Distributed Audio Processing

Alexander Gustafson
University of Applied Sciences,
Zürich,
Switzerland,
alex_gustafson@yahoo.de

October 28, 2015

Dozent: Alexander Herrigel (alexander.herrigel@gmail.com)

School of Engineering, Abteilung Zürich
Studiengang Informatik

Abstract

In modern profesional music studios, the computer has become responsible for tasks that were previously performed by dedicated hardware. Mixing boards, effect processors, dynamic compressors and equalizers, even the instruments themselves, are all available as software. To alleviate the processing load on the CPU there is a growing market for specialized DSP coprocessors which can process mutliple channels of digital audio in realtime. These coprocessors are typically connected via Firewire or PCIe and use multiple DSP chips for the processing. This project will examine an inexpensive alternative based on standard Gigabit Ethernet and higher end Raspberry Pi clones.

Contents

Chapter 1

Introduction

1.1 Background

20 years ago the CPU was just one component of a typical music studio. It was generally used to control and synchronize other equipement such as mixing boards, multitrack recorders, synthsizers and effects processors. Today all of the other equipement exists as software, running in realtime on a CPU host. A typical music studio today is comprised of a CPU, multiple analog to digital inputs and outputs, and some DSP equiped audio processing cards.

Simliar to GPU Cards which can accelerate graphics and visualization applications, audio DSP cards can process multiple streams of high qualtiy digital audio, eliviating the load on the CPU Host Computer. Audio DSP cards typically connect to the CPU via PCI, Firewire, or Thunderbolt. Most vendors of DSP cards offer the possibility to connect several cards in parallel to increase the processing capacity.

Unlike GPU processors however, no open standard has evolved comparable to openGL or openCL. 3D graphics applications profit enormously from the interoperability that openGL offers. No such benefit is available for digital audio applications. Also, unlike OpenGL applications, audio software that is developed to run on an audio DSP card cannot be run on the CPU host. This results in vendor lock-in. The consumer that invests in an audio DSP card and software, must continue to buy from the same vendor in order to build on the the initial investment. If another vendor of DSP hardware creates a superior product, a consumer is unlikely to switch platforms if a significant investment has already been made.

10 years ago this was an acceptable compromise because DSP processors connected via PCIe could provide a significant performance increase. Today however, arm based inexpensive CPUs connected via standard gigabit ethernet could offer a competitive alternative.

1.2 Motivation

The purpose of this semester project is to design a software based music synthesiser that will run on a network of low cost banana pi devices. Limitations in polyphony will be alleviated by adding a new device to the network. In order to be compatible with existing music recording and composition applications the software will include a VST Plugin that allows music software to send MIDI commands to, and receive audio data from the software. All data communication between the VST Plugin and the Banana PI audio generation software will be handled via ethernet. The VST Plugin will send control data information such as pitch, volume, length, and other expression data. The Banana PI will stream back the generated audio data, as well as necessary metadata so the VST plugin can properly collect and prepare the audio data for the host software.

1.3 Goals

- Establishment of the Requirements
- Comparison of various Embedded Systems (Banana Pi, Adapteva, Odroid) in the context of ditributed audio processing.
- Development of an audio processing software in C++
- Development of a VST-Plugin in C++, that will pass data from a host audio production software (DAW) to the distributed processor.
- Analyse der Implementierung, um die Nützlichkeit und Skalierbarkeit zu bewerten. Es ergeben sich dadurch verschiedene Fragestellungen wie z.B. folgende: Kann die Leistung und Polyphonie durch Hinzufügen weiterer Module erhöht werden, oder wird der Kommunikations-Overhead schließlich zu gross?
- Analysis of the implementation in order to evaluate its usefullness. For instance, can more processing power and polyphony be achieved by simply adding a new processing node to the network?

