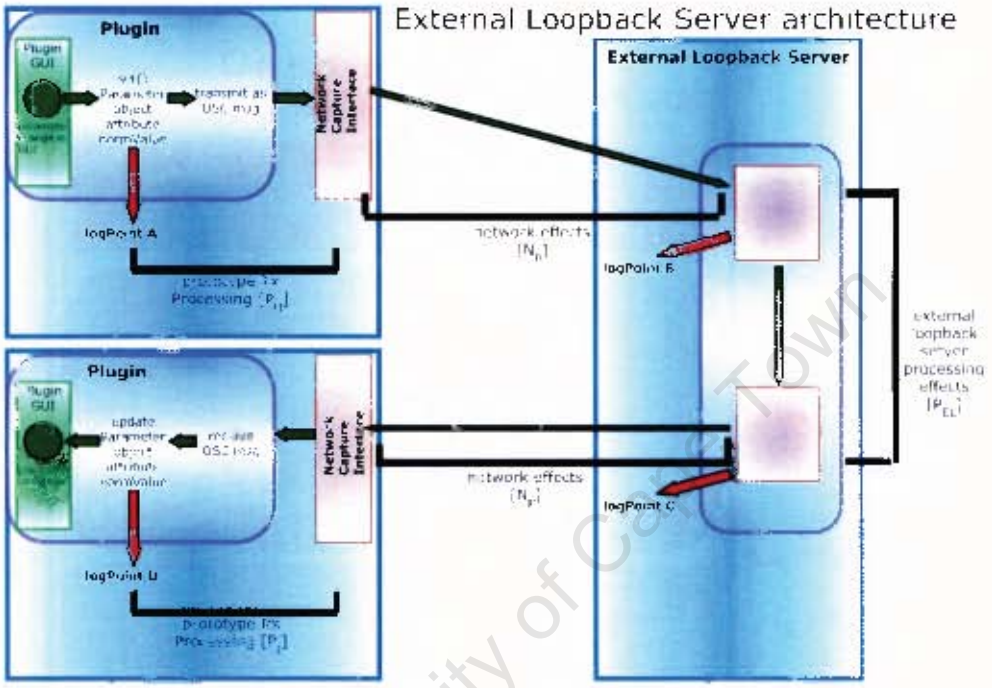

Figure 4.5: External Loopback Server Architecture

The effects of the internal loopback network can be modeled by the equation:

$$T_{EL} = P_O + N_P + P_{EL} + N_P + P_I$$

where
T_{EL} is the effects of the external loopback server system,
P_O is the output processing effects of the prototype system, and
P_I is the input processing effects of the prototype system, and
P_{EL} is the external server processing effects, and
$N_P$is the physical network effects.

Rewriting T_{EL} we have

$$T_{EL} = P_O + N_P + P_I + P_{EL} + N_P$$

and substituting

$$T_{LAN} = P_O + N_P + P_I$$

we have

$$T_{EL} = T_{\text{LAN}} \mid P_{EL} \mid N_P$$

We can therefore see that $T_{EL} > T_{LAN}$.
From this the external server loopback band in Figure 4.6 is deduced.

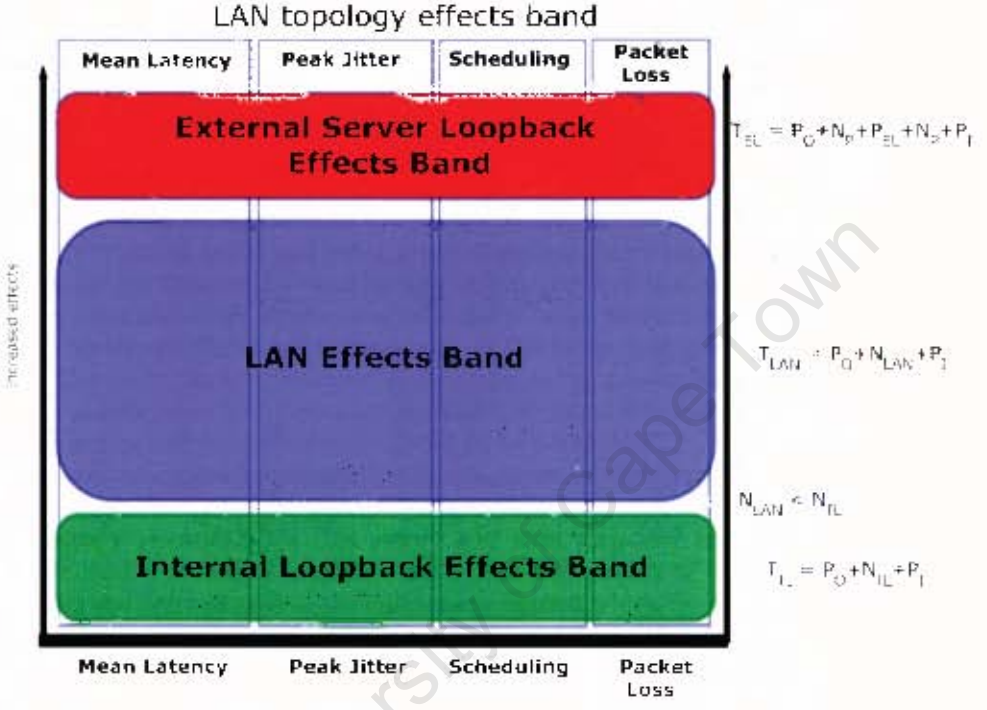

Figure 4.6: External Server loopback effects band

The first 4 experiments aim to demonstrate that the effects for a LAN network is bounded by the External Server Loopback Effects band and the Internal Loopback Effects Band. A typical implementation of the prototype is expected to perform within these limits.

Outline of parameter experiments

The following table indicates the structure of the experiments conducted on the Parameter data type. The upper and lower bounds were determined for each metric using the internal loopback and then the external loopback server. Conclusions were then drawn on that metric. The latency and jitter as well as the scheduling and packet loss metrics are combined as the data set used is the same for both.

Metric	Latency	Jitter	Scheduling	Packet Loss
internal loopback	Experiment 1	Experiment 1	Experiment 3	Experiment 3
external loopback server	Experiment 2	Experiment 2	Experiment 4	Experiment 4

Table 4.1: Parameter experiments and metrics

4.2.2 Experiment 1:

PARAMETER END-TO-END LATENCY AND JITTER MEASURE USING INTERNAL LOOPBACK INTERFACE

4.2.2.1 Introduction

The parameter data type is of a musical control type and can be compared to that expected of MIDI and other musical control definitions. This experiment measured the mean end-to-end latency and approximated the peak jitter of the system and measured against musical control benchmarks accepted in industry. Such benchmarks are hard to determine as the tempo and genre of music influence these significantly. Sounds can be perceived as coming from a single source if the sounds occur less than 25ms apart[20], however latencies of 50ms in piano experiments and even 60ms were found to be acceptable.

Instrument interface design guidelines prescribe that performance latencies not exceed 10ms[20]. The peak jitter is obtained by comparing MIDI interface technology results[37] for this metric and it is proposed that the prototype obtain similar results. Therefore a peak jitter benchmark of 10ms is used. This latency and jitter is used as the benchmark against which the parameter latency and jitter metrics in the prototype are measured for a LAN network setup. We therefore aim to show that the upper bound is less than 10ms for both latency and jitter.

4.2.2.2 Aim

This experiment measures the parameter latency of the prototype using the internal loopback network interface. It also provides an estimate of the peak jitter introduced by the prototype. For this test the latency consists of the prototype OSC message output latency, the internal loopback latency and the prototype OSC message input latency. This latency is compared to the 10ms benchmark latency and to determine the lower bound.

The jitter is measured as the difference in latency between consecutive packets received. The peak jitter is also approximated from the jitter data by observing its dispersion.

4.2.2.3 Methodology, Setup and Architecture

This experiment was conducted by instantiating two separate instances of the prototype within a VST host. The first instance would perform exclusively as an OSC client transmitting OSC messages onto the internal loopback interface.

The second instance would act as a server to the internal loopback interface receiving OSC messages transmitted by the first instance. Figure 4.3 illustrates an architectural diagram of the experiment.

OSC messages are generated by interfacing with the prototype GUI altering the parameter slider until about 500 OSC messages have been generated.

Within the first instance of the plugin where OSC messages are transmitted, a timestamp is logged where the Parameter object is acted upon. In this case it is the set() method on the Parameter object. The ID of the message is logged with the timestamp to identify packets.

In the second instance of the OSC server receiving packets, the timestamp together with the ID is logged where the Parameter object in this instance is acted upon by the update() method.

The timing used is from the Epoch, a measure of the number of milliseconds elapsed since midnight January 1st 1970 UTC.

The latency is therefore a measure of the difference between the OSC server in instance 2 timestamp and its corresponding message determined by ID in the OSC client in instance 1. The mean, standard deviation as well as a probability density function graph is generated from the data set.

The jitter measures the difference between the latencies of consecutive packets. The mean, standard deviation and probability density function graph is generated. The peak jitter is determined from the jitter standard deviation. Specifically:

$$Jitter_{peak} = 2 * 2\sigma\mu_{jitter}$$

as the jitter distribution is double sided. This peak jitter value therefore accounts for 95.45% of the values observed.

This method is used because the sample data has spike values and the conventional method of obtaining peak jitter by subtracting the minimum from the maximum in the sample set will not yield accurate results.

Metric_1 latency

Metric_2 jitter

4.2.2.4 Result Data and Findings

METRIC 1: LATENCY

Result Data

The results for the latency metric are found in the Table 4.2 and Figures 4.7 and 4.8 below:

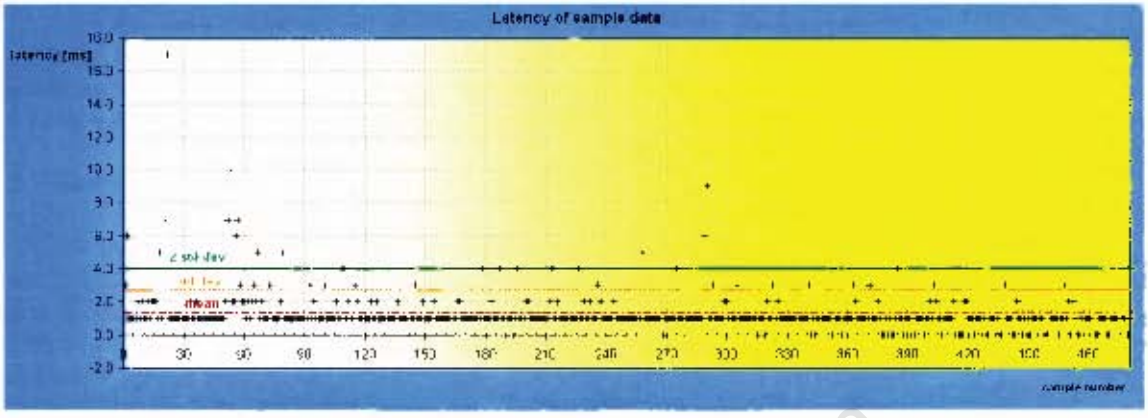

Figure 4.7: Latency of Sample Data

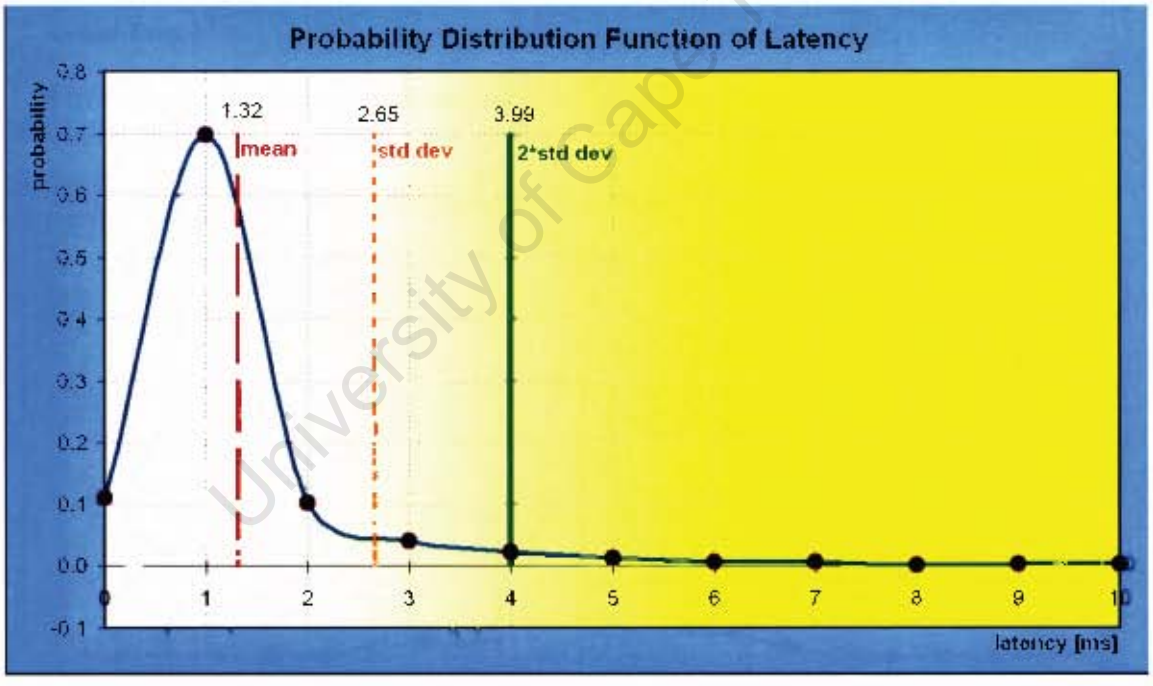

Figure 4.8: Probability Distribution Function graph of latency

Sample Size	Mean	Std Dev	Mean - Std Dev	Mean + 2*Std Dev	Variance	SS
509	1.32	1.344	2.66	3.99	1.81	917.33

Table 4.2: Experiment 1 result data for latency metric

From the data above we can see that the mean latency for the sample data set is 1.32ms with a standard deviation of 1.344. The probability distribution graph appears to be similar to a lognormal or gamma distribution that achieves a peak within a short latency and then tapers off.

Findings

The lower bound mean latency provided by the internal loopback network setup is 1.32ms. The 2 standard deviations from the mean with a value of 3.99ms give the maximum latency in which 95.45% of the sample set lie. This value of 3.88ms therefore gives an acceptable latency for musical control and provides the lower bound used in the experiments. The graph resembles a lognormal or gamma distribution and therefore tapers off at greater latencies. This indicates a very low probability of latencies near 10ms being achieved.

Metric 2: Jitter

Result Data

The results for the jitter metric are found in the Table 4.3 and Figures 4.9 and 4.10 below:

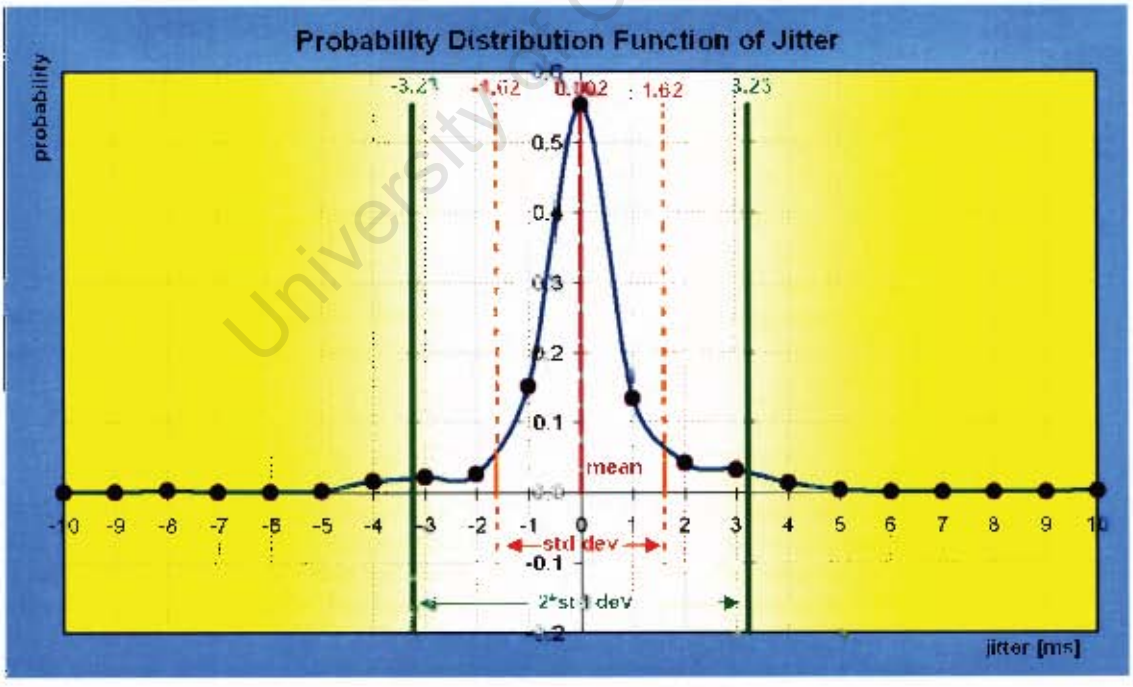

Figure 4.9: Probability Distribution Function graph of jitter

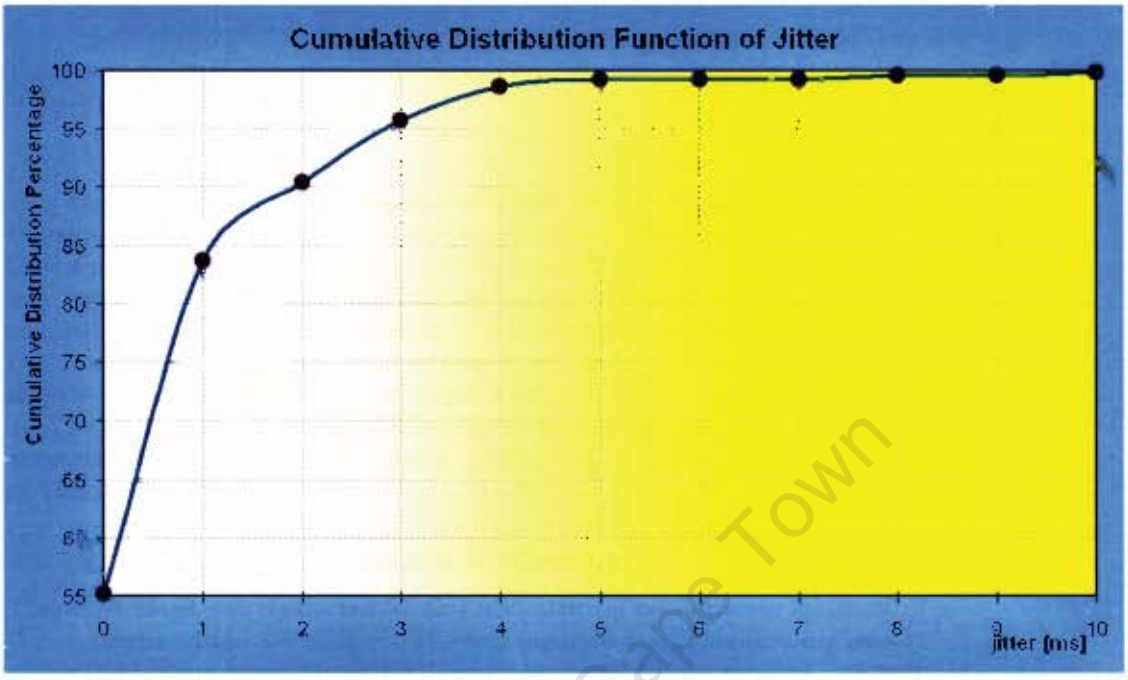

Figure 4.10: Cumulative Distribution Function of jitter

Sample Size	Mean	Std Dev	Mean−Std Dev	Mean− 2*Std Dev	Variance	SS
509	0.02	1.61	1.62	3.23	2.60	1322.998

Table 4.3: Experiment 1 result data for jitter metric

From the data above we can see that the mean jitter for the sample data set is 0.02ms with a standard deviation of 1.614. The probability distribution graph is a normal distribution with a narrow dispersion band.

Findings

The lower bound mean jitter provided by the internal loopback network setup is 0.02ms. The 2 standard deviations from the mean gives a value of 3.23ms and in the cumulative distribution graph of Figure 4.10 the curve 'flattens' at a latency of about 4ms. This indicates the contribution of the samples below 4ms make up the 98.62% of the samples. Therefore the peak jitter if taken from the cumulative distribution graph is 8ms in which 98.62% of the sample set lie. This value of 8ms peak jitter gives a reasonably acceptable jitter for musical control and provides the lower bound used in the experiments and falls within the 10ms jitter benchmark set.

4.2.3 Experiment 2:

PARAMETER END-TO-END LATENCY AND JITTER MEASURE USING EXTERNAL LOOPBACK SERVER

4.2.3.1 Aim

This experiment measures the parameter latency of the prototype using the external loopback server. It also provides an estimate of the peak jitter introduced by the prototype. For this test the latency consists of the prototype OSC message output latency, the physical network latency, the external loopback server processing latency, the return path physical network latency and the prototype OSC message input latency. This latency is compared to the 10ms benchmark latency and used to determine the upper bound.

The jitter is measured as the difference in latency between consecutive packets received. The peak jitter is also approximated from the jitter data by observing its dispersion.

4.2.3.2 Methodology, Setup and Architecture

This experiment was conducted by also instantiating two separate instances of the prototype within a VST host. The first instance would perform exclusively as an OSC client transmitting OSC messages towards the external loopback server located on a separate networked host machine. The second instance would act as a server to the external loopback server receiving OSC messages transmitted by the external loopback server. The loopback server upon receiving messages would timestamp the incoming message and then transmit that message back to the receiving host timestamping the time of transmission. The processing time of the external loopback server is then also determined. Figure 4.5 illustrates an architectural diagram of the experiment.

OSC messages are generated by interfacing with the prototype GUI altering the parameter slider until about 300 OSC messages have been generated.

The timestamp logging points, methodologies and metrics are the same as in experiment 1.

Metric_1 latency

Metric_2 jitter

4.2.3.3 Result Data and Findings

METRIC 1: LATENCY

Result Data

The results for the latency metric are found in the Table 4.4 and Figures 4.11 and 4.12 below:

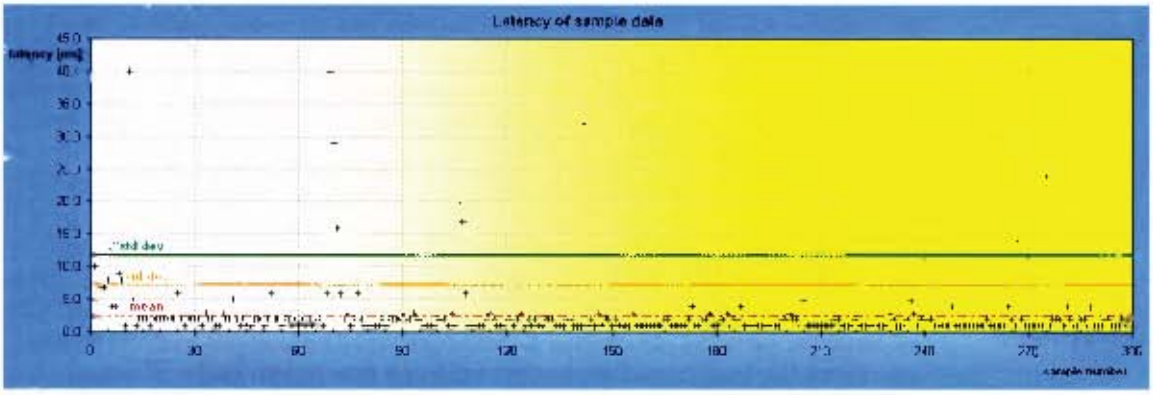

Figure 4.11: Latency of Sample Data

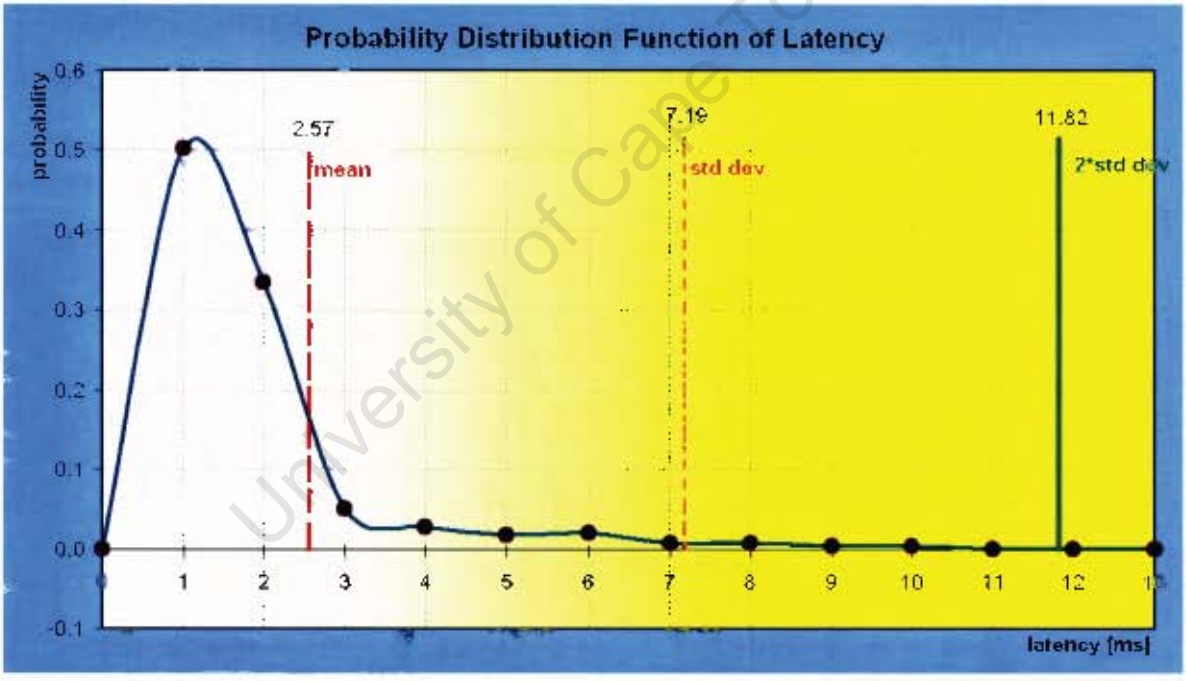

Figure 4.12: Probability Distribution Function graph of latency

Sample Size	Mean	Std Dev	Mean + Std Dev	Mean + 2*Std Dev	Variance	SS
299	2.57	4.62	7.19	11.82	21.36	6365.20

Table 4.4: Experiment 2 result data for latency metric

From the data above we can see that the mean latency for the sample data set is 2.57ms with a standard deviation of 4.62. The probability distribution graph appears to be similar to a lognormal or gamma distribution that achieves a peak within a short period of time and then tapers off.

Findings

The upper bound mean latency provided by the internal loopback network setup is 2.57ms. The 2 standard deviations from the mean with a value of 11.82ms give the maximum latency in which 95.45% of the sample set lie. This value of 11.82ms has reached the boundary of acceptable latencies required for instrument interface design and provides the upper bound used in the experiments. The graph resembles a lognormal or gamma distribution and therefore tapers off at greater latencies. This indicates that there is a less than 5% probability of the latencies in the sample data set reaching beyond the 11.82ms mark.

Metric 2: Jitter

Result Data

The results for the peak jitter metric are found in the Table 4.5 and Figures 4.13 and 4.14 below:

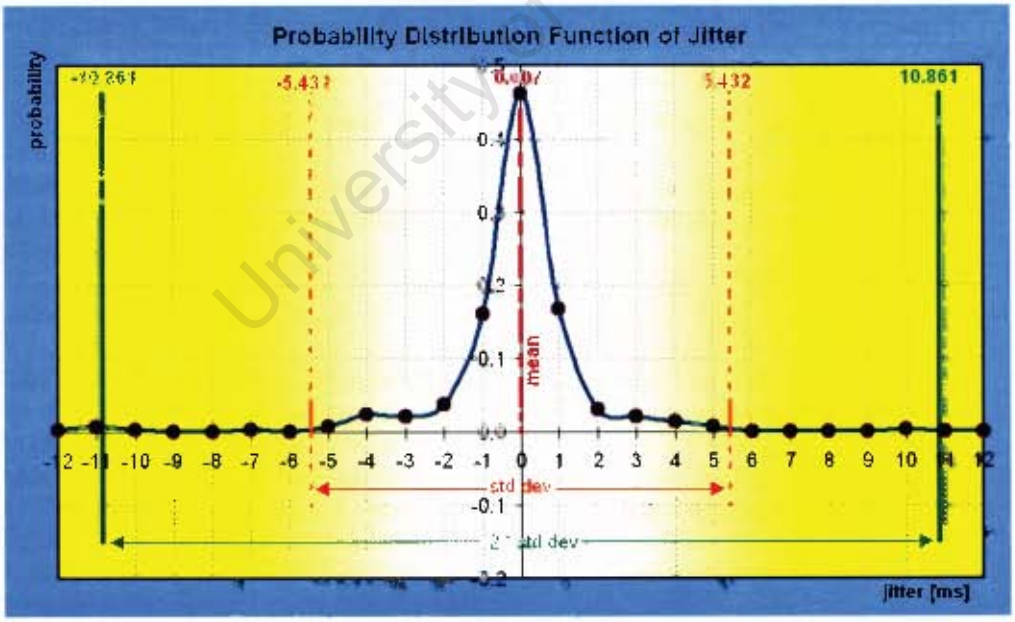

Figure 4.13: Probability Distribution Function graph of jitter

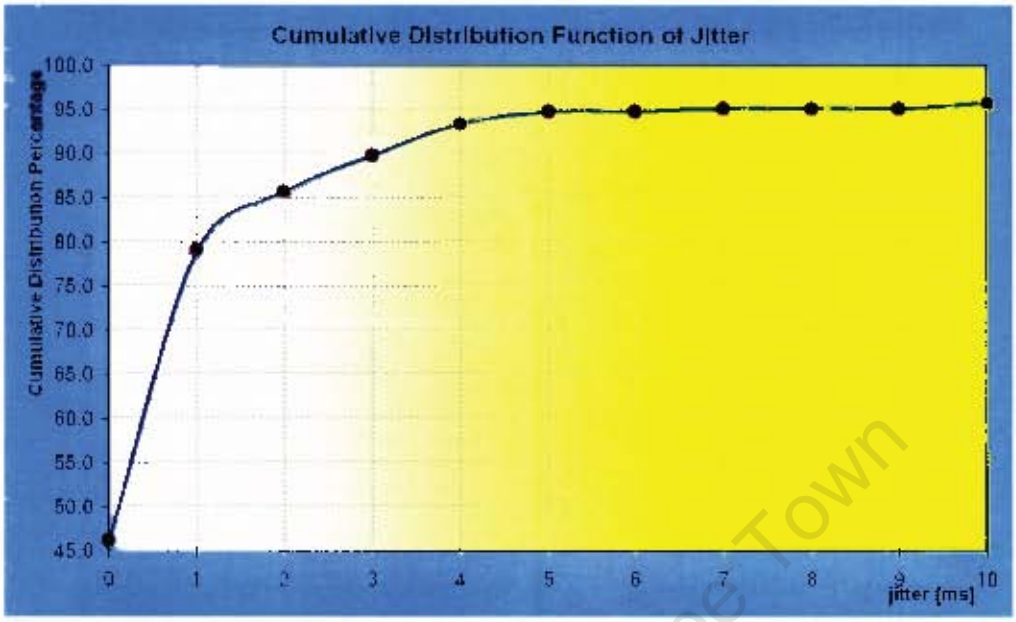

Figure 4.14: Cumulative Distribution Function graph of jitter

Sample Size	Mean	Std Dev	Mean+Std Dev	Mean− 2*Std Dev	Variance	SS
299	0.007	5.427	5.432	10.861	29.456	8777.99

Table 4.5: Experiment 2 result data for jitter metric

From the data above we can see that the mean jitter for the sample data set is 0.007ms with a standard deviation of 5.427. The probability distribution graph is a normal distribution.

Findings

The upper bound mean jitter provided by the internal loopback network setup of 0.007ms. The 2 standard deviations from the mean gives a value of 10.861ms. However from the cumulative distribution graph the curve 'flattens' at 5ms indicating that 94.65% of the samples lie below this latency value of 5ms. Therefore the peak jitter if taken from the cumulative distribution graph in Figure 4.14 is 10ms in which 94.65% of the sample set lie. This value of 10ms peak jitter meets exactly the 10ms jitter benchmark set and should be found to be acceptable for musical control performance.

Discussion on Experiment 1 and 2 - Upper and Lower bounds

The upper and lower bounds for the mean latency and peak jitter metrics can be seen in Figure 4.15.

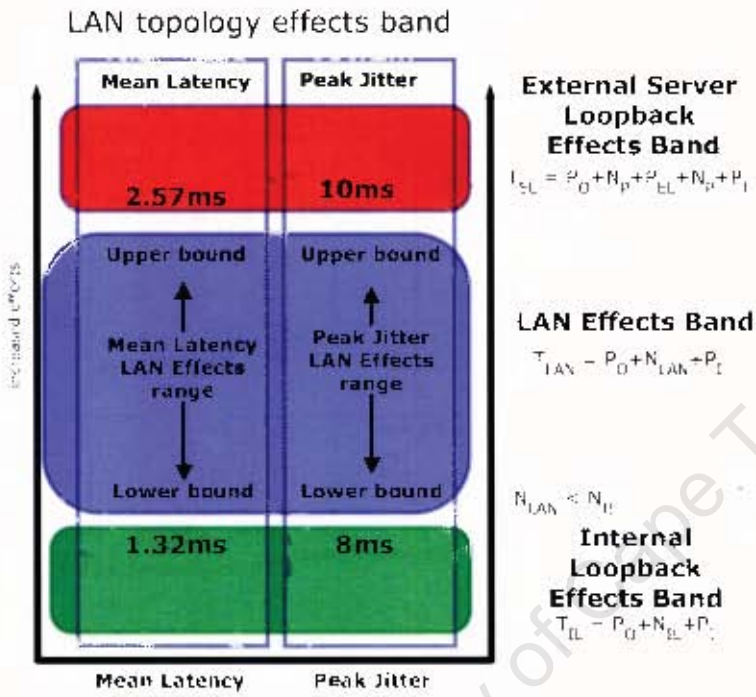

Figure 4.15: Upper and Lower bounds for mean latency and peak jitter

The mean latency range for a LAN network is between 1.32 and 2.57 ms. The upper bound of 2.57ms provides a low latency for musical control and should not compromise musical performances conducted on a LAN network with a single broadcast domain.

The peak jitter range for a LAN network is between 8 and 10ms. The upper bound has not exceeded the benchmark value of 10ms set and will not affect musical performance conducted on a LAN network.

Figure 4.16 is extracted from Brandt and Wright[37] and compares MIDI interface technologies used in industry. The performance criteria are mean latency and peak jitter. The MIDI interface technologies vary from older legacy MIDI interfaces to the newer types that use USB as the transport mechanism.

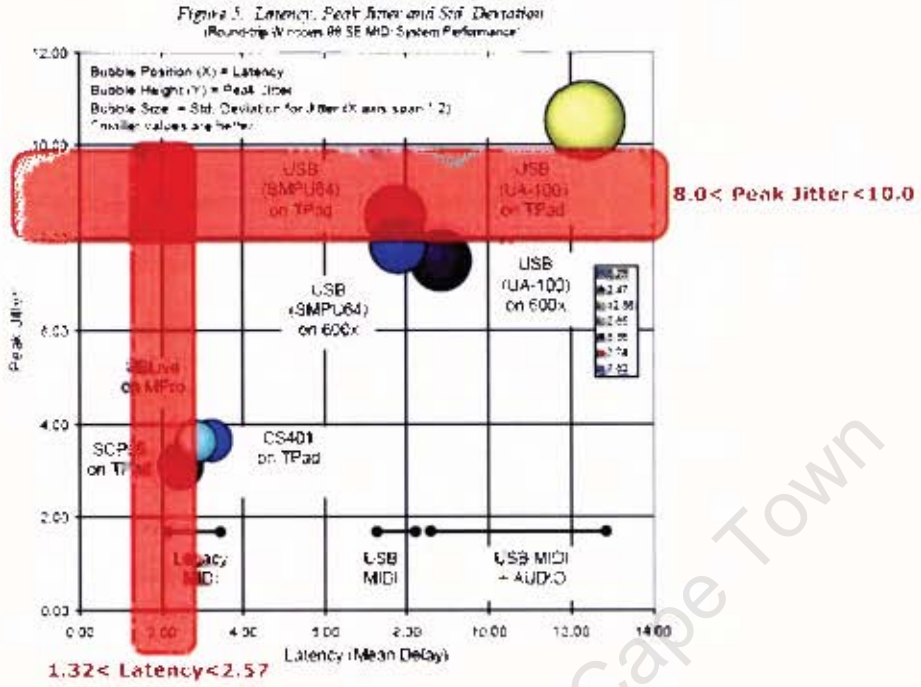

Figure 4.16: Latency and Peak Jittor metrics for MIDI interface technologies

The legacy MIDI devices have a latency range between 2 and 3.5ms and a peak jitter of between 2.5 and 4ms. The newer USB MIDI interfaces have latencies between 7 and 9.5ms and peak jitter range between 7 and 9ms. The USB MIDI interfaces that carry audio as well have 11.5 and 13ms and peak jitter between 9.5 and 11.5ms.

The areas illustrated in red in Figure 4.16 show where the prototype metrics range sits compared to the MIDI interface technologies. The latency range for the prototype has been established as being between 1.32 and 2.57ms for a LAN network.. This latency is lower than all those MIDI interface technologies compared in Brandt and Wright[37].

The peak jitter for the prototype has not faired that well against the MIDI interfaces. The legacy MIDI interface peak jitter is much lower than the prototype. The prototype peak jitter fits in between the USB MIDI interfaces and the USB MIDI interfaces carrying audio data and while not the ideal peak jitter, it is acceptable for musical control.

4.2.4 Experiment 3:

Parameter Scheduling and Packet Loss measure using internal loopback interface

4.2.4.1 Aim

This experiment measures the parameter scheduling[3] of the prototype using the internal loopback network interface. It also measures the packet loss incurred by the prototype. For this test the same data set is used for both metrics of scheduling and packet loss.

The scheduling aims to measure the preservation in the order of packets received compared to those transmitted. The ID argument in the transmitted OSC message is used to determine the scheduling preservation.

The packet loss metric checks whether all the transmitted packets were received at their destination. The ID argument is used to identify packets.

4.2.4.2 Methodology, Setup and Architecture

This experiment was conducted by instantiating two separate instances of the prototype within a VST host. The first instance would perform exclusively as an OSC client transmitting OSC messages onto the internal loopback interface. The second instance would act as a server to the internal loopback interface receiving OSC messages transmitted by the first instance. Figure 4.3 illustrates an architectural diagram of the experiment.

OSC messages are generated by interfacing with the prototype GUI altering the parameter slider until about 5000 OSC messages have been generated.

Within the first instance of the plugin where OSC messages are transmitted, the ID of the message is logged with the timestamp to identify packets.

In the second instance of the OSC server receiving packets, the timestamp together with the ID is logged.

The scheduling is then measured by comparing the order of transmitted packets using the ID to the order of received packets in the destination using the ID in the OSC packet. The unordered packets will be expressed as a percentage of the overall packets transmitted.

The packet loss is to determine which packets, that have been transmitted, have not been received by the destination. Those packet not received will be expressed as a percentage of the overall packets transmitted.

A packet loss will be considered as a packet mis-order as there will be no corresponding packet to match. Therefore the scheduling will either be equal or of a higher value than the packet loss metric. Also if all the scheduling ordering is perfectly complete at 100% then inherently the packet loss metric too is 100% as all the packets are accounted for.

Metric_1 order of packets / scheduling

[3] order of arriving of packets

Metric_2 packet loss

4.2.4.3 Result Data and Findings

METRIC 1: SCHEDULING

Result Data

The results for the scheduling metric are found in the Table 4.6.

Sample Size	ordered packets	unordered packets	% packet unordered
5071	5071	0	0

Table 4.6: Experiment 3 result data for scheduling metric

Findings

The lower bound scheduling provided by the internal loopback network setup with a sample data set of 5071 packets is 100%. This success rate is expected as no packets are sent onto any physical network and only traverse the protocol stack of the NIC. This test concludes that the NIC and the prototype input and output processing preserves the scheduling.

METRIC 2: PACKET LOSS

Result Data

The results for the packet loss metric can be implicitly deduced from the scheduling metric in Table 4.7.

Sample Size	packet not lost	packets lost	% packets lost
5071	5071	0	0

Table 4.7: Experiment 3 result data for packet loss metric

Findings

The lower bound packet loss provided by the internal loopback network setup with a sample data set of 5071 packets is 0% as deduced from the scheduling result data. Therefore no packets were lost.

4.2.5 Experiment 4:

Parameter Scheduling and packet loss measure using external loopback server

4.2.5.1 Aim

This experiment measures the parameter scheduling[4] of the prototype using the external loopback server. It also measures the packet loss incurred by the prototype. For this test the same data set is used for both metrics of scheduling and packet loss.

The scheduling aims to measure the preservation in the order of packets received compared to those transmitted. The ID argument in the transmitted OSC message is used to determine the scheduling preservation.

The packet loss metric checks whether all the transmitted packets were received at their destination. The ID argument is used to identify packets.

4.2.5.2 Methodology, Setup and Architecture

This experiment was conducted by instantiating two separate instances of the prototype within a VST host. The first instance would perform exclusively as an OSC client transmitting OSC messages onto the external loopback server. The second instance would act as a server to the external loopback server receiving OSC messages transmitted by the first instance. Figure 4.5 illustrates an architectural diagram of the experiment.

The ID logging points, methodologies and metrics are the same as in experiment 3.

Metric_1 order of packets / scheduling

Metric_2 packet loss

4.2.5.3 Result Data and Findings

Metric 1: Scheduling

Result Data

The results for the scheduling metric are found in the Table 4.8.

Sample Size	ordered packets	unordered packets	% packet unordered
5155	5155	0	0

Table 4.8: Experiment 4 result data for scheduling metric

[4]order of arriving of packets

Findings

The upper bound scheduling provided by the external loopback server network setup with a sample data set of 5155 packets is 100%. This test concludes that the prototype input and output processing, physical network and external loopback server preserves the scheduling.

Metric 2: Packet Loss

Result Data

The results for the packet loss metric can be implicitly deduced from the scheduling metric in Figure 4.9.