Chapter 2

Background

2.1 General Background

20 years ago one had to go to a recording studio to edit a music or audio recording. There were large racks of equipment for various signal processing tasks. For example, compressors and limiters to process the dynamic range, or reverb and echo effects to give a track more ambience. 20 years ago the equipment needed for this kind of processing filled large racks. Today all of these tasks run as software plugins on the CPU.

In 1996 Steinberg GmbH, the developers of Cubase, a popular audio production software (or DAW, digital audio workstation), released the VST interface specification and SDK. [8] The VST plugin standard was special because it allowed realtime processing of audio in the CPU and it allowed other developers to programm plugins which could be run from within Cubase. The VST plugin standard quickly had widespread industry acceptance and was adopted by competing DAWs. Although alternative standards exist, VST is still the most widely adopted crossplatform standard.

The number of realtime plugins that could run on a CPU was limited by several factors, hard disc access speeds, bus speeds, amount of ram, and OS schedulers for instance [2]. User's didn't expect to be able to run more than 10 plugins at a time. Simply playing back multiple tracks of digital audio in realtime was so taxinig on the CPU that an application's graphical interface would quickly become unresponsive.

Today it's possible to playback hundreds of channels of audio and hundreds of plugins in realtime. While the performance threshhold has risen, so have user expectations. The algorithms driving today's plugins are much more complex that those from 1996. Plugins are available today that model acoustic systems or emulate the analog circutry of popular vintage synthsizers. Even though CPU performance has increased significantly, it's still easy to reach the limits, especially with the more complex high qualtiy plugins.

Serval DSP based systems exist that can alleviate the load on the CPU much in the same way that GPU accelerator cards work. Audio processing jobs are delegated to external specialized hardware via PCIe or Thunderbolt interfaces. However, these DSP

based accelerators are proprietary and expensive. Developing plugins for a DSP chip is also significatly more complex than developing for the CPU.

2.2 Realtime Audio Plugins

Music composition and production is typically done with the assistance of a digital audio workstation (DAW) application. Midi events and audio recordings are arranged, mixed, edited, and processed. In order to make changes undo-able edits are made in a non-destructive fasion, calculated dynamically in realtime during audio playback. The original audio data is always preserved. The user can change the parameters of an effect or process in realtime and experiment with various parameters without fearing that the original audio recording might be permenantly altered.

A music sequencer or audio production application will usually include several built in realtime effects that a user can apply to an audio track. In addition to the built in options all professional applications will also be able to load 3d party effect plugins. Depending on the platform and vendor one or several available plugin standard will be implmeneted, the most common standard being Steinberg's VST standard.

Regardless of the standard most audio plugins function in a similar fashion. The host application will periodically poll the plugin via a callback, providing access to the source audio data stream and expecting the plugin to return the processed data.

Audio plugins can also provide a gui to the user that allows processing parameters to be modified, saved, and sequenced as well. This might be the cutoff frequency of a low pass filter, or the delay time of a reverb effect, for example.

From a programmer's point of view audio plugins are always compiled as dynamically loadable libraries that implement a standard's specific API. The host DAW application can load then at run time and stream audio data through them [4].On the Windows platform VST plugins are compiled to Dynamic Link Libraries, on Mac OSX they are Mach-O Bundles. The native apple Audio Unit plugins are also compiled as Mach-O bundles, they have almost identical functionality, but differ in their API implementation. Other alternative plugin formats are Avid's RTAS and AAX plugin formats, Microsoft's DirectX architecture, or LADSPA, DSSI and LV2 on Linux.

Realtime Audio Plugins, as the name implies, must be able to complete their tasks fast enough to comply with realtime audio requirements. How fast is fast enough? Well, that depends how you define "real time". In audio applications, real time is defined in terms of an audio system's latency. The total delay between the time an audio signal enters the system (at the analog to digital converter for example), is processed, and leaves the system (at the digital to analog conterver) is the latency. The maximum acceptable latency is considered to be around 10ms [5]. Any higher and the latency becomes disturbing and not acceptable for live performance applications.