Sample Size	packets not lost	packets lost	% packets lost
5155	5155	0	0

Table 4.9: Experiment 4 result data for packet loss metric

Findings

The upper bound packet loss provided by the internal loopback network setup with a sample data set of 5155 packets is 0% as deduced form the scheduling result data. Therefore no packets are lost.

4.2.6 Discussion and Conclusion

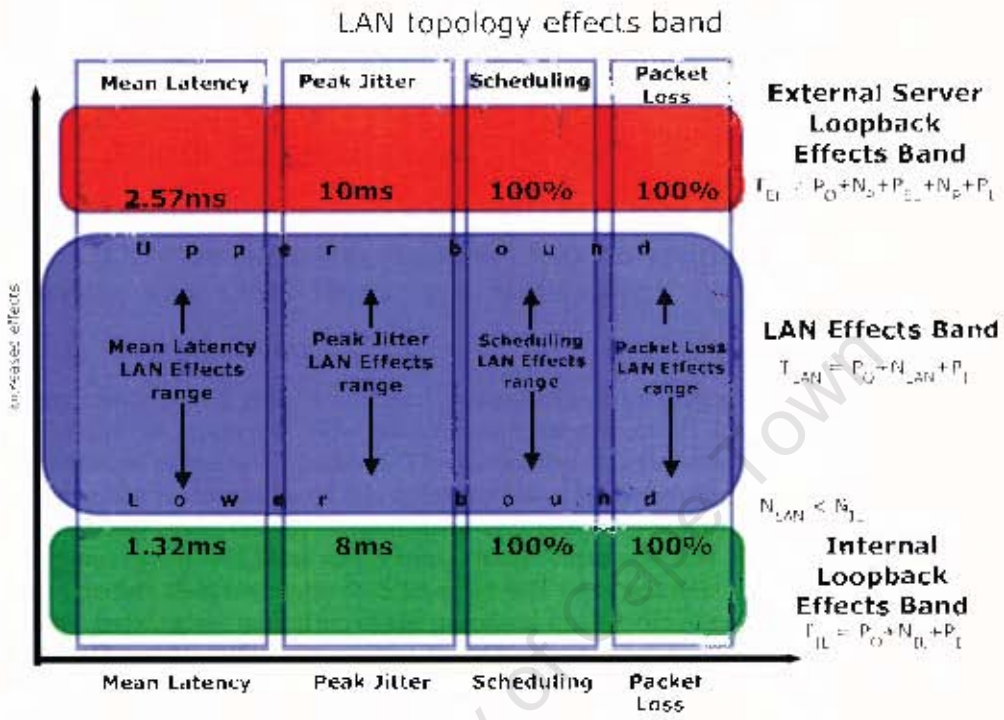

Figure 4.17: Upper and Lower bounds for mean latency, peak jitter, scheduling and packet loss

The parameter data type performance evaluation has been conducted through 4 experiments measuring the metrics of mean latency, peak jitter, scheduling and packet loss. An upper and lower bound has been established for each LAN-type network with a single broadcast domain. Of particular importance is the upper bound which determines the worst case performance for the prototype parameter data type. For the mean latency and peak jitter metrics a benchmark of 10ms was set for each. This benchmark is a strict requirement placed on the metric as musical control and performance requires less strict benchmarks as acceptable performance. However with these strict benchmarks the mean latency of the parameter data type has an upper bound of 2.57ms and is well within the benchmark level of 10ms set. Latency therefore does not degrade the performance of the prototype.

The peak jitter upper bound was found to meet exactly but not exceed the benchmark set of 10ms. When compared to MIDI interfaces the prototype performance with regards to peak jitter faired relatively well outperforming the newer USB MIDI type interfaces but not the legacy older interfaces. For musical control this peak jitter value of 10ms is acceptable.

The scheduling and packet loss metrics attained 100% in that all packets received were in the same order as transmitted and that there were no packets lost out of a large sample of about 5000 packets. The upper and lower bounds achieved 100% for both metrics and therefore the LAN network performance should attain the same.

4.3 Audio Experiments

Comparison on audio QOS benchmarks for 3 different QOS control mechanisms [no control, AudioReceiver, and OSC Bundle Scheduling]

4.3.1 Introduction

Experiments 5 to 7 aim to test and evaluate the audio data type within the OscVstBridge prototype. The audio data type transports uncompressed audio between prototype instances. The remaining experiments therefore aim to measure the performance of this transportation by measuring metrics such as packet order preservation, latency, jitter and temporal fidelity. The packet order preservation and jitter metrics are vitally important and if their result set is not perfect then the audio data integrity will be compromised. The temporal fidelity metric gives indication of the perceived quality of the received audio and will verify packet order preservation and jitter not attaining perfection.

The OscVstBridge prototype contains three control mechanisms that interface the incoming network data in the receiver to the VST host. These control mechanisms are evaluated independently in the experiments and compared to one another.

Audio receiver control mechanisms

The VST host controls the flow of audio data to and from the VST plugin. The plugin processes requests of the VST host and has no control over the size of the audio buffer received or those it has to return to the host. When the OscVstBridge prototype plugin receives audio data from the network it has to place that data in a holding area until the host requests it. The holding area will receive audio data that is unordered and contains jitter but will need to return this data to the host when requested without any jitter and ordered. A process therefore needs to be applied to this holding area. Three separate processes were deployed in the OscVstBridge prototype and are categorised as the audio receiving control mechanisms. These control mechanisms are the DumbStore (or no control mechanism), OSC Scheduler and the JitterBuffer [containing an ordering algorithm] as seen in Figure 4.18.

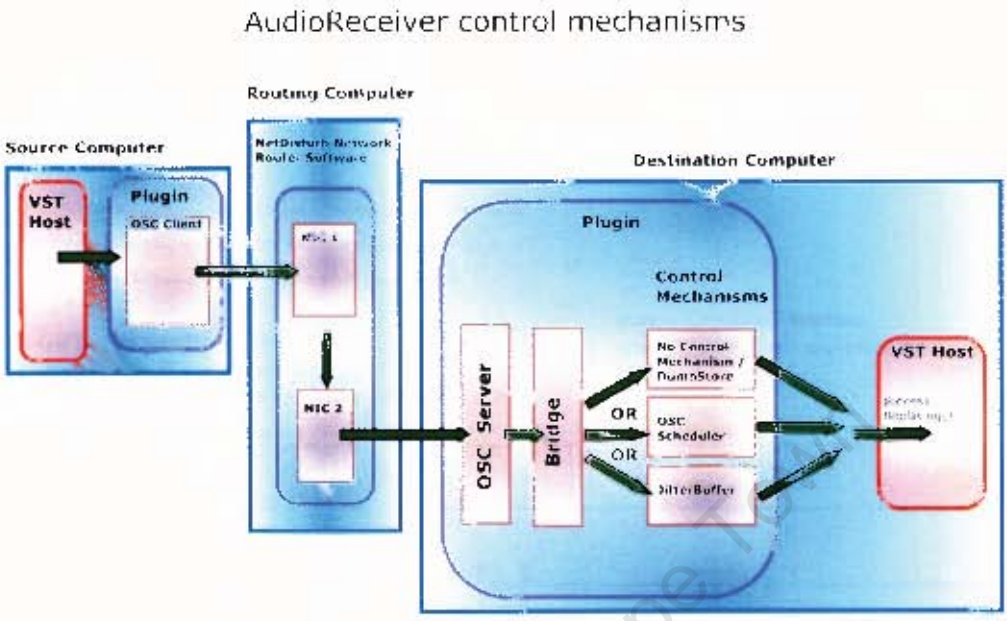

Figure 4.18: AudioReceiver control mechanisms

The DumbStore is simply a buffer that receives audio from the network. If the VST host requests audio data from the plugin it fetches data from this DumbStore. If the VST host requests more data than the DumbStore has in its buffer, the Dumbstore returns what it has and pads with zeros to fulfill the request. There is no management of jitter or order preservation of audio data.

The OSC Scheduler deploys a timing mechanism to preserve packet order and remove network jitter. When OSC packets are transmitted as OSC bundles a scheduled execution time [schedTime] is transported in the OSC bundle header. The OscVstBridge also transported an extra argument in the OSC message that logged the source time [srcTime] at which the message was generated in the OSC client. When packets arrive at the OSC server, the network latency [NetTime] is calculated by subtracting the source time from the arrival time [nowTime]. Packets are then scheduled for execution calculating a 'wait' time [Wait] by subtracting the network latency from the scheduled execution time. This mechanism ensures that packet execution is not adversely affected by network latency and jitter since the 'wait' time compensates for this. This is illustrated in Figure 4.19.

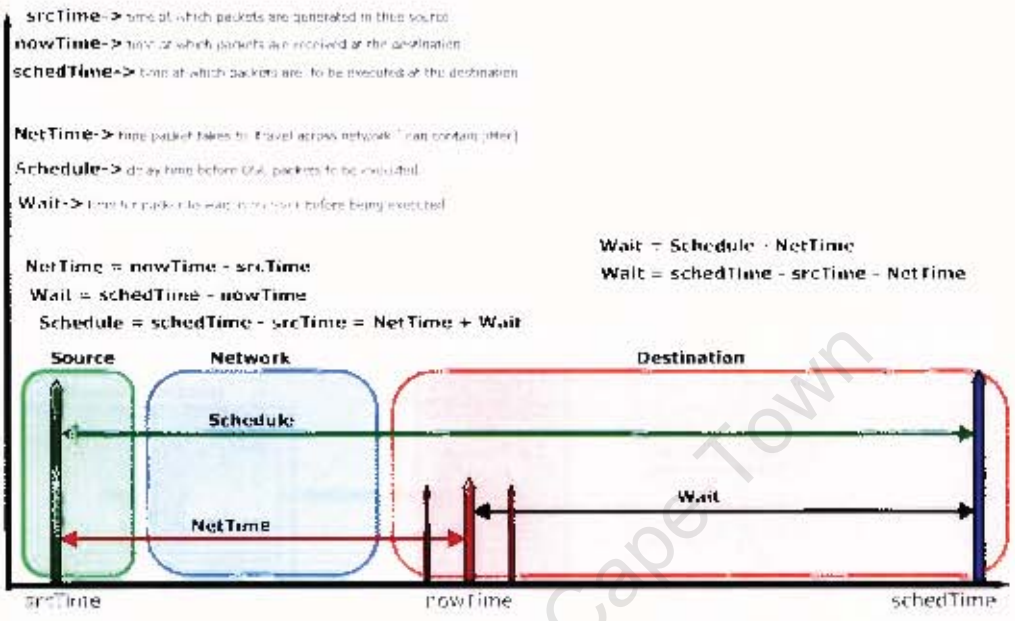

Figure 4.19: OSC Scheduling Illustration

The last control mechanism is the JitterBuffer that includes a packet ordering algorithm. When packets are received from the network, they are firstly ordered by the order algorithm before being placed in a JitterBuffer. The JitterBuffer buffers the data received from the order algorithm and removes jitter. VST host requests for audio data are met by removing data from this buffer.

Metrics

The metrics tested in the audio experiments are scheduling, latency, jitter and temporal fidelity. The scheduling and jitter metrics are most important since they can degrade the content of the audio data. Scheduling measures the preservation of the order of the packets at the receiver as compared to those transmitted. This metric has no margins for error as a packet not ordered degrades the musical content significantly.

The latency metric measures the delay from the time the audio data is available to the transmitting OscVstBridge plugin to the time the same data passes to the VST host from the receiving OscVstBridge plugin. The latency therefore has components of the network delay and the audio receiver control mechanisms.

The jitter metric is derived from the latency metric and is the variation experienced in the latency. The jitter is important as it can degrade and possibly

destroy the musical or audio content.

Temporal fidelity is the perceived quality of the audio material and can be classified as a subjective measure. The experiment has attempted to transform this subjective metric into a quantitative objective one. It therefore measured glitch counts in audio data and analyses spectrum data.

Network Description

The network used in the audio experiment can be found in Figure 4.20.

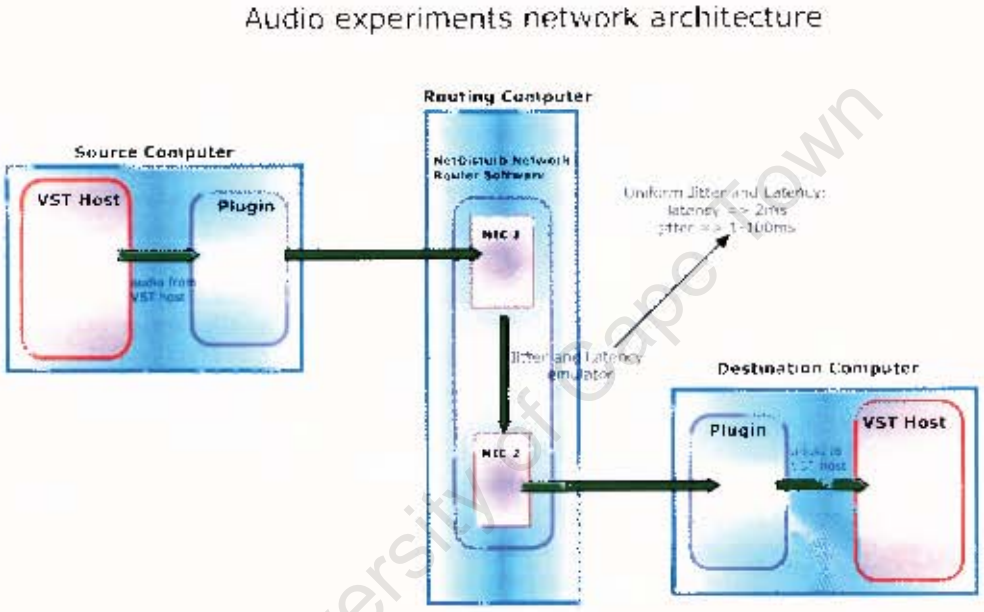

Figure 4.20: Audio Experiments Network Architecture

Audio data is transported between two VST hosts running on separate computers. Each VST host has a single instance of the OscVstBridge prototype plugin instantiated from which audio data is transported over OSC. The routing computer has two NICs and the routing software NetDisturb installed. The routing software passes packets between the two NICs but applies some network effect to the transported packets. For the experiments a fixed latency of 2ms and a jitter range between 1 and 100 ms was set up. Although this jitter is somewhat excessive, it would truly test the capabilities of the audio receiving control mechanisms. A single uncompressed audio channel was used during the audio experiments utilising network bandwidth of about 1.6Mbits per second.

Sources of error

The VST host does not request audio data from the VST plugin in a synchronous manner. The synchronicity of the audio data is dependent on the period between host requests to the plugin and the size of the audio data requested in each request. The VST host requests audio data from the VST plugin through the method call processReplacing(). This method passes an audio buffer array[5] to the VST plugin and receives an audio buffer array of the same size. The size of each buffer in each call is controlled by the host and can vary but has been found not to exceed 197. The time between processReplacing() method calls is approximately 2ms but can vary. Through a series of exploratory experiments a relationship between the sound card latency[6] on the computer running the VST host and the time between certain processReplacing() calls has been established.

For a sound card or audio interface latency of 50ms the standard audio driver uses a buffer size of 2240 samples when the sampling rate is 44100Hz. The processReplacing() calls that are 2ms apart have the maximum buffer size of 197 for 12 processReplacing calls. 197 * 12 = 2364. The last processReplacing() call buffer size is 73 instead of 197. This totals the 2240 buffer size and satisfies the audio interface buffer requirements. The VST host has therefore met the audio interface requirements in 12 calls * 2ms = 24ms and does not need to make any more calls to the plugin for data till 50ms - 24ms = 26ms. The next processReplacing() call therefore is only made after 26ms. This can be seen in Figure 4.21 demonstrated that the processReplacing() calls by the host are not synchronous.

An audio interface latency of 2ms at 44100Hz has a buffer size of 128 samples. Therefore the VST host processReplacing() calls to the plugin cannot have a buffer size request of more than 128 samples. There is no period between processReplacing() calls greater than 2ms as the audio interface requirements are just met by the VST host before the next request comes along.

The VST host requests to the VST plugin through the processReplacing() method calls are therefore not synchronous and will introduce error into the data for the audio experiments conducted, particularly experiment 6 that measures latency and jitter. The latency metric will be most affected but has little effect on the jitter metric which is the conclusive metric in the experiment. It is therefore adequate to include this source of error into the result data set.

[5] depending on the number of audio channels the plugin utilises

[6] or bufferSize

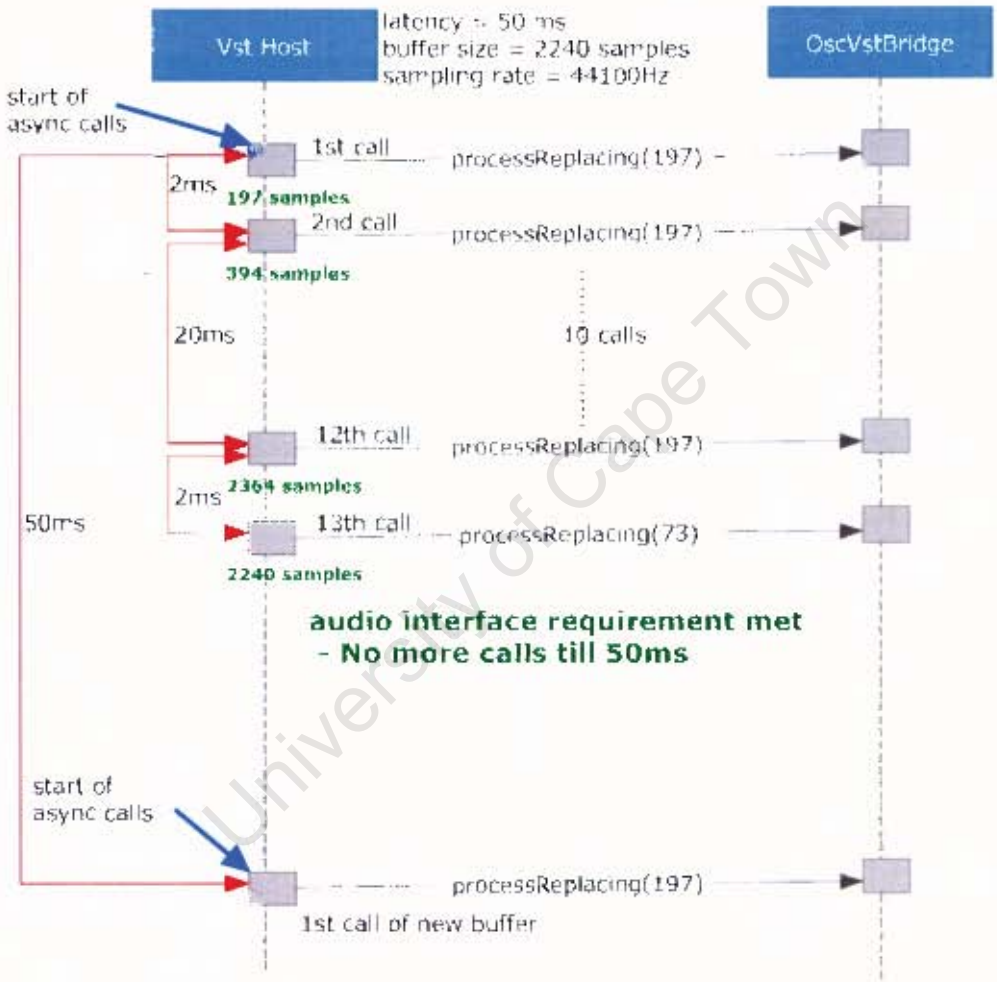

Figure 4.21: Asynchronous calling sequence by the VST host

The OscVstBridge development could not acquire clock sources from the VST host due to the asynchronous nature of the audio data calls in processReplacing(). The soundcards in both the source and destination computers have a sampling rate of 44100Hz implying that the data rate of the source soundcard and VST host sample requests to plugins equals that in the destination

computer. Therefore there is no net gain or loss of data through the network. In a well managed jitter buffer underflow or overflow would not occur in these conditions. Figure 4.22 illustrated this.

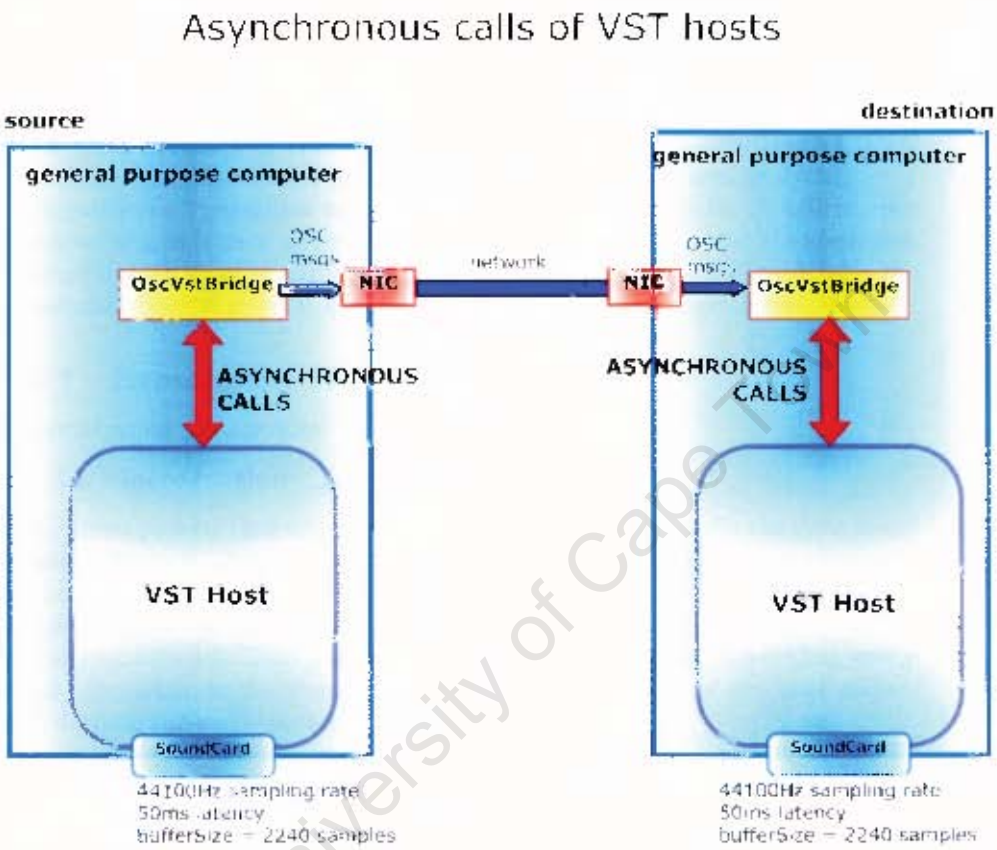

Figure 4.22: Asynchronous calls of VST hosts

Outline of audio experiments

Table 4.10 indicates the structure of the experiments to be conducted on the Audio data type.