Any process will introduce some amount of delay. However within the audio processing function, the programmer must take care not to introduce any unneccessary or uncalculateable delays. It's also important to understand that the audio processing function is working in the context of a high-proirity system thread. Nothing in the process should have to lock or wait for resources provided by other lower-priority threads. Examples of things to avoid are memory allocations or deallocation, waiting

or locking a mutex conditional expressions inside loops that might break pipe-lining optimizations [4], or updating the graphic interface directly.

2.3 Audio Over Ethernet

Sending audio data over a network is not new. Sending audio data in ”realtime” is also not new. The IETF (Internet Engineering Task Force) RFC 3550 Describes the Real-time Transport Protocol for delivering audio and video in real-time over IP networks. RTP is used as the basis of most media streaming and video conferencing applications.

Other very recent specifications such as AVB[1] and AES67[2] build on top of RTP and add more mechanisms to guarentee acurate timing and synchronization across a network for professional audio applications. The synchronisation is important in these standards because they are concerned with driving audio hardware attached to different hosts on a network.

Hardware synchronization is not relevent to this project because we are not concerned with external audio hardware. The goal of this project it to utilise external CPUs as audio coprocessors connected via gigabit Ethernet. Even so, the AVB and AES67 standards offer many insights into how to optimize data transmission for low latency applications and they also offer a proof-of-concept that it is possible. The AES67 defines guidelines that can achieve latencies well below 1 ms for hundreds of simultaneos channels of high quality audio. This is much faster than the legacy PCI and Firewire rates used in many DSP based coprocessing systems [1].

2.4 Single Board Computers

The popularity of the Raspberry Pi has spurned a whole industry around single-board computers (SBCs). Based on hardware used in mobile phones, these small low power devices are extremely popular because they are inexpensive and easy to use. The biggest advantage of SBCs compared to other embeded devices ist that they can run the Android and Linux operating systems, allowing them to be programmed using the same tools available on desktop computers.

Recent higher-end SBCs even come equiped with gigabit Ethernet and Dual and Quad Core CPUs running at rates well over 1GHz. If we compare these systems to the 450MHz G3 PPC systems that the first VST Software was available for we can expect that the newer high-end SBCs should be excellent audio coprocessors.

2 SBCs are worth special consideration because they potentially offer even better performance as audio coprocessors. The Parallella Board has a 16-Core Epiphany co-processor that could be used to perform audio processing in parallel. Standard frameworks such as OpenCL, MPI, or OpenMP can be used to target the Epiphany cores.

[1] Audio Video Bridging refers to a set of IEEE standards that allow time-synchronized low latency streaming services

[2] AES67,created by the Audio Engineering Society, defines standard that allow existing low latency streaming systems to interoperate. AES67 does not define any new technologies but attempts to set a lowest common denominator by which existing standards can be compatible.

The Odroid-XU4 SBC includes a Mali-T628 GPU coprocessor which is also OpenCL compatible. Both are available for under $100.

Programming audio processing routines as OpenCL kernels might be considerably more complex than in C++, but OpenCL offers vendor-independent access to GPGPU computing and has the added benefit that it can also be used on a CPU without GPU acceleration [3].

Investigating these, and other OpenCL enabled SBCs might be an interesting followup project.

2.5 Virtual Analog Synthesis

Virtual analog synthesis is the term used to describe the emulation of analog synthsizers of the 60s and 70s digitally in real time. The complexity and goals of an emulation can vary. Some emulations go so far as to simulate the actual electronic components of vintage synthesis circuits, others just model ths signal flow loosly.

Regardless of the type of emulation, virtual analog systhesis has two primary concerns, latency and aliasing. The problem of latency has already been described above. Any processing will introduce a delay in the signal, the complexity of the processing can increase the delay, or use more CPU cycles. Aliasing is audible distortion introduced by signals that have a higher frequency content than the sampling rate of the system allows for.