Metric	Scheduling	Latency	Jitter	Temporal Fidelity
no control mechanism	Exp 5	Exp 6	Exp 6	Exp 7
OSC Scheduling	Exp 5	Exp 6	Exp 6	Exp 7
JitterBuffer	Exp 5	Exp 6	Exp 6	Exp 7

Table 4.10: Audio experiments and metrics

The audio experiments begin with experiment 5 measuring the packet order preservation or scheduling within audio data. The experiment gathers and analyses data for no control mechanism, OSC Scheduling and the JitterBuffer control mechanisms. Experiment 5 uses the count attribute in the OscAudio objects received from the network to measure packet order preservation.

Experiment 6 follows measuring latency of the audio data. The variance of the latency yields the jitter metric. A control audio channel is set up transporting audio over the physical audio interfaces of the two networked computers used. The latencies of the audio interfaces are included. No control, OSC Scheduling and JitterBuffer control mechanisms are tested and analysed.

Experiment 7 measures temporal fidelity in the audio data. A 440Hz sinusoidal waveform is used and audio glitches like 'clicks' and 'pops' are measured. Spectrum analysis of the received audio signal is done in order to quantify the preservation of the original audio content for each of the control mechanisms.

4.3.2 Experiment 5:

SCHEDULING MEASURES

4.3.2.1 Introduction

OSC audio packets that are transmitted into a network contain the data found in each attribute of an OscAudio object. This includes the count attribute. Packets transmitted are incrementally ordered but received packets may not be ordered due to the non-preservation of packet order that is introduced by the underlying network. This packet order preservation is called scheduling and is vitally important in the preservation of audio content. Out of sequence packets will degrade the audio content.

4.3.2.2 Aim

This experiment measures the scheduling of the OSC audio packets received from the network. A uniform jitter range between 1-100ms is introduced by the NetDisturb routing software. Packets received will therefore not exhibit scheduling because of the jitter introduced. The control mechanisms are called into action to preserve packet order before the audio data is passed to the VST host. This experiment firstly measures the scheduling for no control mechanism, and then for each of the OSC Scheduler and the JitterBuffer.

It is envisaged that the result set will have poor scheduling for no control mechanism and will be directly related to the network jitter introduced. It is also envisaged that both control mechanisms will incrementally order the unordered packets received from the network thereby perfectly preserving the packet order and the audio content.

4.3.2.3 Methodology, Setup and Architecture

The architecture of the experiment used is found in Figure 4.23.

Figure 4.23: Scheduling experiment network architecture

The count attribute in the OscAudio object is incrementally increased in the OSC client of the source computer. This count is transmitted to the destination computer where it is processed by the control mechanism if any before being logged and passed to the VST host through the processReplacing() method call. The logged received count value is compared to the incremental count value generated in the source computer and graphed accordingly. The source computer count value will always be incrementally increased with perfect packet order. There is therefore no need to log packets in the source computer for the experiment data set. The first count value received will be used as the first count for the incremental source count and all subsequent source count values will be an increment of that. The difference between the received count and the source count is termed as jitter and is graphed as a Probability Density Function of packet number jitter. The distribution of this Probability Density Function is observed and analysed.

The experimental was conducted by generating audio content in the source computer and logging the count values in the destination computer. The sample size is 300. Three runs were conducted for the control mechanisms including no control mechanism. For the OSC Scheduler control mechanism, OSC packets were scheduled to be executed 300ms after being generated in the source computer. This time was determined to be adequate to compensate for the introduced network jitter. The JitterBuffer size used was 10ms.

Metric 1 packet order preservation / scheduling

4.3.2.4 Result Data and Findings

Control Mechanism 1: No control Mechanism

Result Data

The results for the scheduling metric for no control mechanism are found in Figures 4.24, 4.25 and 4.26.

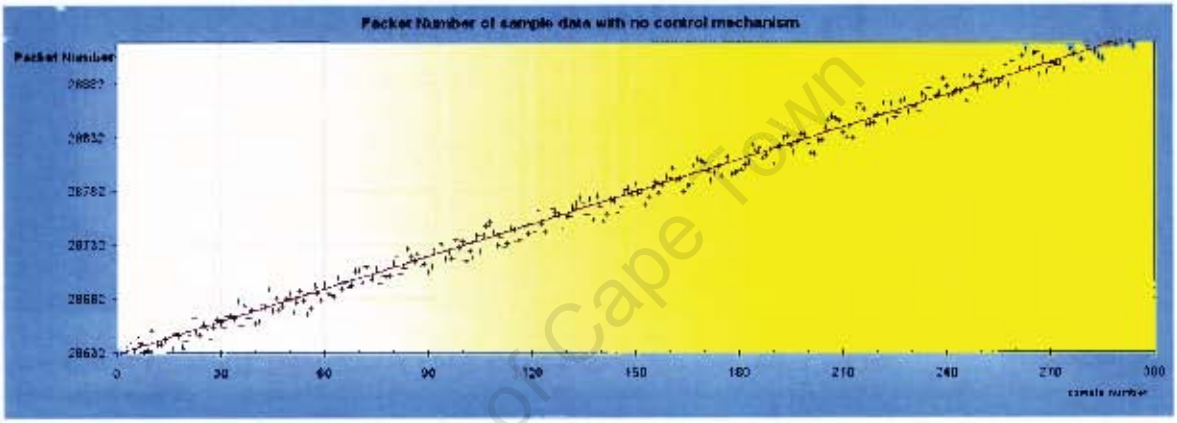

Figure 4.24: Packet Number of Sample Data with no control mechanism

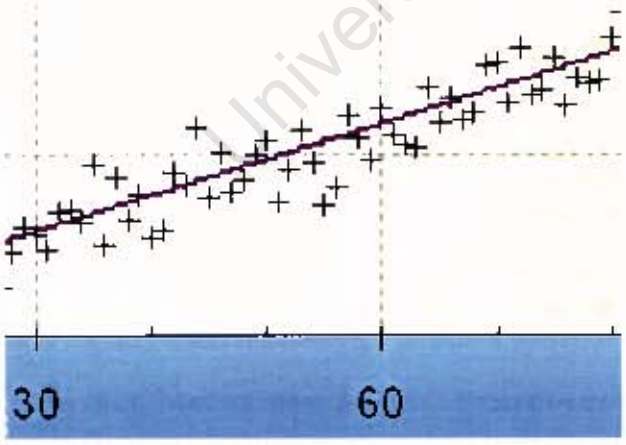

Figure 4.25: Section Blowup of Packet Number of Sample Data with no control mechanism

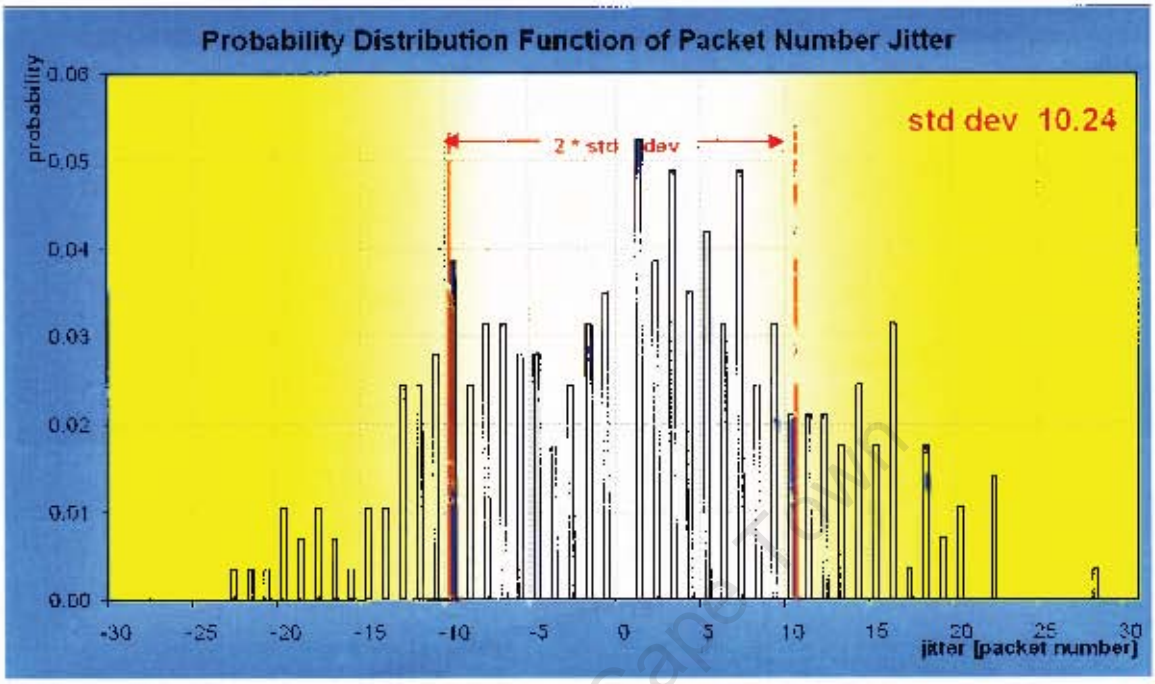

Figure 4.26: Probability Distribution Function of Packet Number Jitter for no control mechanism

From the data above we can see that the count value starts at 28632 continuing for 300 samples. The packet number count from the source can be seen as a purple straight line. The received data can be seen to plot itself around this line as plot points [illegible]. The probability distribution graph appears to be similar to a normal distribution with a standard deviation of 10.24.

Findings

The data indicates that the count value received from the network does not align to that transmitted into the network. This is evident in that the received plot points do not accurately follow the source data count in the purple line in Figures 4.24 and 4.25. Packet order preservation is therefore not evident and hence the audio data content will not be preserved. The probability density function graph in Figure 4.26 shows the spread of the received count number from the source count number. The standard deviation is 10.24.

Control Mechanism 2: OSC Scheduling

Result Data

The results for the scheduling metric for the OSC Scheduling control mechanism are found in Figures 4.27, 4.28 and 4.29.

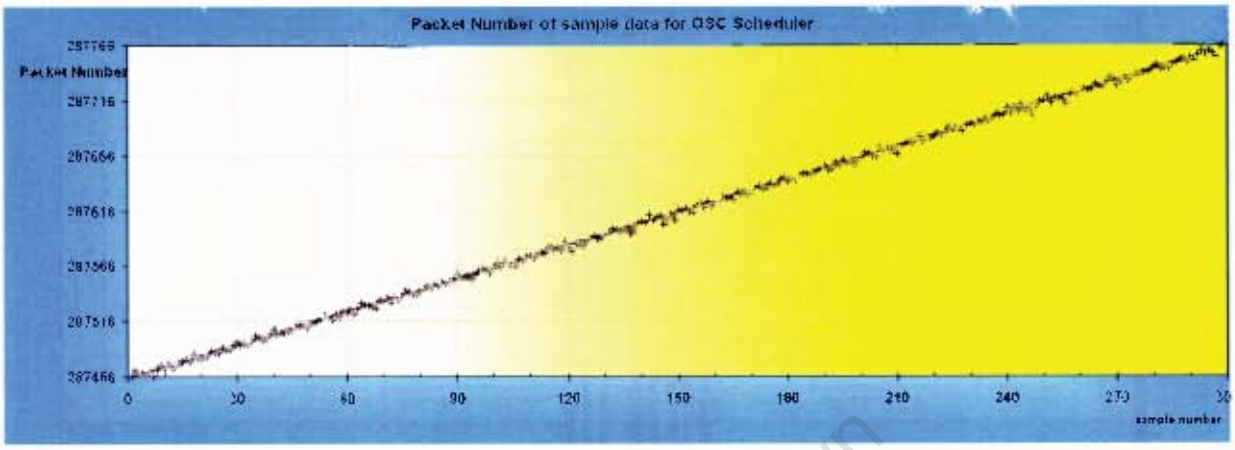

Figure 4.27: Packet Number of Sample Data with OSC Scheduler

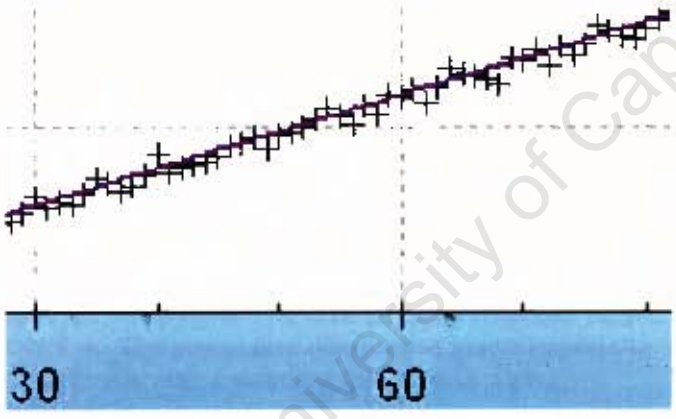

Figure 4.28: Section Blowup of Packet Number of Sample Data with OSC Scheduler

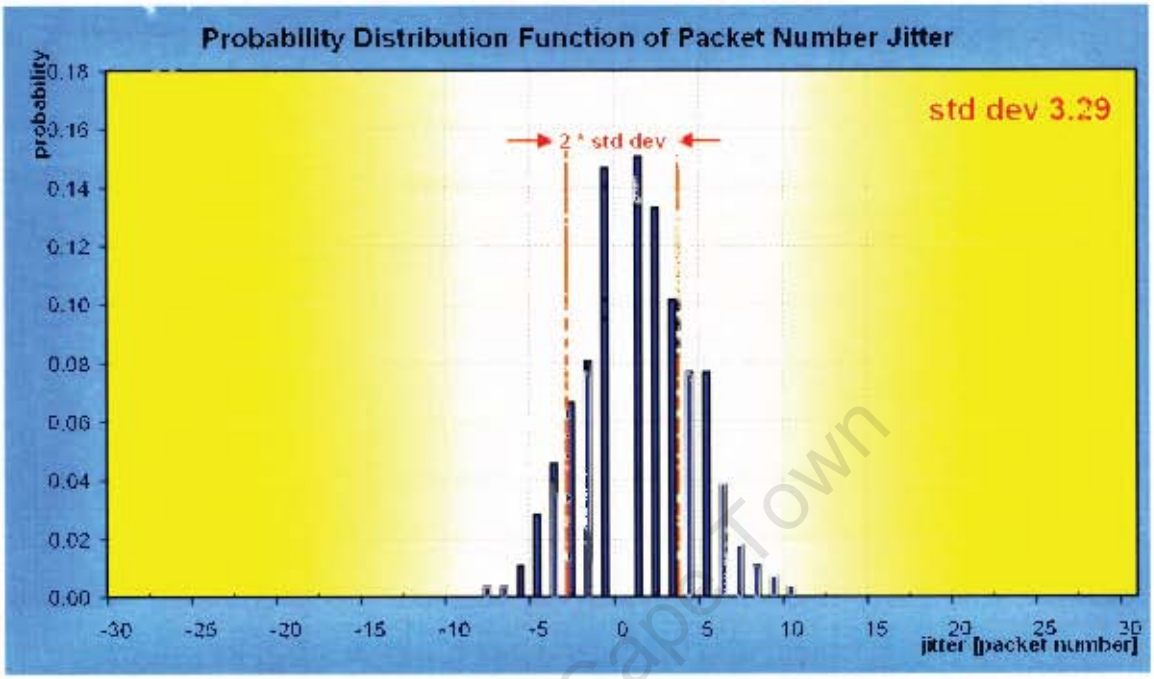

Figure 4.29: Probability Distribution Function of Packet Number Jitter for OSC Scheduler

From the data above we can see that the count value starts at 287466 continuing for 300 samples. The packet number count from the source can be seen as a purple straight line. The received data can be seen to plot itself around this line. The probability distribution graph appears to be similar to a normal distribution with a standard deviation of 3.29.

Findings

The data indicates that the count value received from the network does not align perfectly to that transmitted into the network. This is evident in that the received plot points do not perfectly follow the source data count in the purple line in Figures 4.27 and 4.28. However it is a better and closer match than when compared to the data for no control mechanism. Exact packet order preservation is not achieved and hence the audio data content will not be preserved. The probability density function graph in Figure 4.29 shows the spread of the received count number from the source count number. The standard deviation is 3.29. The change in standard deviation from 10.24 to 3.29 when moving from no control mechanism to OSC Scheduling verifies that the packet order preservation has improved but has not achieved complete scheduling.

Control Mechanism 3: JitterBuffer

Result Data

The results for the scheduling metric for the JitterBuffer control mechanism are found in Figures 4.30, 4.31 and 4.32.

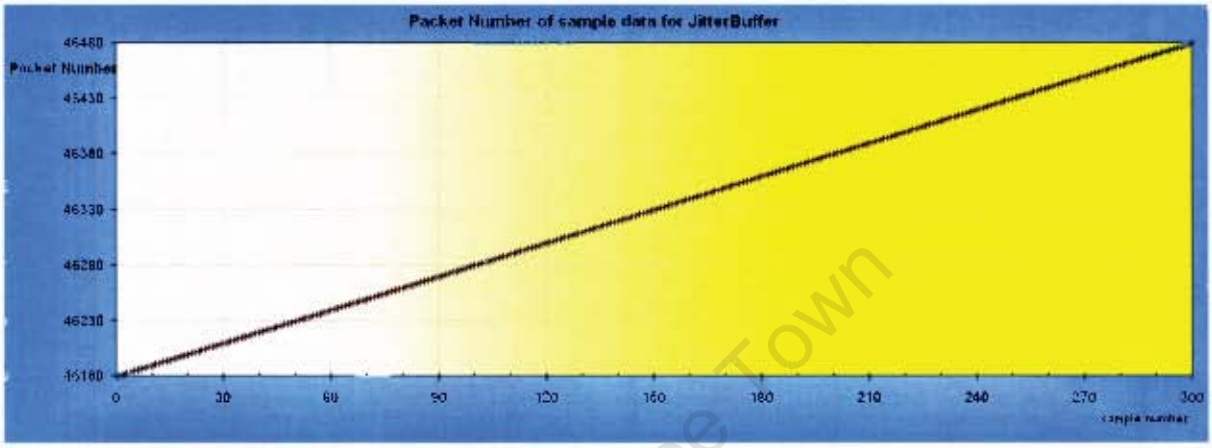

Figure 4.30: Packet Number of Sample Data with JitterBuffer

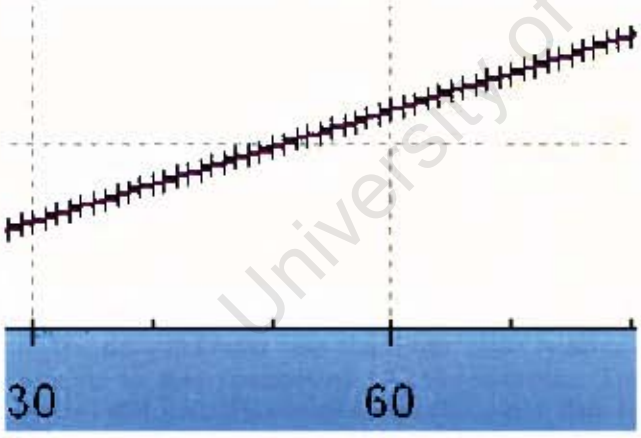

Figure 4.31: Section Blowup of Packet Number of Sample Data with JitterBuffer

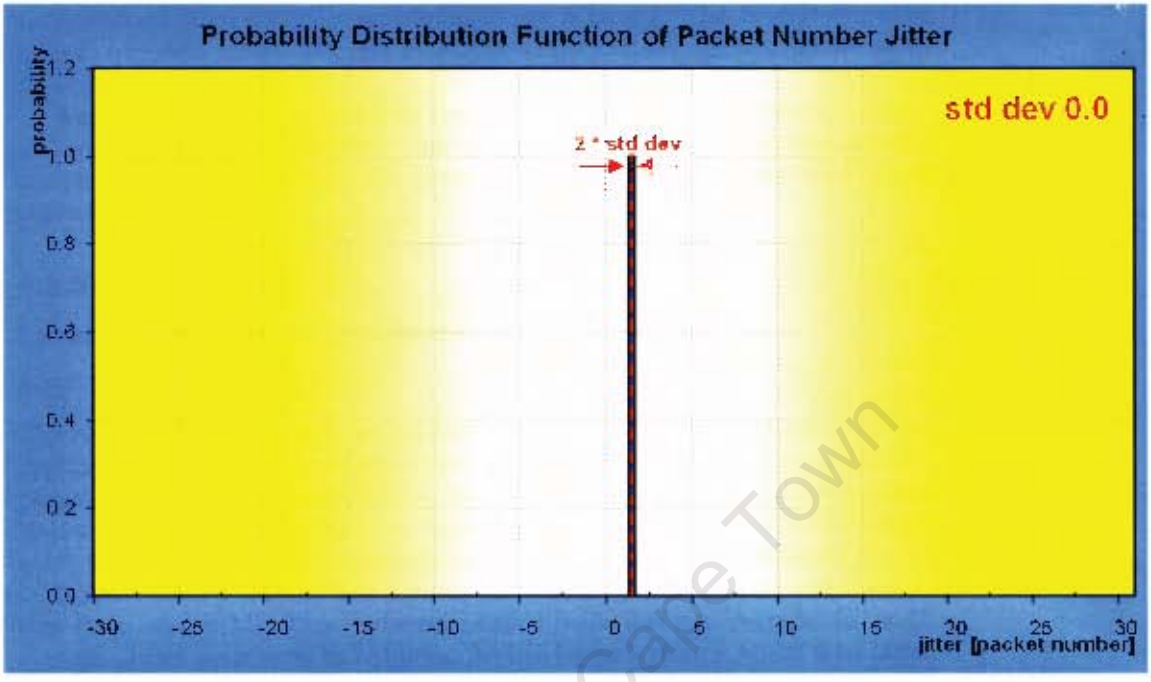

Figure 4.32: Probability Distribution Function of Packet Number Jitter for JitterBuffer

From the data above we can see that the count value starts at 46180 continuing for 300 samples. The packet number count from the source can be seen as a purple straight line. The received data can be seen to plot itself exactly on this line. The probability distribution graph has all of its distribution at count value of 1 with a standard deviation of 0.0. This indicates that the received count data is incrementally increased

Findings

The data indicates that the count value received from the network aligns perfectly to that transmitted into the network. This is evident in that the received plot points perfectly follow the source data count in the purple line in Figures 4.30 and 4.31. Exact packet order preservation is achieved and hence the audio data content will be preserved. The probability density function graph in Figure 4.32 shows that there is no spread of the received count number from the source count number. The standard deviation is therefore 0.0. Complete scheduling is achieved.

When comparing the three control mechanisms, no control mechanism has the worst performing data with a standard deviation of 10.24 in the jitter of the scheduling metric. The OSC Scheduling metric improves on this standard deviation bringing its data's standard deviation for jitter of the scheduling metric down to 3.29. This reduction however does not contribute to the preservation of

audio content since complete scheduling is not achieved. The JitterBuffer does achieve complete scheduling and preserves audio content across a jitter-ridden network.

Apart from the OSC Scheduling control mechanism the result data has been anticipated. It was anticipated that the OSC Scheduler would also achieve complete scheduling but has not proved to be the case. Further work could explore this.

4.3.3 Experiment 6:

Latency and Jitter measures

4.3.3.1 Introduction

The requirements of latency introduced by a computer network depends on the application. Transporting audio content over a network in a radio broadcast application can accommodate relatively large latencies provided that it is constant. Real-time interactive applications such as networked musical collaboration require low latency as large delays can make musical interaction difficult[20].

The variance of latency has a much more devastating effect on musical content than latency[37]. This variance is called jitter and can degrade the audio content. Jitter is reduced in receiving devices by providing a buffer that introduces added latency. A trade-off between jitter and latency therefore exists and can be manipulated to suit the application in which these metrics apply. The OscVstBridge prototype has such a buffer called the JitterBuffer. OscVstBridge also has OSC Scheduling which also attempts to compensate for network jitter by introducing added latency. Experiment 6 performs tests on both of these as well as for no control mechanism.

4.3.3.2 Aim

In this experiment an audio signal that is transported over a jitter-ridden network is measured against a control audio signal that uses the physical audio interfaces or sound card to determine the latency introduced by the network and the control mechanisms. The spread of this latency is the jitter.

This experiment collects data sets for both the OSC Scheduling and JitterBuffer control mechanisms as well as data for no control mechanism. The data sets are analysed and compared to one another.

Audio markers are inserted into a silent audio stream which will be used to measure the latency. The audio markers are audible as clicks consisting of short duration, 3 cycle, normalised square waves.

The experiment anticipates that the control mechanisms will remove the network jitter introduced and hence provide the VST host at the receiver with a jitter free audio signal. The cost of the jitter removal is added latency introduced.

4.3.3.3 Methodology, Setup and Architecture

The architecture of the experiment used is found in Figure 4.33.

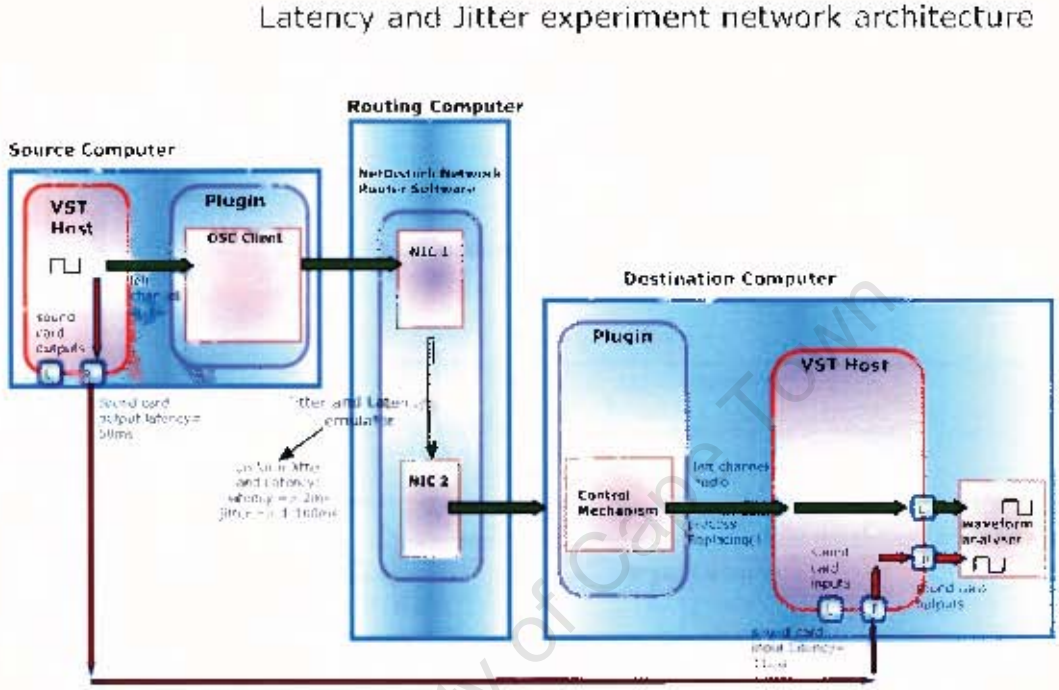

Figure 4.33: Latency and jitter experiment network architecture

A 4410Hz square wave consisting of three cycle waves was used as the audio markers to identify audio segments. The period between each marker was 868ms. The control audio signal traversing the physical audio interfaces through a cable connection is carried on the right audio channel of a stereo system. The sound card output latency of the source computer is 50ms with 2240 samples at 44100Hz sampling rate. The sound card input latency at the destination computer is 11ms with 512 samples at 44100Hz sampling rate. Assuming the cable transmission time is negligible, the total latency introduced by the control path is $50 + 11 = 61$ms.

The left channel transports audio data via the OscVstBridge plugin and the underlying network. The OSC client in the source computer, as well as the input processing in the destination computer, introduces negligible latency and is ignored. The significant latencies are introduced by the network and the control mechanisms.

The VST host in the destination computer receives the right channel control audio via the physical, soundcard, cabled interface and the left channel 'under test' audio via the OscVstBridge plugin instances and the computer network.

The VST host outputs these channels into a waveform analyser in order to determine the offset between the left and right markers in the stereo audio signal. This measured offset is added to 61ms control latency to determine the latency for the OscVstBridge and networked audio transport system as seen in Figure 4.34.

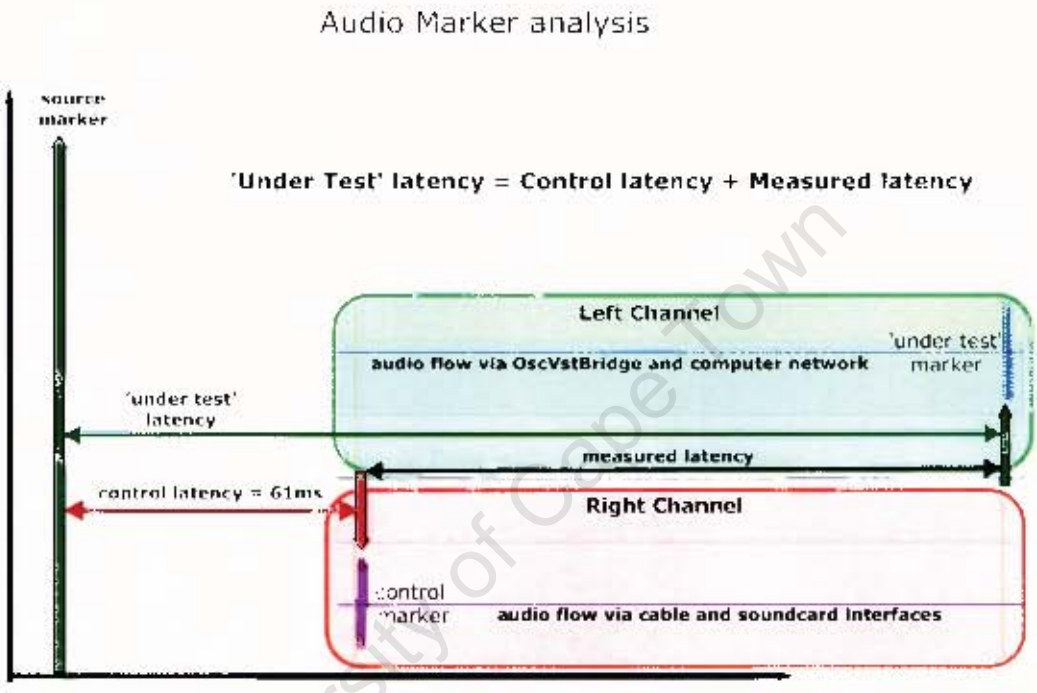

Figure 4.34: Audio Marker analysis

The single sample in the waveform analyser consists a control marker and an 'under test' marker. The total samples for each control mechanism is 100.

The experiment was conducted by preparing the network architecture in Figure 4.33. The audio marker samples were loaded into the source computer's VST host. The output of the destination computer's VST host was recorded and analysed. Data was gathered for both the OSC Scheduler and the JitterBuffer control mechanisms as well as for no control mechanism.

Metric_1 latency

Metric_2 jitter

4.3.3.4 Result Data and Findings

Control Mechanism 1: No control Mechanism

Result Data

The results for the latency and jitter metrics for no control mechanism are found in Figures 4.35 and 4.36.

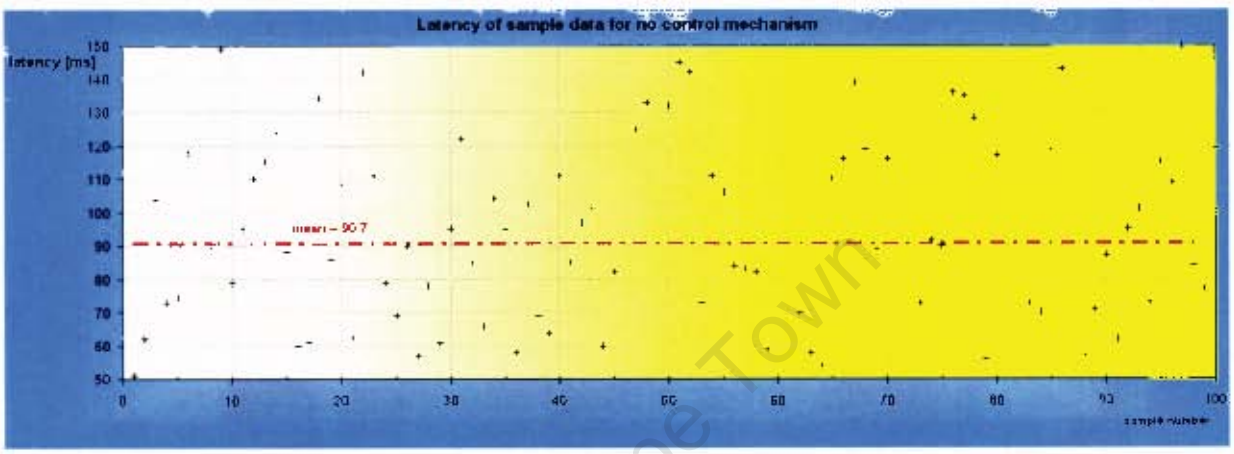

Figure 4.35: Latency of Sample Data with no control mechanism

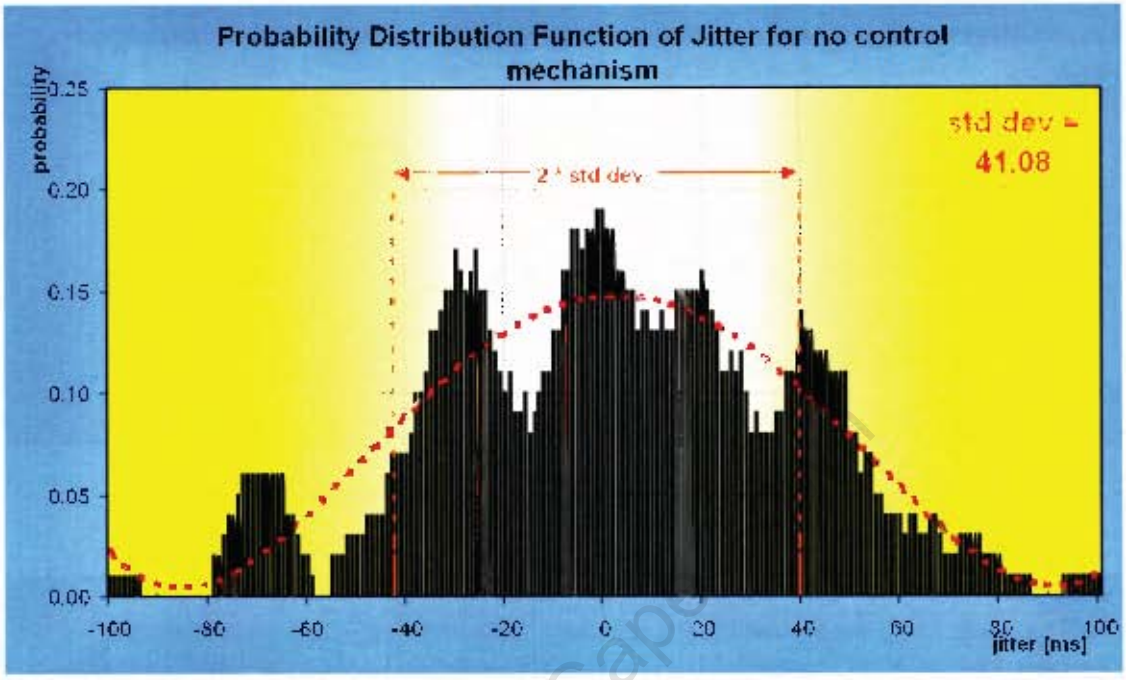

Figure 4.36: Probability Distribution Function of Jitter for no control mechanism

From the data above we can see that the latency varies between 50-150ms. The mean value of this 100 sample data set is 90.7. The probability distribution graph shows the jitter spread and appears to be similar to a normal distribution with a standard deviation of 41.08. A 'best-fit' 4th order polynomial trendline can be seen in red in Figure 4.36.

Findings

The analysis of the data indicates that there is a large variance in the latency of received packets. This is contributed by the jitter introduced by the network. The jitter range in the data set in Figure 4.35 corresponds to the jitter range of the network ie. 1-100ms. The probability density function graph in Figure 4.36 shows the spread of the jitter with a standard deviation is 41.08. The jitter spread is flat and wide with a uniform distribution

Control Mechanism 2: OSC Scheduler

Result Data

The results for the latency and jitter metrics for the OSC Scheduler control mechanism are found in Figures 4.37 and 4.38.

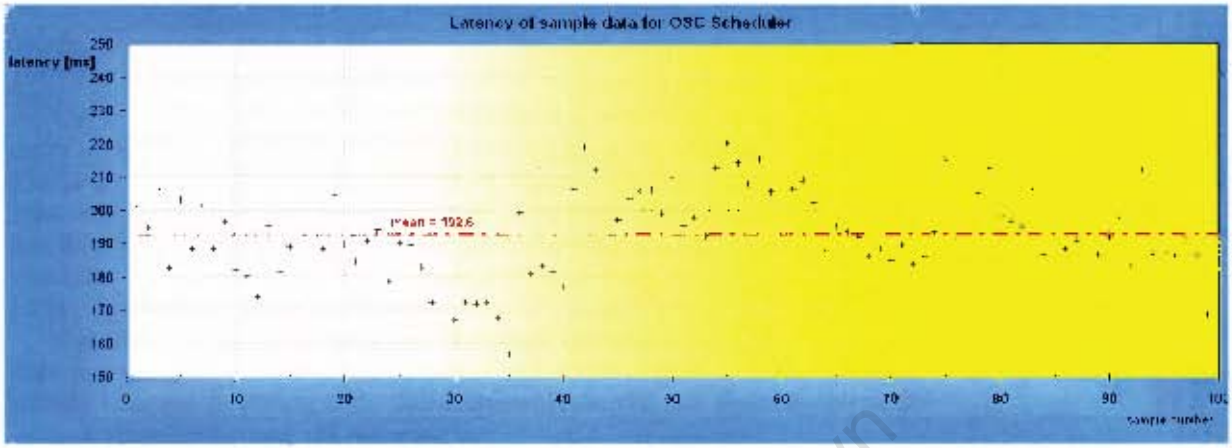

Figure 4.37: Latency of Sample Data for OSC Scheduler

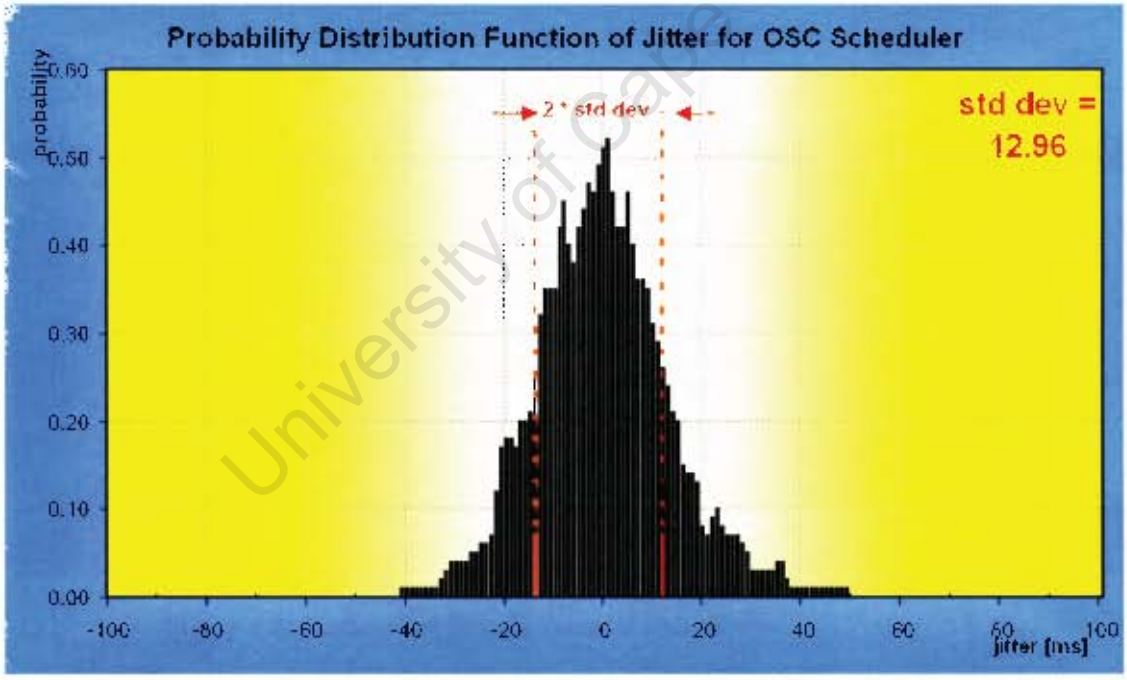

Figure 4.38: Probability Distribution Function of Jitter for OSC Scheduler

From the data above we can see that the latency varies between 160-220ms. The mean value of this 100 sample data set is 192.6. The probability distribution graph shows the jitter spread and appears to be similar to a normal distribution with a standard deviation of 12.96.

Findings

The analysis of the data indicates that there is a reduced variance in the latency of received packets for the OSC Scheduler than for no control mechanism. The probability density function graph in Figure 4.38 shows the spread of the jitter with a standard deviation of 12.96. The jitter spread is uniform and is less flat and narrower than that for no control mechanism. This improved jitter reduction is also evident in the standard deviation being reduced from 41.08 to 12.96.