Analog synthsizers usually employ subtractive synthsis. One or more sound generators or oscillators would create signals with particular harmonic qualities. These signals are passed through filters that ”subtract” frequencies from the signal. The oscillators, filters, and amplitude can be modulated. Figure 2.1 is a simple block diagram of a typical subtractive synth voice.

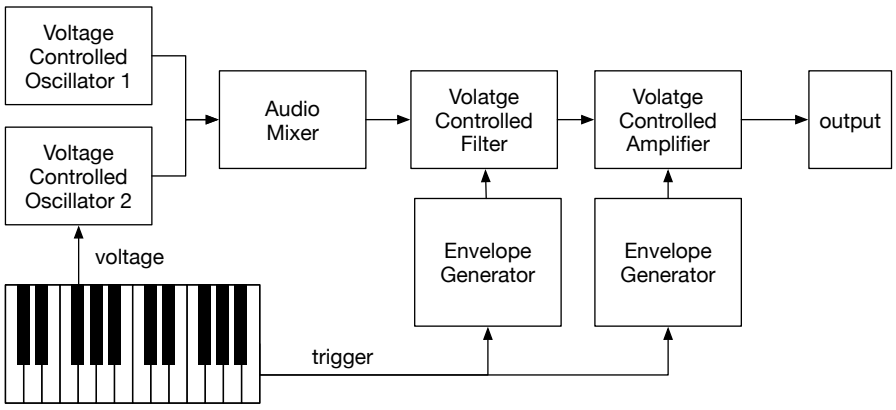

Figure 2.1: Block Diagram of a Subtractive Synthesis Voice

The voltage controlled oscillators generate simple waveforms at the pitch core-

sponding to the note played on the keyboard. The user can typically choose between some combination of sawtooth, squareware, or triangle waveforms. The frequencies or timbre of the waveforms can then be modulated by the following filter and amplitude blocks.

One's first impression might be that modeling the oscillator would be simple. A digitally generated squarewave or sawtooth waveform should be trivial to implement. The 5kHz sawtooth waveform for example, would have a period of 8.82 samples when generated in a 44.1kHz audio environment. So the waveform would increase linearly from -1.0 to 1.0 over a length of 8.82 samples, then jump back to -1.0 and cycle through again. What does 0.82 sample mean in a discreet digital system? Figure 2.2 illustrates the problem with a trivial implementation. The left column shows a portion of an idealized 5kHz sawtooth waveform and the corresponding frequency content. Above the 5kHz fundamental frequency are harmonics that will be audible well beyond 100kHz. The right column shows the same portion of a 5khz sawtooth waveform in a 44.1kHz discrete environment. The waveform itself is distorted and the higher harmonics are visible reflected back from the 22.05kHz nyquist limit.