The OSC Scheduler does reduce the jitter introduced by the network but does not completely remove it as there is still jitter present in the probability density function in Figure 4.38. The reduced jitter will still degrade the audio content transported over the network.

Control Mechanism 3: JitterBuffer

Result Data

The results for the latency and jitter metrics for the JitterBuffer control mechanism are found in Figures 4.39 and 4.40.

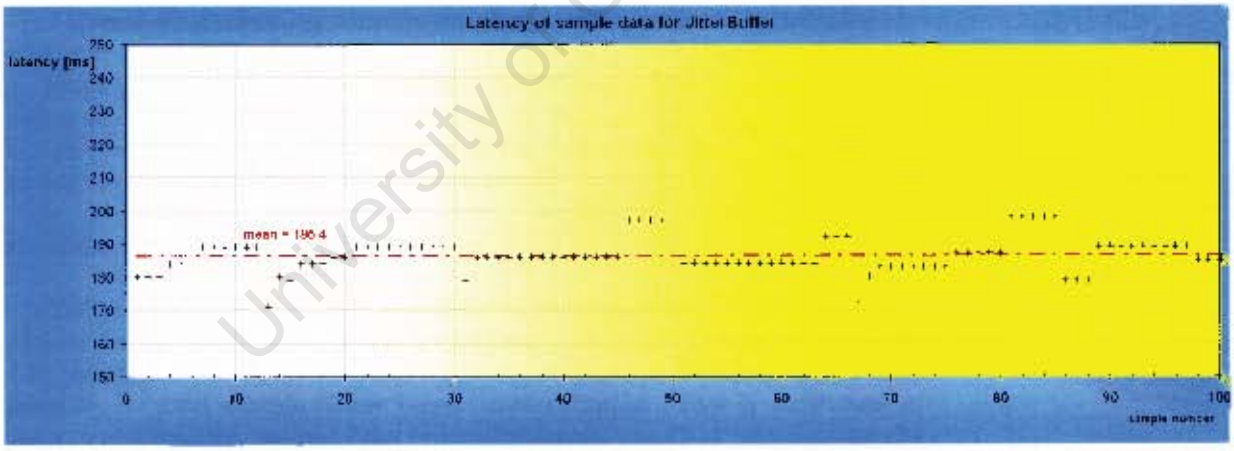

Figure 4.39: Latency of Sample Data for JitterBuffer

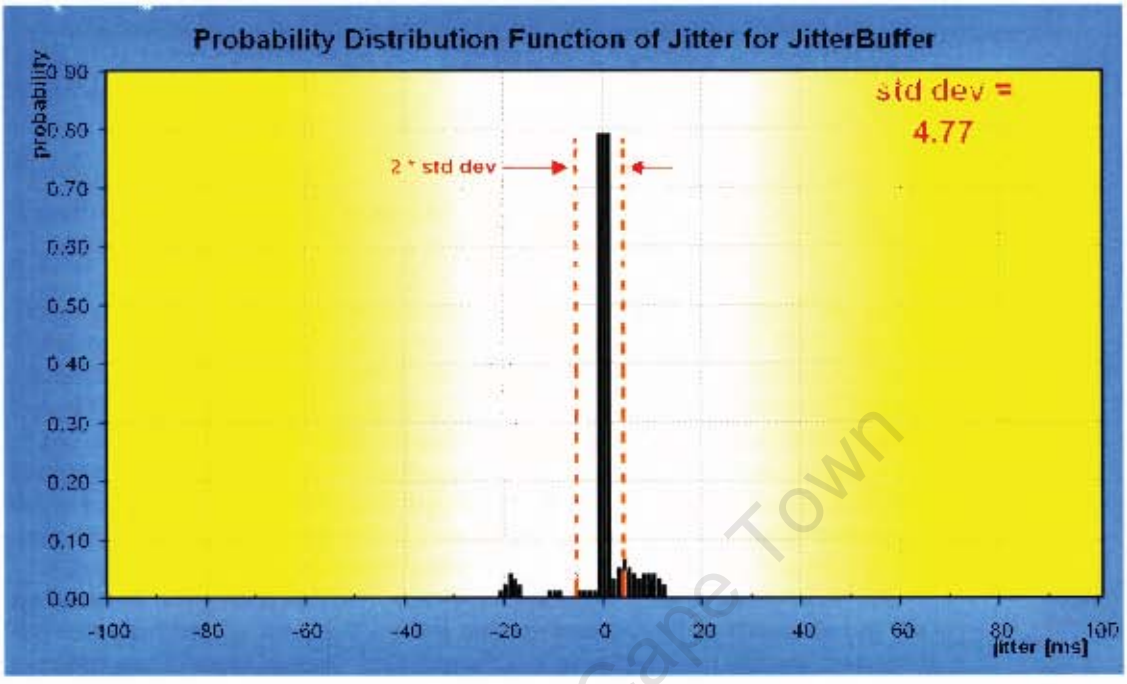

Figure 4.40: Probability Distribution Function of Jitter for JitterBuffer

From the data above we can see that the latency is fairly constant between 180ms and 195ms and with a mean value of 186.4ms. The probability distribution graph shows the jitter spread with the bulk of the sample data (>75%) having no jitter. The standard deviation is 4.77.

Findings

The analysis of the data indicates that the variance in the latency of received packets for the JitterBuffer is nearly completely removed. The graph in Figure 4.39 shows that the sample latencies are constant over a few samples but then jump to the next sample before being constant again. This is attributed to the non-synchronous nature of the processReplacing() method calls done by the VST host. The audio data content is fully preserved as audibly observed in the output audio signal thereby suggesting that a constant latency with no jitter is present. The probability density function graph in Figure 4.40 shows the spread of the jitter with a standard deviation of 4.77 caused by the non-synchronous processReplacing() method calls.

The JitterBuffer completely removes the network jitter preserving the audio signal. The JitterBuffer is therefore the best control mechanism in terms of the latency and jitter metrics and the preservation of an audio signal. The OSC Scheduler reduces the jitter introduced by the network but does not completely remove it as there is still jitter present in its probability density function. It

was anticipated that the OSC Scheduler would also completely remove the network jitter but the results show only a reduction in the jitter and not complete removal.

4.3.4 Experiment 7:

Temporal Fidelity Measures

4.3.4.1 Introduction

Temporal Fidelity is defined in this context as the audible quality of the audio signal received from the jitter-ridden network. Packet order and jitter can degrade the audio signal introducing 'clicks' and 'pops' into the audio stream with loss of the original content. Temporal Fidelity is therefore the perceived quality of the audio stream with high temporal fidelity having no loss of the original audio content with no 'clicks' and 'pops'. It should be noted that temporal fidelity has a subjective evaluation associated with it. This experiment therefore aims to transform those subjective criteria into objective measurable metrics.

Experiments 5 and 6 measured scheduling, jitter and latency metrics on an audio signal using the 3 different control mechanisms. This experiment measures the temporal fidelity for the 3 control mechanisms as well as consolidating the previous audio experiments. This experiment provides the ultimate metric of perceived audio quality that end users will be exposed to.

4.3.4.2 Aim

This experiment is divided into two parts. The first is an analysis at the receiver of the spectral content of the audio signal when a pure sinusoidal waveform is injected at the source. The introduction of additional spectral content to the sinusoid will give indication that 'clicks' and 'pops' are observed in the audio stream. The second part measures the glitches such as clicks and pops in 250ms by observing the audio waveform at the receiver.

Both sections are conducted for each of the control mechanisms. The results are analysed and combined with that of experiments 5 and 6 to produce a conclusive summary for the 3 control mechanisms.

Based on the result data for experiments 5 and 6, the JitterBuffer control mechanism is expected to reproduce the source audio stream perfectly. It is therefore anticipated that its spectral content will only contain the injected sinusoid and no glitches will be observed in the received audio stream. The OSC Scheduler and no control mechanism will display glitches and contain added spectral content.

4.3.4.3 Methodology, Setup and Architecture

The architecture of the experiment used is found in Figure 4.41.

Figure 4.41: Temporal Fidelity experiment network architecture

This experiment's architecture is similar to that of experiment 6 except that the physical cable audio interface path is not present. A continuous sinusoidal waveform of 440Hz is injected as the source audio signal and transported across the jitter network. The injected source audio signal and its spectrum analysis can be seen in Figures 4.42 and 4.43 respectively. Each of the control mechanisms are applied and separate data sets are collected for each at the waveform analyser. A spectrum analysis of the received audio signal is performed using a Blackman-Harris smoothing window and a FFT size of 16 384. If the injected audio signal is fully preserved then the spectrum analysis will yield a 'spike' at 440Hz representing the injected pure sinusoidal waveform. If the audio signal is not preserved, glitches as 'clicks' and 'pops' will introduce new spectral information. 1 second of mono audio data is spectrally analysed in the waveform analyser.

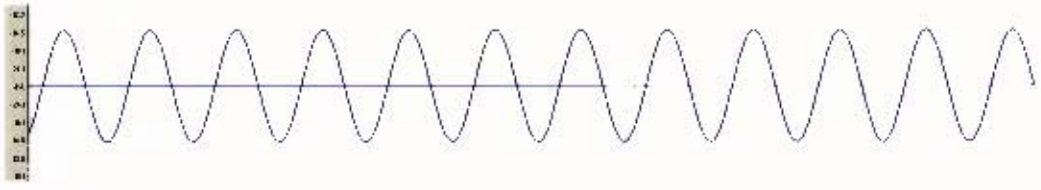

Figure 4.42: Injected 440Hz sinusoidal waveform

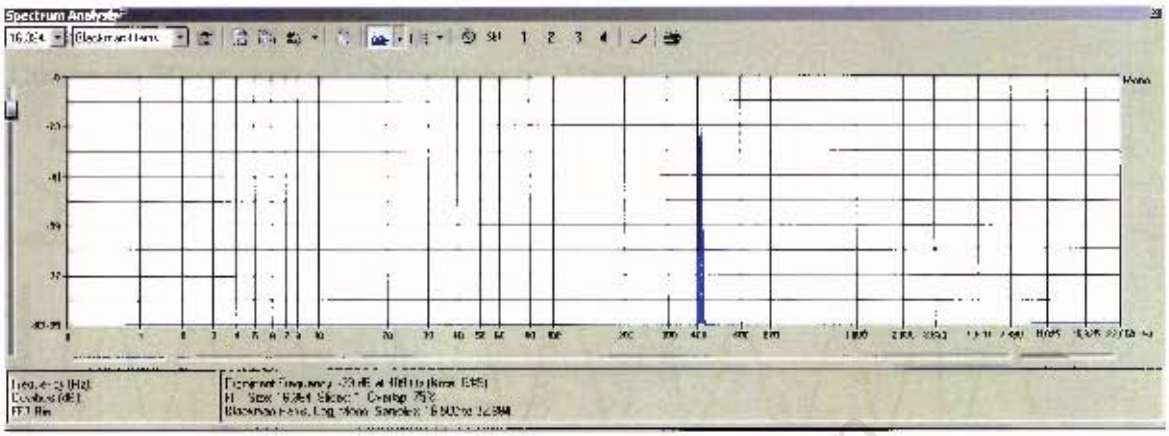

Figure 4.43: Spectrum analysis of injected 440Hz sinusoidal waveform

The second part of the experiment visually analyses 250ms of the audio signal recorded in the waveform analyser and identifies irregularities called glitches in the received audio signal. The glitches are counted and presented for each of the control mechanisms. Figure 4.44 illustrates a glitch in the sinusoidal waveform.

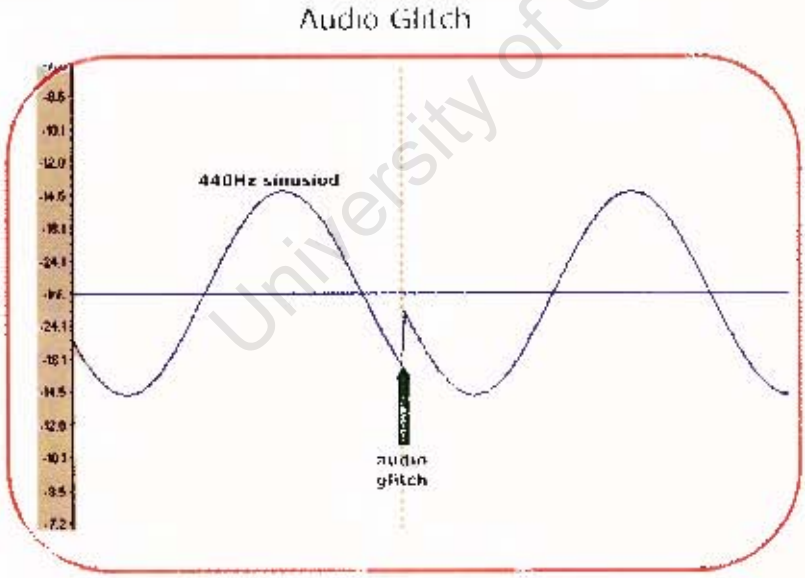

Figure 4.44: Illustration of an audio glitch

Metric_1 Temporal fidelity

Metric_2 Audio glitches

4.3.4.4 Result Data and Findings

Control Mechanism 1: No control Mechanism

Result Data

The results for the temporal fidelity and audio glitch metrics for no control mechanism are found in Figures 4.45 and 4.46.

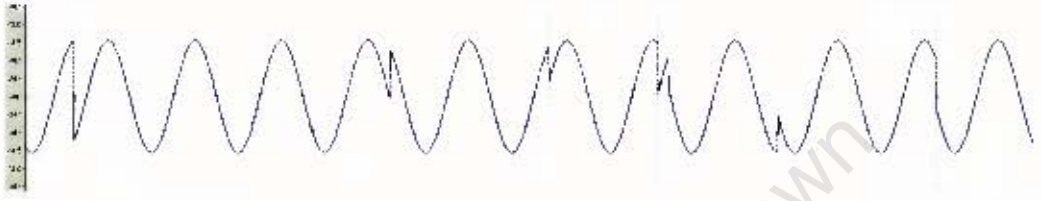

Figure 4.45: Received sinusoidal waveform for no control mechanism

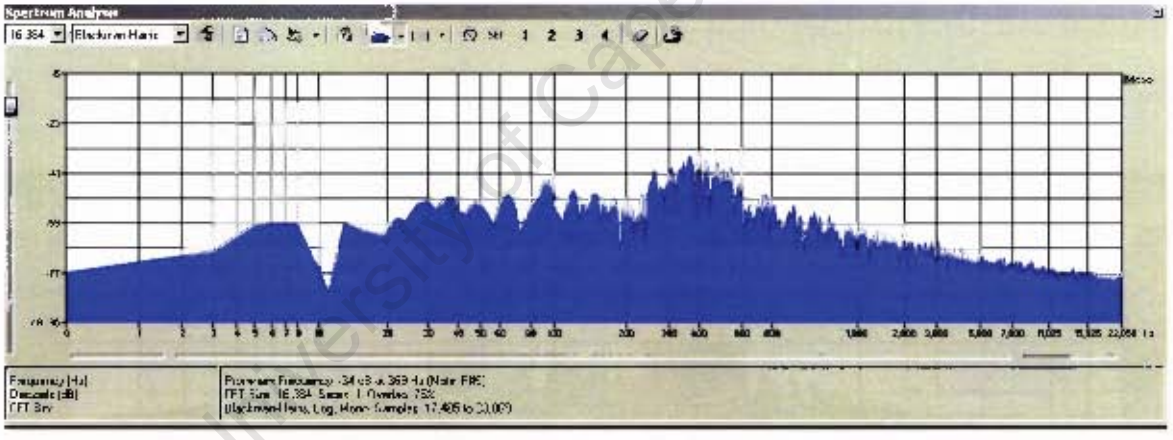

Figure 4.46: Spectrum analysis of received sinusoidal waveform for no control mechanism

From the data above we can see that the sinusoidal shape is visible in Figure 4.45 but does contain glitches. The glitch count amounted to 57 glitches for 250ms of this received audio data. The spectrum analysis graph in Figure 4.46 shows that while the prominent frequencies are around the 440Hz mark, the magnitude of all humanly audible frequencies is above -95dB.

Findings

The analysis of the data indicates glitches were introduced into the system as seen in Figure 4.45. The glitches are confirmed by analysing the spectrum

of the received audio signal and finding additional spectral content. This rate of glitches [57/250ms] significantly degrades the audio signal. While the visual inspection of the sinusoid might seem not adversely affected, the spectrum analysis as well an audible evaluation shows that the sinusoid is completely masked by the glitch effect. The audible evaluation sounds like static white noise with a low frequency element with no evidence of the injected 440Hz sinusoid.

Control Mechanism 2: OSC Scheduler

Result Data

The results for the temporal fidelity and audio glitch metrics for the OSC Scheduler are found in Figures 4.47 and 4.48.

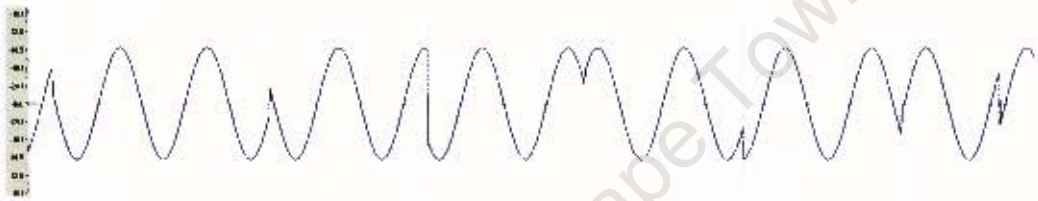

Figure 4.47: Received sinusoidal waveform for OSC Scheduler

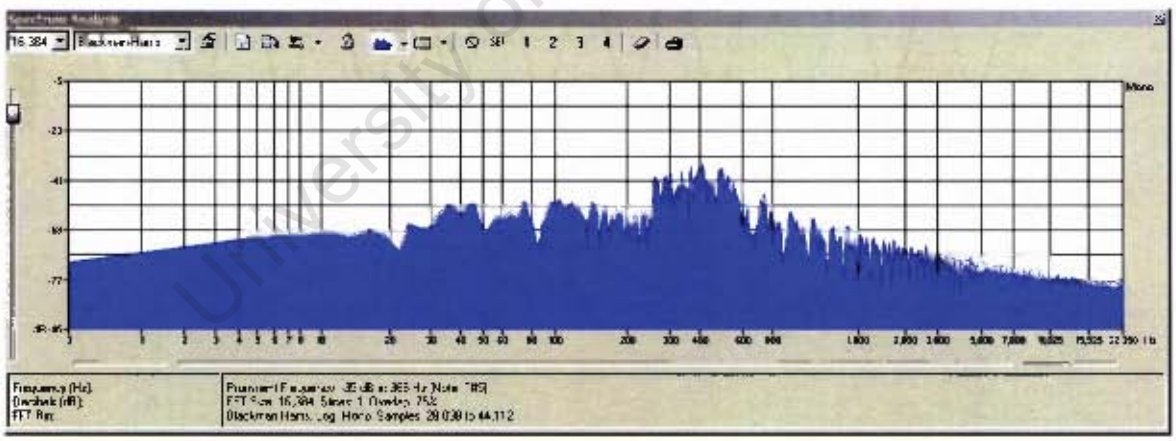

Figure 4.48: Spectrum analysis of received sinusoidal waveform for OSC Scheduler

From the data above we can see that the sinusoidal shape is visible in Figure 4.47 but does also contain glitches. The glitch count amounted to 49 glitches for 250ms of this received audio data. The spectrum analysis graph in Figure 4.48 shows that while the prominent frequencies are around the 440Hz mark, the magnitude of all humanly audible frequencies is above -95dB.

Findings

The analysis of the data indicates glitches were introduced into the system as seen in Figure 4.47. The glitches are confirmed by analysing the spectrum of the received audio signal and finding additional spectral content. This rate of glitches [49/250ms] is slightly reduced when compared to using no control mechanism but still significantly degrades the audio signal. The frequency content around the 440Hz mark in Figure 4.48 is slightly more prominent than that for no control mechanism. While the visual inspection of the sinusoid might seem not adversely affected, the spectrum analysis as well an audible evaluation shows that the sinusoid is completely masked by the glitch effect. The audible evaluation for the OSC Scheduler also sounds like static white noise with a low frequency element with no evidence of the injected 440Hz sinusoid.

Control Mechanism 3: JitterBuffer

Result Data

The results for the temporal fidelity and audio glitch metrics for the JitterBuffer are found in Figures 4.49 and 4.50.

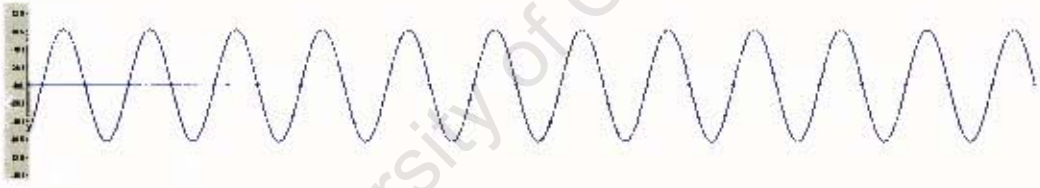

Figure 4.49: Received sinusoidal waveform for JitterBuffer

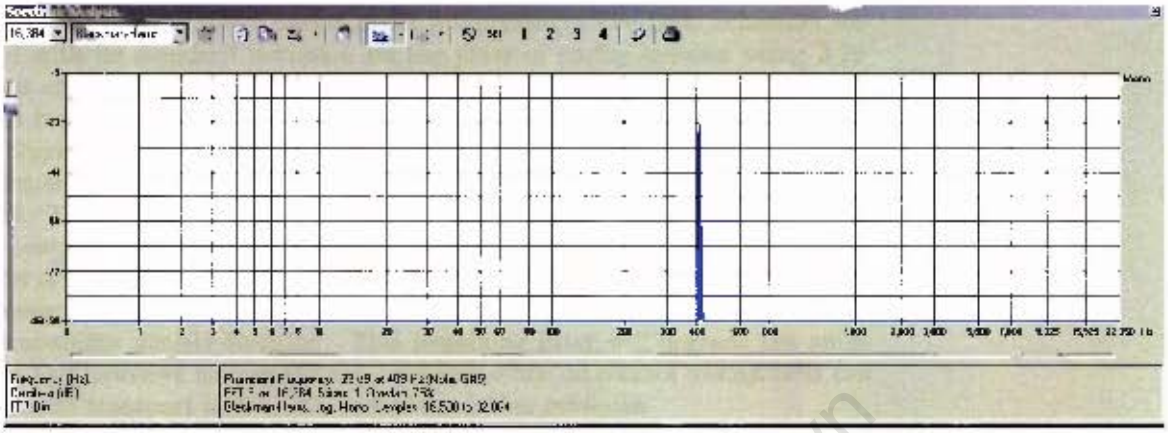

Figure 4.50: Spectrum analysis of received sinusoidal waveform for JitterBuffer

From the data above we can see that the sinusoidal shape is perfectly preserved in Figure 4.49 and does not contain any glitches. There was therefore no glitch count for the JitterBuffer control mechanism. The spectrum analysis graph in Figure 4.50 shows that there is only frequency content at the 440Hz mark and no other additional frequency content above -95dB.

Findings

The analysis of the data show that no glitches were introduced into the system as seen in Figures 4.49 and 4.50. There is no evidence to indicate degradation of the audio signal. The audible evaluation confirms this with a pure frequency tone observed. The JitterBuffer control mechanism therefore removes all jitter and preserves the order of packets at the receiving end and completely reproduces the injected audio signal.

The results data for the 3 control mechanisms proved as expected with only the JitterBuffer completely preserving the injected audio signal. The OSC Scheduler and no control mechanism results displayed glitches and confirms that they could not completely order packets and completely remove jitter introduced by the network.

4.3.5 Discussion and Conclusion

Experiments 5, 6 and 7 have done an evaluation of the audio data type and the performance requirements when transporting uncompressed audio data between OscVstBridge prototype plugin instances. The metrics measured are packet order preservation in experiment 5, latency and jitter in experiment 6 and temporal fidelity in experiment 7. Each experiment evaluated each of the control mechanisms in the prototype.

Experiment 5 showed that the JitterBuffer control mechanism achieved the

best results with complete packet order preservation. The OSC Scheduler followed with its standard deviation on the jitter of packet number being 3.29. This is an improvement on the standard deviation of 10.24 for no control mechanism but does not achieve complete packet order preservation and therefore will degrade the audio signal.

Experiment 6 measured latency and jitter for each of the control mechanisms. The JitterBuffer control mechanism has been demonstrated to be the best control mechanism in terms of the latency and jitter metrics and the preservation of an audio signal. The OSC Scheduler reduces the jitter introduced by the network but does not completely remove it as there is still jitter present in its probability density function. This remaining jitter will degrade the audio signal and therefore neither the OSC Scheduler nor no control mechanisms can be used to transport audio data over jitter-ridden networks.

The results data for experiment 7 shows that only the JitterBuffer completely preserves the injected audio signal. The OSC Scheduler and no control mechanism results displayed glitches and confirm that they could not completely order packets and completely remove jitter introduced by the network.

Experiment 7 is an 'end-user' application-type test and confirms the results obtained in experiments 5 and 6. The JitterBuffer control mechanism outperformed the other control mechanisms in both experiments 5 and 6 and hence also in experiment 7. The JitterBuffer completely preserved the audio signal in experiment 7 and therefore must have full packet order preservation and complete jitter removal. Experiment 5 confirms the packet order preservation and experiment 6 has most of the sample data [>75%] having no jitter for the JitterBuffer. The jitter introduced is due to the non-synchronous processReplacing() method calls done by the VST host and if this cause is removed there will be 100% complete jitter removal as shown in experiment 7 with the audio signal being preserved completely.

The OSC Scheduling control mechanism shows an improvement when compared to no control mechanism. In experiment 5 the standard deviation on the jitter of packet number is reduced, and in experiment 6 the jitter is also reduced. This reduction is however of no benefit to transporting audio over a jitter network since full packet order preservation and complete jitter removal is necessary.

4.4 Summary of Results and Findings

The experiments evaluating the parameter and audio data types have positive results for the performance of the OscVstBridge prototype.

The results for the experiments conducted on parameter data type with mean latency, peak jitter, scheduling and packet loss metrics have surpassed or met the benchmark set against which to measure. The mean latency upper bound of 2.57ms fell within its requirement benchmark of 10ms. The peak jitter upper bound met its benchmark at 10ms. The scheduling and packet loss achieved an upper bound of 100% each.

The performance of the parameter data type within the OscVstBridge plugin should be acceptable when comparing to the benchmarks set.

The audio data type experiment set with scheduling, latency, jitter and temporal fidelity metrics have been performed on each of the control mechanisms. The JitterBuffer control mechanism is the only control mechanism that can preserve audio data transportation over a jitter-ridden network. OSC Scheduling does reduce the jitter and packet order preservation but is not suitable to transport audio data over a jitter network.

Chapter 5

Discussion and Conclusions

5.1 Summary of research done, aims and goals

The focus of this research was to develop a prototype VST plugin called OscVstBridge bridging VST systems to the OSC network domain. OscVstBridge would share audio resources such as parameter control, audio, transport and synchronisation data between VST and OSC systems. The prototype core features developed are:

- a VST host spoofer, a data bridge between OSC and VST,
- OSC data type development,
- OSC environment development,
- VSTXML to OSC namespace mapping, and
- JitterBuffer and OSC Scheduling control mechanism implementations.

Upon completion of the prototype development the research evaluated the performance of the OscVstBridge prototype by conducting 7 experiments. The first 4 experiments focussed on the parameter data type with mean latency, peak jitter, scheduling and packet loss metrics. The result data for these experiments were compared against industry benchmarks. The last 3 experiments focussed on the audio data type and control mechanisms with scheduling, latency, jitter and temporal fidelity metrics. The control mechanisms are compared against one another.

The main research goals of this dissertation were:

1. *Can we use Open Sound Control [OSC] as an open protocol interface to implement networked communications between VST systems and OSC environments facilitating distributed audio processing.* This was done by developing the OscVstBridge prototype plugin that bridged between VST

and OSC systems. The prototype development constituted a large portion of this dissertation with 4791 lines of code, 52 classes and 10 packages being developed. The data types of parameter, audio, transport and synchronisation deployed in the OscVstBridge prototype allowed data to be shared between the two systems. This data sharing facilitated distributed audio processing.

2. *Can we build a rich OSC namespace using the VstXml definition.* This was achieved in the development of the OscVstBridge prototype plugin by accepting a VSTXML file upon plugin instantiation and constructing the OSC address space from this XML file. The mapping from VSTXML to OSC was achieved using the JDOM library.

3. *How can we measure and evaluate system performance based on QOS and industry accepted benchmarks for control [parameter] data.* This was done by conducting and evaluating the parameter experiments. Conservative music control benchmarks (such as using the music instrument interface design system maximum latency of 10ms) were set against which the experimental data was evaluated.

4. *To determine and compare the system performance in applying different QOS control mechanisms on audio data.* This was achieved by conducting and evaluating the audio experiments in experiments 5 to 7. The Jitter-Buffer, OSC Scheduling as well as no control mechanism were evaluated and compared to each other.

5.2 Overview of experiments and results

5.2.1 Parameter experiments

The parameter experiments measured lower and upper bounds for the mean latency, peak jitter, scheduling and packet loss metrics. The upper bound gives the worst-case results for the metric under test. The mean latency metric results are compared to the maximum instrument interface design latency of 10ms[20] used in the audio and music industry. The peak jitter benchmark is obtained by comparing MIDI interface technology results[37]. A peak jitter of 10ms is used.

Figure 5.1 shows the upper and lower bounds for the mean latency, peak jitter, scheduling and packet loss metrics evaluated in the experiments.

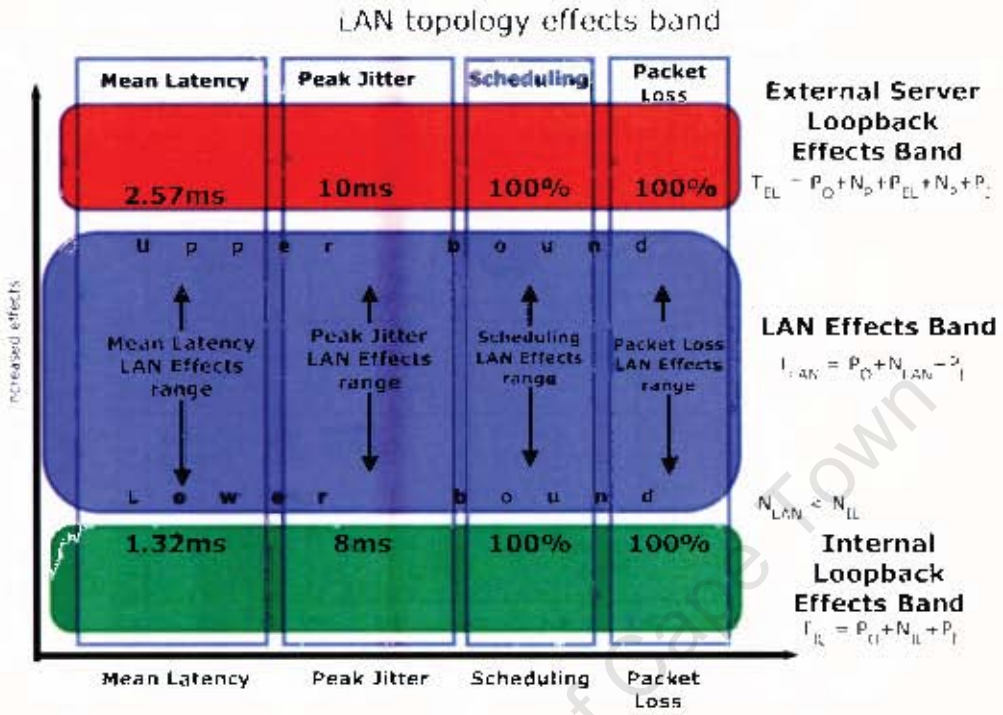

Figure 5.1: Upper and Lower bounds for mean latency, peak jitter, scheduling and packet loss

The mean latency upper bound achieved 2.57ms - well within the benchmark of 10ms set. The latency metric therefore does not degrade the performance of the OscVstBridge prototype. The peak jitter upper bound for the prototype is 10ms which met the benchmark set. When this metric is compared to MIDI interfaces, the prototype outperformed the newer USB MIDI type interfaces but not the legacy older interfaces. This can be seen in Figure 5.2. In the context of musical control this peak jitter value of 10ms is acceptable. The scheduling and packet loss metric obtained perfect results, achieving 100% scheduling and having no packet loss for a sample number of about 5000 packets. The performance of the parameter data type within the OscVstBridge plugin is acceptable, exceeding or meeting the benchmarks set.

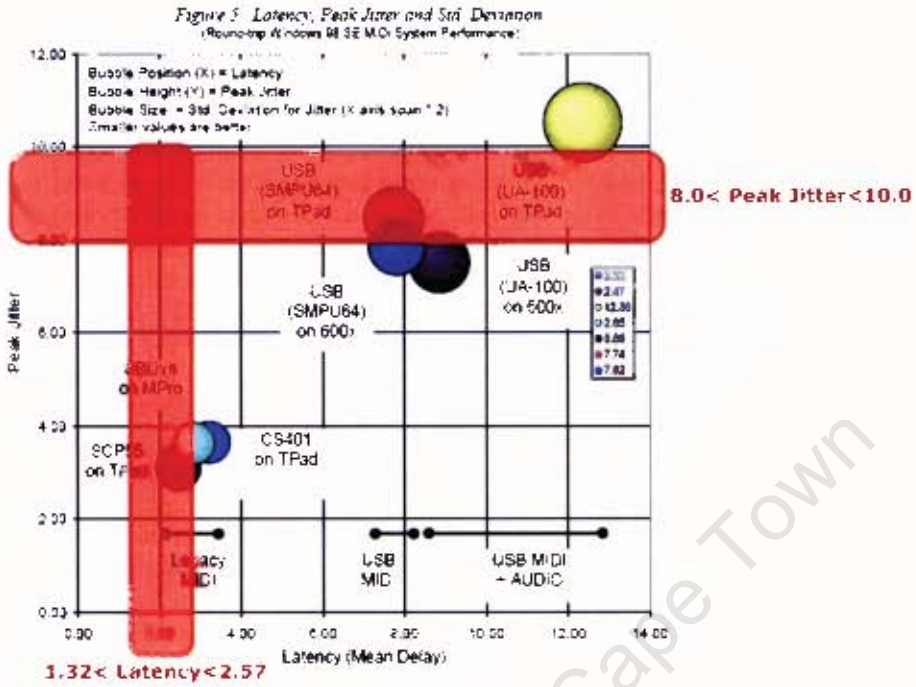

Figure 5.2: Latency and Peak Jitter metrics for MIDI interface technologies

5.2.2 Audio experiments

The audio experiments evaluate and compare the control mechanisms deployed in the OscVstBridge prototype with scheduling, latency, jitter and temporal fidelity metrics. Of particular importance are the scheduling and jitter metrics since if they do not achieve perfect results, the transported uncompressed audio data will be degraded. The control mechanisms aim to have an improved effect on the metrics. The temporal fidelity metric consolidates the scheduling, latency and jitter metrics as it measures the perceived audio quality that the end user will be exposed to. The network jitter emulation range is 1-100ms.

No control mechanism has a degrading effect on the transported audio signal over a jitter-ridden network. The scheduling metric performed poorly with a standard deviation on the packet number received being 10.24. The latency and jitter metrics also perform poorly with a latency of 90.7ms and a jitter standard deviation of 41.08. The consolidating temporal fidelity metric concludes with 57 glitches observed in 250ms of received audio data. An audible evaluation of the received audio signal is observed as static white noise with a low frequency element despite a pure 440Hz sinusoid being injected at the source. The scheduling and jitter for no control mechanism do not achieve perfect results. The temporal fidelity metric results confirm that the audio signal is degraded when no control mechanism is used over a jitter-ridden network.

The OSC Scheduler control mechanism also degrades a received audio signal when transported over a jitter network. The scheduling metric achieves better results than for no control mechanisms with a standard deviation on packet number of 3.29. The latency and jitter metric also attain improved results with a latency of 192.6ms and a jitter standard deviation of 12.96. The scheduling and jitter metrics are improved upon going from 10.24 to 3.29 for the former and from 41.08 to 12.96 for the latter. However since the scheduling and jitter metrics do not attain perfect results, the audio signal is expected to be degraded. The temporal fidelity metric confirms this by 49 glitches observed in the received audio signal. The received audio signal is also audibly observed as degraded static white noise with some low frequency content. The OSC Scheduler control mechanism therefore does not preserve an audio signal when transported over a jitter network.

The JitterBuffer control mechanism obtained promising results and was found not to degrade the audio signal over a jitter-ridden network. The scheduling metric obtained perfect result data with a standard deviation on packet number of 0. The jitter metric also obtained near perfect results with a standard deviation of 4.77. The cause of this imperfection is due to the non-synchronous nature of VST host calls to the plugin. The mean latency is 186.4ms. The temporal fidelity metric contained no glitches and perfectly reproduced the injected source audio signal of 440Hz. This indicates that the standard deviation on the jitter metric is indeed caused by the non-synchronous VST host calls. The JitterBuffer control mechanism does preserve the audio signal when transported over a jitter-ridden network.

The jitter for the JitterBuffer has presented interesting results. By observing the preserved non-degraded audio signal in the temporal fidelity experiments the expected jitter should be 0 with a constant latency. However there is latency variance that is caused by the non-synchronous calls from the VST host to the plugin.

The JitterBuffer as well as no control mechanism performed as expected with the former perfectly preserving audio data and the latter not at all. The OSC Scheduler results are surprising as it was expected that this control mechanism would preserve the audio signal over a jitter network by yielding perfect metric results for scheduling and jitter. The reduction in these metrics was not adequate for audio reconstruction.

5.3 Contributions

5.3.1 Prototype contributions

The OscVstBridge prototype has provided a novel bridging interface between the widely used VST system and the open OSC network domain and therefore

has facilitated modern network communications amongst VST audio processes.

OSC has been compared to MIDI[13] and is proposed as its successor. The OscVstBridge prototype developed assists towards OSC achieving this successor status by contributed towards OSC establishing itself within the audio processing system of VST and in industry.

The benefits of the OscVstBridge prototype providing high speed musical networking in VST systems allows audio processing to be distributed amongst many networked computers. This network is scalable and therefore VST processing farms become possible in such a networked system. Other benefits are that multiple, simultaneous and possibly remote access to audio resources become possible, as well as location independent musical collaborations such as NMP.

The open framework which OSC implements allows inter-vendor networked communications amongst VST host and plugins. This facilitates inter-operability of audio and musical systems promoting standardisation.

The OscVstBridge prototype has also developed a novel process to facilitate the construction of a rich, hierarchical OSC namespace using VSTXML. The mapping from VSTXML to the OSC namespace provides improved control structure within VST plugins by utilising the flexible namespace provided by OSC. Multiple, intuitive parameter names as well as aliases, high level controls and range mapping with multiple parameter addressing and OSC pattern matching can be deployed, simplifying user control in VST plugins.

The OscVstBridge prototype also developed an OSC type in Java using the NetUtil OSC library that would transport uncompressed PCM audio data. This OSC type was developed as the non-standard vector of floats 'v' type encapsulating audio data within an OSC message.

5.3.2 Experiment contributions

The parameter experiments show that transporting parameter data over an OSC network has no performance issues. The worst case latency of 2.57ms meets the audio interface guideline latency[20]. The peak jitter obtained is also 10ms which performs better than USB type MIDI interfaces[37]. The scheduling and packet loss metrics achieve perfect results and therefore the parameter data type performance measures are acceptable.

The OSC Scheduler control mechanism does not completely remove jitter and perfectly preserve packet order when transporting OSC packets over a jitter-ridden network with a jitter range of 1-100ms. The OSC Scheduler does reduce the jitter and improve on the packet order preservation but this is not adequate to prevent audio data degradation at the receiver. Therefore the OSC Scheduling control mechanism is not suitable for real-time transportation of uncompressed audio using OSC over a jitter-ridden network.

5.4 Recommendations for future work

5.4.1 OSC Scheduler performance investigation

The OSC Scheduling control mechanism's inability to transport audio data over a jitter network is amongst the most surprising results in this dissertation. The jitter network deployed for the evaluation was extreme, with up to 100ms jitter experienced between packets. Further investigation into the reasons for the OSC Scheduler not preserving audio over this jitter network needs to be done. This future work could be coupled with experiments evaluating the performance of the OSC Scheduler in the transportation of audio over a lesser jitter-ridden network.

5.4.2 Compression of audio for transportation over OSC network

Transporting 1 channel of 44100Hz uncompressed PCM audio data over a computer network requires substantial bandwidth. The float type used in the float vector type 'v' is 32-bit and therefore the bandwidth requirement is 44100Hz * 32bit = 1.411 Mbit/sec. A typical VST plugin would be configured as a stereo plugin with 2 input channels and 2 output channels. Therefore a total of 4 channels amounts to 5.64Mbit/sec. This is a large amount of bandwidth and future work could comprise compression and decompression schemes for this uncompressed PCM audio data for efficient transportation over computer networks. This would result in reduced bandwidth consumption with added latency introduced to the overall transport system due to codec compression latencies. Exploration into loss-less compression algorithms would add most value.

5.4.3 Transport and synchronisation data type implementations

The OscVstBridge prototype developed a TimeInfo class or data type that contains information regarding transport and synchronisation of musical and audio systems such as VST. This data type is fully developed but has no application implementation in the OscVstBridge prototype. The benefits of an implementation include synchronisation of multiple VST hosts and plugins and facilitates a master control VST host with all slave VST hosts following using the transport attributed in the data type. Future work could include such an implementation.

5.4.4 OscVstBridge prototype evaluation over the Internet

The evaluation of the OscVstBridge prototype was done over a LAN for the parameter experiments and a jitter emulated network for the audio experiments.

Future work could include evaluating the performance of the OscVstBridge prototype over the Internet since this network is most likely to have applications deployed over it. Separate evaluations for each data type can be done.

Bibliography

[1] D. Anderson, R. Doris, and J. Moorer. A Distributed Computer System for Professional Audio. In *Proceedings of the Second ACM International Conference on Multimedia (MULTIMEDIA '94)*, pages 373–380, New York, October 1994. ACM Press.

[2] A. Bailey. *Network Technology for Digital Audio*. Focal Press, 2001.

[3] Á. Barbosa. *Computer-Supported Cooperative Work for Music Applications*. PhD thesis, Pompeu Fabra University, Barcelona, 2002.

[4] Á. Barbosa. Displaced Soundscapes: A survey of network systems for music and sonic art creation. *Leonardo Music Journal*, 13:2003, 2003.