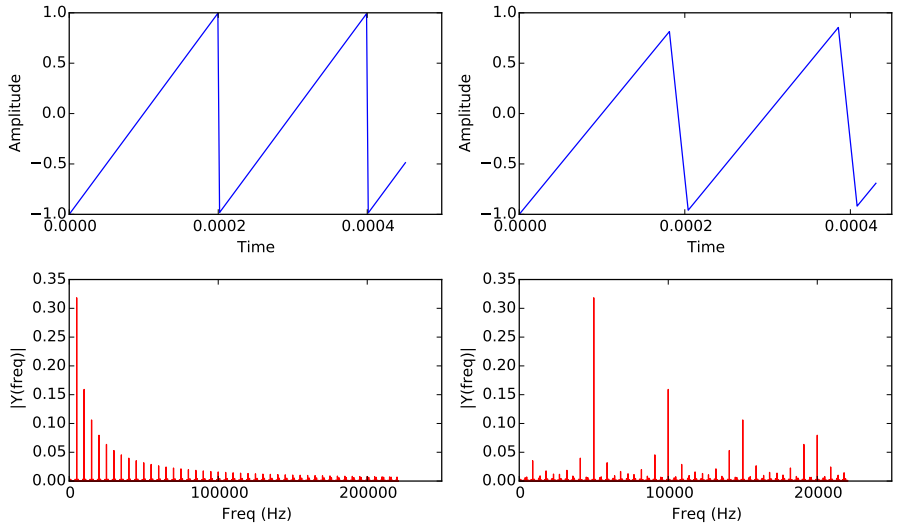

Figure 2.2: Ideal and Aliased 5kHz Sawtooth waveform

There are various methods to eliminate or reduce aliasing. The most effective is to generate the harmonics using a series of sin waves up to the nyquist limit or half of the system's sampling rate, but this also very processing intensive. Less CPU intensive strategies involve using oversampling, bandlimiting, or other alias-supressing methods [7].

Chapter 3

Requirements

3.1 General Requirements

The Application has two components, the audio plugin hosted on the main CPU machine, and the processing nodes which run on networked SBC devices. The audio plugin forwards midi control and audio data to the processing nodes. The nodes stream the processed audio data back to the audio plugin, which in turn streams it back to the host audio application. The total round-trip time, including processing, should not exceed 10ms. This is the maxium allowed latency for live sound applications. [5]

The application must be self contained and work without the user needing to install any system libraries, frameworks or servers.[1]

3.2 Distributed Processing

In order to lighten the load of the host CPU we are interested in distributing real-time audio processing jobs to remote CPUs connected by gigabit ethernet. On the host CPU an audio plugin functions as the master node. To the host DAW, the distribution of jobs should be completely transparent. The master node receives audio and control data from the host application and returns the results just like any other audio plugin.

In contrast to other distributed processing environments where large data sets are parceled out to worker nodes, the plugin master is only given access to the audio data in small buffers as it is needed. The plugin then has a very small amount of time to process the data and pass it back to the host application. This puts some limits on how processing jobs can be distributed.

There are various degrees to which processing jobs can be distributed. Each plugin instance could send it's entire job to one networked node as in figure 3.1. Each processing block in a plugin could send it's partial job to a networked node as in figure 3.2. In the case of a virtual synth plugin each voice performed could be distributed to it's own node as in figure .

[1] The only exception might be ZeroConf/Bonjour on Linux or Windows. See Appendix