[5] S. Biaz, R.O. Chapman, and J.P. Williams. RTP and TCP based MIDI over IP protocols. In *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, pages 112–117, New York, NY, USA, 2005. ACM.

[6] S. Bray and G. Tzanetakis. Distributed audio feature extraction for music. In *Proceedings of the International Conference on Music Information Retrieval*, pages 434–437, 2005.

[7] A. Chadda. Quality of Service testing methodology. Master's thesis, University of New Hampshire, 2004.

[8] A. Chaudhary. *Perceptual Scheduling in Real-time Music and Audio Applications*. Phd thesis, University of California, Berkeley, 2001.

[9] A. Chaudhary. Open Sound World. http://osw.sourceforge.net, 2002.

[10] F. Ehrentraud. OSC-SYN namepace. Open Sound Control website http://opensoundcontrol.org/topic/148, 2007.

[11] A. Fraietta. Open Sound Control: Constraints and Limitations. In *International Conference on New Interfaces for Musical Expression (Nime08)*, 2008.

[12] D. Frankowski and J. Riedl. Hiding jitter in an audio stream. Technical Report TR-93-50, Department of Computer Science, University of Minnesota, 1993.

[13] A. Freed, A. Schmeder, and M. Zbyszynski. OSC Showcase. In *Maker Faire 2007*, San Mateo, CA, USA, 20/10/2007 2007.

[14] T. Hankins, D. Merrill, and J. Robert. Circular Optical Object Locator. In *Conference on New Interfaces for Musical Expression (NIME-02)*, pages 163–164, Dublin Ireland, 2002.

[15] D.M. Huber. *The MIDI manual.* Sams, Indianapolis, IN, USA, 1991.

[16] D. Hutchison, G. Coulson, A. Campbell, and G. Blair. Quality of Service management in distributed systems. In *Department of Computing, Lancaster University, Lancaster*, pages 1–4. Addison Wesley, 1994.

[17] T. Jehan and B. Schoner. An audio-driven perceptually meaningful timbre synthesizer. Havana, Cuba, 2001. International Computer Music Conference.

[18] A. Jones and A. Hopper. Handling Audio and Video Streams in a Distributed Environment. In *Proc. ACM Symp. on Operating Systems Principles*, pages 231–243, 1993.

[19] M. Karam and F.A. Tobagi. Analysis of delay and delay jitter of voice traffic in the Internet. In *Proceedings of IEEE INFOCOM*, pages 824–833, 2001.

[20] J. Kleimola. Latency Issues in Distributed Musical Performance. Technical Report T-111.5080, Helsinki University of Technology, 2006.

[21] J. Lazzaro and J. Wawrzynek. A case for network musical performance. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 157–166, New York, NY, USA, 2001. ACM.

[22] J. Lazzaro and J. Wawrzynek. An RTP Payload Format for MIDI. *Audio Engineering Society*, 2004.

[23] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC1305, http://www.faqs.org/rfcs/rfc1305.html, March 1992.

[24] F.R. Moore. The dysfunctions of MIDI. *Comput. Music J.*, 12(1):19–28, 1988.

[25] D. Murphy, C. Newton, and D. Howard. Digital waveguide mesh modelling of room acoustics: Surround-sound boundaries and plugin implementation. pages 198–202, Limerick, Ireland, November 02 2001. Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFx).

[26] D.T. Murphy and J. Mullen. Digital Waveguide Mesh Modelling of Room Acoustics: Improved Anechoic Boundaries. In *Proceedings of the 5th International Conference on Digital Audio Effects (DAFX-02)*, Hamburg, Germany, 2002.

[27] D. Overholt. The MATRIX: a novel controller for musical expression. In *International Conference on New Interfaces for Musical Expression (NIME 01)*, 2001.

[28] L. Parziale. *TCP/IP Tutorial and Technical Overview.* Redbook. IBM International Technical Support Organization, 2006.

[29] C. Perkins, O. Hodson, and V. Hardman. A survey of packet loss recovery techniques for streaming audio. *IEEE Network*, 12:40–48, 1998.

[30] S.T. Pope and A. Engberg. Distributed Control and Computation in the HPDM and DSCP Projects. In *Symposium on Sensing and Input for Media-Centric Systems (SIMS)*, pages 38–43, Santa Barbara, CA, 2002.

[31] Muse Research. Receptor. World Wide Web electronic publication, (http://www.museresearch.com/receptor.php), 2008.

[32] A. Schmeder and A. Freed. Implementations and applications of Open Sound Control timestamps. Technical report, 2008.

[33] A. Schmeder and A. Freed. micro-OSC: The Open Sound Control Reference Implementation for Embedded Devices. Technical report, 06/2008 2008.

[34] W. Stallings. *Data and Computer Communications (3rd ed.).* Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1991.

[35] L. Stuck. La Kitchen's "Toaster" and "Kroonde" wired and wireless Data-Acquisition Systems. In *OSC Conference 2004*, 30/07/2004 2004.

[36] Steinberg Media Technologies. VST Plug-ins SDK 2.4. World Wide Web electronic publication, http://www.steinberg.net/steinberg/ygrabit/vstsdk/OnlineDoc/vstsdk2.4/index.html, 2005.

[37] J. Wright and E. Brandt. System level midi performance testing. In *Proceedings of the International Computer Music Conference, Havana*, 2001.

[38] M. Wright. Open Sound Control 1.0 Specification. 2002.

[39] M. Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10:193–200, 2005/12/01 2005.

[40] M. Wright, A. Freed, and A. Momeni. Open Sound Control: State of the Art 2003. pages 153–159, Montreal, 2003. International Conference on New Interfaces for Musical Expression.

[41] M. Zbyszynski and A. Freed. OSC Control of VST Plug-ins. In *OSC Conference 2004*, 30/07/2004 2004.

[42] M. Zbyszynski and A. Freed. Control of VST Plug-ins Using OSC. pages 263–266, Barcelona, Spain, 2005. International Computer Music Association.

Nomenclature

AUDIO AND CONTROL LATENCY are the latencies associated with the performance of the VST wrapper.

AUDIO BLOCK SIZE is the buffer size that the VST host passes to the VST plugin to let it know of the size of the audio data block sent to it.

AUDIO DATA is the type for the audio objects contained within the VST wrapper

DATA BRIDGE bridges data between the OSC and VST domains

DISTRIBUTED AUDIO PROCESSING distributes the processing load across networked processing units.

EXTERNAL OSC DEVICES are deviced located outside the boundary of the VST wrapper that are capable of transmitting and receiving valid OSC messages

IP ADDRESS is a unique address utilised on a computer network that together with a IP port forms an IP socket

IP PORT are the ports attached to an IP address forming a IP socket.

JAVA is the programming language used to develop the VST wrapper

JVSTWRAPPER is the Java plugin framework that is used to develop the VST wrapper.

LOOPBACK COMMUNICATIONS uses the 'localhost' or ethernet loopback address of '127.0.0.1' to direct traffic back into the same IP address

NETWORK is an Ethernet datalink layer using the TCP/IP protocol suite.

OSC is Open Sound Control and is a message based protocol developed to facilitate modern networked communications amongst audio processing units.

OSC ADDRESS PATTERN is the destination address target within an OSC server.

OSC ADDRESS SPACE is the hierarchical tree structure of the addressable nodes within an OSC server for which OSC messages can be processed

OSC CLIENT sends OSC data over networked technology to OSC servers.

OSC DATA is data transmitted and received from the OSC domain complying to the OSC protocol definition

OSC DATA TYPE is a type identifying the data contained within an OSC message

OSC ENVIRONMENT contains external OSC servers and clients capable of interfacing to the VST wrapper.

OSC HIERARCHICAL NAMESPACE is the structured hierachical namespace within OSC servers.

OSC METHOD is a certain behaviour of the OSC server that is executed when an OSC message is matched via the OSC address space to that OSC method

OSC SERVER receives OSC data from OSC clients and processes it through its address space matching and OSC methods

OSC SETTINGS are settings within the VST wrapper for configuring the communication parameters of the OSC server and client

PARAMETER CONTROL DATA is the type for the parameter objects contained within the VST wrapper

PLUGIN UI is the user interface of the VST wrapper system

PROCEDURE CALLS are used by the VST host to communicate with the VST plugin and transfer data between the two entities.

REAL TIME INFO DISPLAY is a display in the plugin UI that shows real time data about the VST wrapper

ROUTING ELEMENTS will allow for mapping of OSC data to VST data and vice versa.

STATISTICAL INFO DISPLAY is a display in the plugin UI that shows statistical data about the VST wrapper

SYNCHRONISATION CONTROL DATA is the type for the synchronisation objects contained within the VST wrapper

TRANSPORT CONTROL DATA is the type for the transport objects contained within the VST wrapper

VST is Virtual Studio Technology by Steinberg is the informal standard for audio plugin processing

VST DATA is data sourced and supplied directly to the VST host.

VST DATATYPES has two favours of audio and control types. Control types are further subdivided into parameter, event, transport and synchronisation.

VST HOST is an audio application capable of loading and hosting VST plugins

VST HOST SPOOFER is an implementation of a VST plugin providing all the procedure calls required by the VST host.

VST PARAMETER is hosted by VST plugins and are controllable by VST host and the OSC domain through the VST wrapper

VST SDK v2.4 is a release of the VST plugin SDK that accomodates the *VSTXML* definition

VST SYSTEM is a VST plugin attaching to a VST host.

VST TRANSCEIVER is a combination of a VST Transmitter and VST Receiver that will interface to the VST environment.

VST WRAPPER is functionally a VST plugin that can be loaded by a VST host. It bridges data between OSC and VST domains

VSTXML DEFINITION released with the VST SDK v2.4 by Steinberg allows for a hierarchical namespace of vst plugin parameters to be constructed.

Appendix A

Class Descriptions of OscVstBridge prototype

A.1 VstPlugin Class

DESCRIPTION

This is the main class of the prototype. The VST host call the methods defined in this class upon instantiating the plugin as well as during real time operations. The interfacing between the VST host and the VST plugin environment is done through this class. Blocks of code that send and receive parameter data from the OSC domain are the getParameter() and setParameter() methods. The equivalent methods for audio data are processReplacing() and the implemented method from the Observer interface update(). The processReplacing() method is called by the host frequently as it is responsible for the transfer of audio data between VST host and plugin.

This class also registers Observer classes with Observable classes which provides the conduit for data flow within the plugin.

PACKAGE

vst.spoof

INHERITS THESE CLASSES

This class inherits from the VSTPluginAdapter class from the JVSTwRapper library. This library facilitates development of VST plugins from the Java environment through an implementation of the JNI.

IMPLEMENTS THESE INTERFACES

The VstPlugin class also implements the Observer for observing changes in other objects. The Observer class and Observable interface are core to the data flow processes within the plugin

INSTANTIATES OBJECTS FROM THESE CLASSES

- Gui
- VstProgram

- IconFlashthread
- XmlParser
- AudioReceiverThread
- Bridge
- OscManager
- VstManager
- VstAudio

A.2 Gui Class

DESCRIPTION

This class is responsible for the user interface associated with the plugin displaying information about the plugin internals to the user. The configuration settings, real time data, the OSC Address space are displayed to the user. This class constructs the OSC Address Space that is displayed to the user.

PACKAGE

vst.spoof

INHERITS THESE CLASSES

VSTPluginGUIAdapter

IMPLEMENTS THESE INTERFACES

ChangeListener

INSTANTIATES OBJECTS FROM THESE CLASSES

- VstPlugin
- DefaultMutableTreeNode

A.3 VstProgram Class

DESCRIPTION

A simple class used to create a VstProgram object used by the VstPlugin class.

PACKAGE

vst.spoof

INHERITS THESE CLASSES

none

IMPLEMENTS THESE INTERFACES

none

INSTANTIATES OBJECTS FROM THESE CLASSES

none

A.4 Config Class

DESCRIPTION

A static class used to store static variables that are used throughout the prototype especially within constructors of different classes. It stores data relevant to the core VstPlugin, Vst Audio, OSC server, OSC client, JitterBuffer, and Gui.

PACKAGE

vst.spoof

INHERITS THESE CLASSES

none

IMPLEMENTS THESE INTERFACES

none

INSTANTIATES OBJECTS FROM THESE CLASSES

none

A.5 OscManager Class

DESCRIPTION

A class that is primarily used for instantiating OSC server and OSC client objects. It is also used in the data flow process for parameter and audio data.

PACKAGE

osc

INHERITS THESE CLASSES

none

IMPLEMENTS THESE INTERFACES

Observer

INSTANTIATES OBJECTS FROM THESE CLASSES

- OscServer
- OscClient

A.6 OscServer Class

DESCRIPTION

This is the receiving class for OSC messages. This is achieved through the messageReceived() method in the OSCListener interface. All OSC messages received at the SocketAddress is addressed by different blocks of the messageReceived() method. This caters for all OSC data types.

PACKAGE

osc

INHERITS THESE CLASSES

Observable

IMPLEMENTS THESE INTERFACES

none

INSTANTIATES OBJECTS FROM THESE CLASSES

- OSCServer from the NetUtil library
- OSCListener from the NetUtil library
- SocketAddress
- ByteBuffer

A.7 OscClient Class

DESCRIPTION

This class sends data from the plugin to the OSC domain through the DatagramChannel. There are methods for opening and closing the DatagramChannel. This class constructs OSC messages from the VST data types it receives before transmitting it onto the network. The three main data type are parameter, audio and VstTimeInfo.

PACKAGE

osc

INHERITS THESE CLASSES

none

IMPLEMENTS THESE INTERFACES

none

INSTANTIATES OBJECTS FROM THESE CLASSES

- OSCBundle
- OSCPacket
- OSCMessage
- OSCPacketCodec
- DatagramChannel
- SocketAddress
- ByteBuffer

A.8 VstManager Class

DESCRIPTION

Primarily used as a parent class for objects such as VstData, VstAudioReceiver and VstTransmitter. There are no methods within this class besides the constructor.

PACKAGE

vst.transceiver
INHERITS THESE CLASSES
Observable
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES

- VstData
- VstAudioReceiver
- VstTransmitter
- VstParameter

A.9 VstData Class

DESCRIPTION
this class is used for creating VST objects of the parameter, audio and VST-TimeInfo data types. It also creates a XmlParser object used in the translation of VSTXML data for the population of data within the parameter data type.

This class extends and implements the Observable and Observer classes respectively which are used in the data flow processes for the different data types.

PACKAGE
vst.transceiver
INHERITS THESE CLASSES
Observable
IMPLEMENTS THESE INTERFACES
Observer
INSTANTIATES OBJECTS FROM THESE CLASSES

- XmlParser
- VstParameter
- VSTTimeInfo
- VstAudio

A.10 Bridge Class

DESCRIPTION
This class is a critical path in the data flow process. All data in any data flow process uses this class. The class uses the Mapping object as argument into determining whether objects are bridged or not.

PACKAGE
bridge

INHERITS THESE CLASSES
Observable
IMPLEMENTS THESE INTERFACES
Observer
INSTANTIATES OBJECTS FROM THESE CLASSES

- Mapping
- VstParameter
- VstAudio
- VSTTimeInfo
- OscParameter
- OscAudio
- OscTimeInfo

A.11 Mapping Class

DESCRIPTION

This class serves as arguments in the bridging class to determine whether data objects are bridged or not. There are get() and set() methods associated to each variable within the class in order to dynamically change the bridging within the bridge class.

PACKAGE
bridge
INHERITS THESE CLASSES
none
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES
none

A.12 XmlParser Class

DESCRIPTION

This class is responsible for importing a VSTXML file and converting its data to local variables. The local variables are then used to populate the data within the parameter objects as well as to construct the OSC Address Space for use by the plugin.

PACKAGE
xmlParser
INHERITS THESE CLASSES
none

IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES

- Document from JDOM library
- SAXBuilder from JDOM library
- File
- XMLOutputter from JDOM library
- Element from JDOM library
- List
- JTreeOutputter from JDOM library

A.13 VstAudioReceiver Class

DESCRIPTION
This class as the name suggests is used in the receiving of network audio by the plugin. It handles the ordering of audio packets and provides a buffer that is used to handle the network jitter experienced.
PACKAGE
vst.transceiver
INHERITS THESE CLASSES
VSTV10ToHost
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES

- OscAudio
- LinkedList
- AudioJitterBuffer

A.14 AudioJitterBuffer Class

DESCRIPTION
This class implements a buffer that is used to manage jitter experienced on the network. This is required as the audio traffic destination application requires fixed rate audio data and cannot handle network jitter. The implementation of a jitter buffer introduces latency into the plugin of the range of 5-20 ms.
PACKAGE
vst.transceiver
INHERITS THESE CLASSES

VSTV10ToHost
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES
none

A.15 IconFlashThread Class

DESCRIPTION
This class allows for the Gui object to use its own threads for the processing required for intensive JLabels required to give the appearance of a flashing LED. This threading affords more processing to the plugin core functions.
PACKAGE
threads
INHERITS THESE CLASSES
Thread
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES

- JLabel
- Gui

A.16 Parameter Class

DESCRIPTION
A Parameter data type class defining the variables required for objects of the parameter type.
PACKAGE
datatypes
INHERITS THESE CLASSES
none
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES
none

A.17 Audio Class

DESCRIPTION
An Audio data type class defining the variables required for objects of the audio type.
PACKAGE

datatypes
INHERITS THESE CLASSES
none
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES
none

A.18 TimeInfo Class

An TimeInfo data type class defining the variables required for objects of the TimeInfo type.

PACKAGE
datatypes
INHERITS THESE CLASSES
none
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES
none

A.19 VstParameter Class

DESCRIPTION
The VST version of the Parameter class
PACKAGE
datatypes.vst
INHERITS THESE CLASSES
Parameter
IMPLEMENTS THESE INTERFACES
none
INSTANTIATES OBJECTS FROM THESE CLASSES
none

A.20 VstAudio Class

DESCRIPTION
The VST version of the Audio class
PACKAGE
datatypes.vst
INHERITS THESE CLASSES
Audio
IMPLEMENTS THESE INTERFACES
none

Instantiates objects from these classes
none

A.21 VstTimeInfo Class

Description
The VST version of the TimeInfo class
Package
datatypes.vst
Inherits these classes
TimeInfo
Implements these interfaces
none
Instantiates objects from these classes
none

A.22 OscParameter Class

Description
An OSC version of the Parameter class
Package
datatypes.osc
Inherits these classes
Parameter
Implements these interfaces
none
Instantiates objects from these classes
none

A.23 OscAudio Class

Description
An OSC version of the Audio class
Package
datatypes.osc
Inherits these classes
Audio
Implements these interfaces
none
Instantiates objects from these classes
none

A.24 OscTimeInfo Class

DESCRIPTION

An OSC version of the TimeInfo class

PACKAGE

datatypes.osc

INHERITS THESE CLASSES

TimeInfo

IMPLEMENTS THESE INTERFACES

none

INSTANTIATES OBJECTS FROM THESE CLASSES

none

Appendix B

VSTXML file definition

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE VSTPluginProperties SYSTEM "vstXml.dtd">
<VSTPluginProperties>
  <VSTParametersStructure>
   <!--Value types:-->
    <ValueType name="Switch">
        <Entry name="Off" value="[0,0.5]"/>
        <Entry name="On" value="[0.5,1.0]"/>
    </ValueType>
    <ValueType name="OscTypeSelect">
        <Entry name="Sawtooth" value="[0,0.33]"/>
        <Entry name="Sine Wave" value="[0.33,0.66]"/>
        <Entry name="Square Wave" value="[0.66,1.0]"/>
    </ValueType>
    <ValueType name="FilterTypeSelect">
        <Entry name="LowPassFilter" value="[0,0.33]"/>
        <Entry name="BandPassFilter" value="[0.33,0.66]"/>
        <Entry name="HighPassFilter" value="[0.66,1.0]"/>
    </ValueType>
    <!--Templates: -->
    <Template name="Osc">
        <Param name="Type" shortName="type" type="OscTypeSelect" id="offset+0"/>
        <Param name="Frequency" shortName="freq" label="Hz" id="offset+1"/>
        <Param name="PulseWidth" shortName="pw" id="offset+2"/>
    </Template>
    <!--Parameters: -->
    <Param name="SynthActive" shortName="synAct" type="Switch" id="0"/>
    <!--Groups: -->
    <Group name="Global" values="offset=1">
        <Param name="Volume" shortName="vol" label="dB" id="offset+0"/>
        <Param name="Pan" shortName="pan" id="offset+1"/>
```

```
    </Group>
    <Group name="Envelope" values="offset=3">
        <Param name="Attack" shortName="A" label="ms" id="offset+0"/>
        <Param name="Decay" shortName="D" label="ms" id="offset+1"/>
        <Param name="Sustain" shortName="S" label="ms" id="offset+2"/>
        <Param name="Release" shortName="R" label="ms" id="offset+3"/>
    </Group>
    <Group name="Filter" values="offset=7">
        <Param name="Type" shortName="type" type="FilterTypeSelect" id="offset+0"/>
        <Param name="Cutoff" shortName="cutoff" label="Hz" id="offset+1"/>
        <Param name="Resolution" shortName="res" id="offset+2"/>
        <Param name="Envelope" shortName="env" id="offset+3"/>
    </Group>
    <!--Global: -->
        <Group name="Oscillator1" shortName="osc1" template="Osc" values="offset=11
        <Group name="Oscillator2" shortName="osc2" template="Osc" values="offset=14
  </VSTParametersStructure>
</VSTPluginProperties>
```

Appendix C

OSC Quick Reference