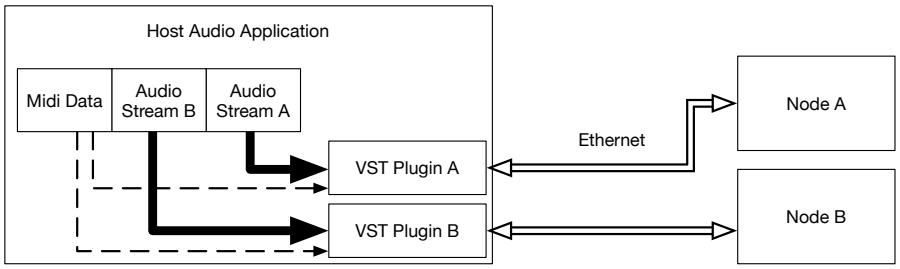

Figure 3.1: Each Plugin Distributes to One Node

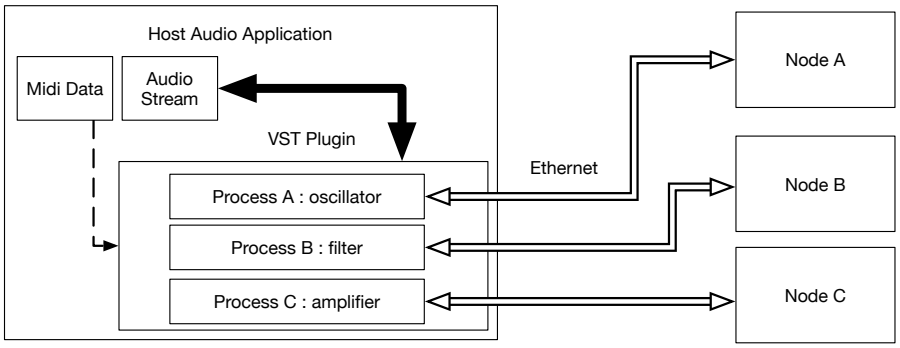

Figure 3.2: Each Process Block Distributes to a Node

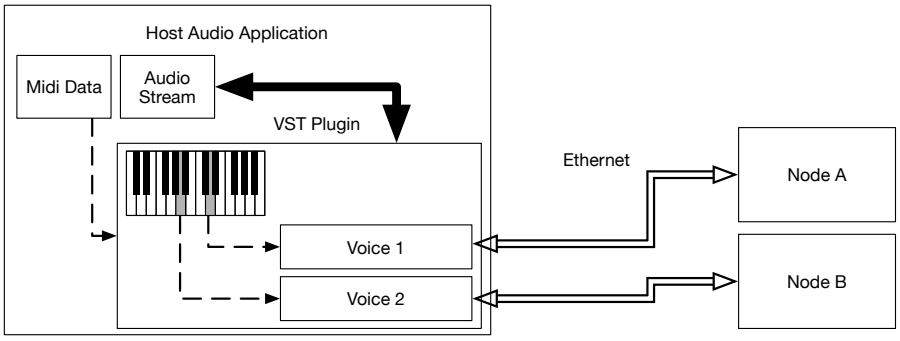

Figure 3.3: Each Synth Voice is Distributed to a Node

For this project the second option will be implemented for the sake of testing, al-

though it might not be the most efficient implementation.

3.3 Audio Plugin

The plugin must appear to the host DAW application as a normal audio processing plugin, managing the distribution of tasks to networked nodes internally. The audio plugin does not run in an isolated environment. It will be running in tandem with its host application which might be running any number of other plugins. Therefore every care should be made to reduce it's load on the CPU.

Audio plugins typically have a state that will include parameter settings controlled by the audio host or the user. This is important in order to allow an audio plugin to hand over an on-going process to a newly connected node. It could also be used to allow a networked node to be responsible for the processing of more that one audio plugin by switching it's active state accordingly.

The audio plugin has the following requirements:

- runnable as a realtime VST audio plugin in a standard DAW application
- locate and connect with one or more processing nodes on the network
- forward midi and audio data from the host audio application to the networked nodes
- receives audio data from the networked nodes and streams this back to the host application
- in the absence of a coressponding node on the network perform the audio processing locally
- the audio plugin must forward it's current state to the node

3.4 Remote Processing Node

The processing nodes are applications that run on the networked SBC devices. Depending on the type of parallelisation implemented in the audio plugin the processing nodes could be delegated the entire processing job of the corresponding plugin or just a part of the job. For this project the processing nodes will be implemented as a ”bank” of processors loaded into a parent application. The parent application will implement a socket listener that monitors an array of sockets for incoming requests, then call the callback of the node that is responsible for that particular socket. The availability, type, and location of node and it's corresponding socket will be broadcasted over the network via bonjour/zeroconf.

The processing node should be stateless, each cycle of the audio processing algorithm should only take the state data of the corresponding packet into account, and updates to the state must be sent back as state data to the audio plugin. This is to ensure

that if a node losses connectivity the state can be retrieved and another node can take over. It also has the added benefit that one processing node could be able to process jobs for several instances of a particular plugin.

The remote processing nodes have the following requirements:

- broadcasts its availability and location on the network via bonjour/zeroconf
- accepts session initiated by the audio plugin
- accepts controll data from the audio plugin
- processes incoming audio data and midi data from the audio plugin
- streams audio and midi data back to the audio plugin immediately

3.5 Software Requirements

In realtime audio applications timing is critical. This may sound obvious, but to a programmer it means giving up many of the comforts of modern programming made available working with high level interpreted languages such as java or python. Most audio application interfaces and libraries such as the VST SDK are for C/C++.

Professional audio applications generally run on Mac OSX or Windows Operating Systems, therefore the audio plugin must be compatible on these systems. The processing node will be run on SBC devices which typically run with a Linux based OS. Yet both applications should share their codebase since there is alot of crossover of responsibility between the audio plugin and the nodes.

There are many C++ libraries and framework that address the issue of cross platform compatibility while also giving the programmer access to high level constructs like smart pointers and reference counted objects that make C++ programming easier.

3.5.1 Evaluated C++ Frameworks

Boost is the most popular cross platform C++ frameworks. Many of it's features have been added to the C++11 standard library. Other frameworks like Cinder and Open Frameworks offer many high level features to quickly build cross platform media rich interactive applications. Two libraries of special note offer specific features to build cross platform audio applications and plugins, JUCE and WDL. Of these two Juce has a much larger community of users (including vendors of dsp based audio coprocessors).

Software Framework Criteria:

- Crossplatform for OSX, Linux, and Windows
- Offers high level constructs like smart pointers
- Support for crossplatform audio integration

- Should be well documented and have an active community
- Support for crossplatform network streaming

Framework	High Level Utilities	Audio Utilities	Network Utilities	VST Utilities	Community
JUCE	ja	ja	ja[2]	ja	gross
WDL	ja	ja	nein	ja[3]	klein
Open Frameworks	ja	ja	ja[4]	nein	gross
Boost	ja	nein	ja	nein	gross
Cinder	ja	ja	ja	nein	klein
LibSourcey	nein	nein	ja	nein	nein
Qt	ja	nein	ja	nein	gross

Based on the criteria comparison above and experience in previous projects JUCE was chosen for this projects implementation.

[2]basic socket management classes
[3]enabled using one of the additional iplug libraries
[4]the ofxNetwork addon allow simple management of TCP or UDP sockets

Chapter 4

Implementation

4.1 Architecture

Figure 4.1 illustrates a high level overview of the distributed audio proessing environment. The user of the system interacts with the audio production software running on the host CPU and can choose to activate an audio processing plugin on a specific audio track. A single audio plugin might contain one or several individual processors. Processors that are enabled for distributed processing will search for a corresponding processor on a network SBC device.

The devices are networked via gigabit ethernet. It is assumed that the network is not being used for any other significant traffic.

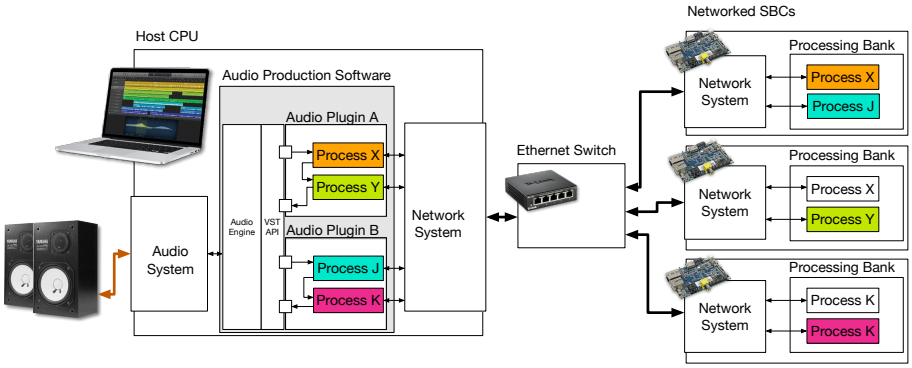

Figure 4.1: Architectural Overview

4.1.1 Host CPU

Figure 4.2 zooms in on the components involved in realtime audio on the CPU. The audio hardware needs to provide a constant stream of data to it's digital to analog

converters. It does this by periodically polling the operating system via hardware interrupts. The requested buffer size can be as small as 32 samples and the polling intervals less than 1 ms depending on the hardware and drivers.

The operating system provides an abstraction layer in the form of an API to the application software. This gives the application software a single API to interface with regardless of the brand of audio hardware and drivers installed.

The VST API is another abstraction layer that offers a unified interface for plugin vendors, regardless of the OS. But VST is not the only plugin API. The Juce library offers it's own plugin API, which is simpler and abstracts away the differences between various plugin APIs.

The requests for data, manifested as interrupts triggered by the audio hardware, are passed on through the system to the DAW application by means of callbacks that the application has registered with the system. The DAW application in turn polls all the active plugins for data through thier defined VST callback functions.

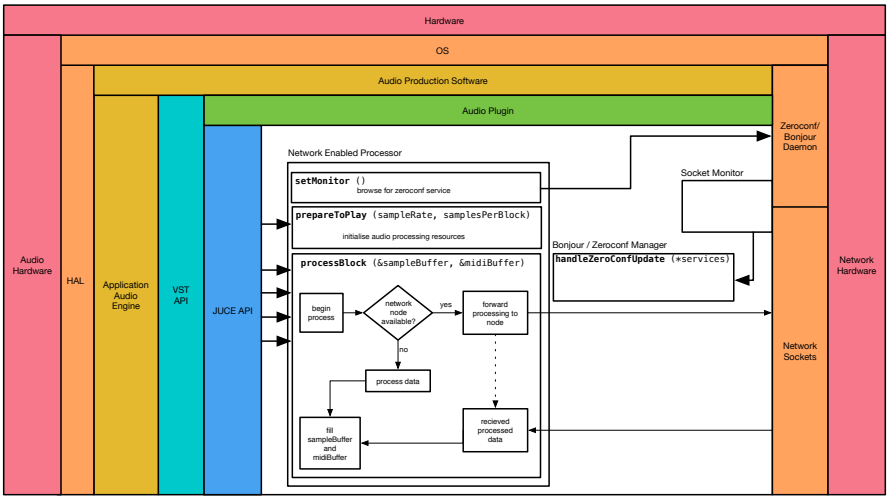

Figure 4.2: Host CPU Overview

The plugin implemented for this project has several network enabled processors, in figure 4.2 only one is shown as an example. When the plug in the instantiated by the host software, it in turn instantiates each of it's processors. The processors in turn immediately calls the operating system's Bonjour / Zeroconf daemon with a request to browse the network for a specific service corresponding to a matching processor node. The request includes a socket by which the daemon will notify the processor of machtes it has found.

The Bonjour / Zeroconf daemon notifies the audio plugin by means of the socket, at which point the plugin's Bonjour / Zeroconf manager scans through the list of matching services to find on that is available. The plugins activeNode member holds a reference

to the matching network service.

When a request for audio data is passed from the audio production software to the plugin, it does so by calling the processBlock function of the plugin. The plugin in turn calls the processBlock of each of its processors. In figure 4.2 this is simplified showing the processBlock of just a single processor. The processBlock function is given a reference to the current audio buffer and midi buffer to process. The audio buffer contains the individual samples for each channel to be processed as float values. The midi buffer contains performance data including the timing and pitch of notes to be played.

Within the process block the processor checks if a activeNode is available. If so it immediately forwards the buffers of audio and midi as well as it's own state information to the activeNode and awaits the response. When the response arrives the data is copied back to the buffers. The audio production software continues to poll the following plugins until all processing is complete. The resulting buffers are provided to the audio hardware, via the HAL API services.

4.1.2 Networked SBCs

Figure 4.3 illustrates the components of the networked SBC devices. A processing application can hold any number of processors that each register their services to the Zeroconf / Bonjour daemon installed on the devices operating system. The Zeroconf / Bonjour daemon broadcasts the availability, type and the port numbers of each of the processors available.

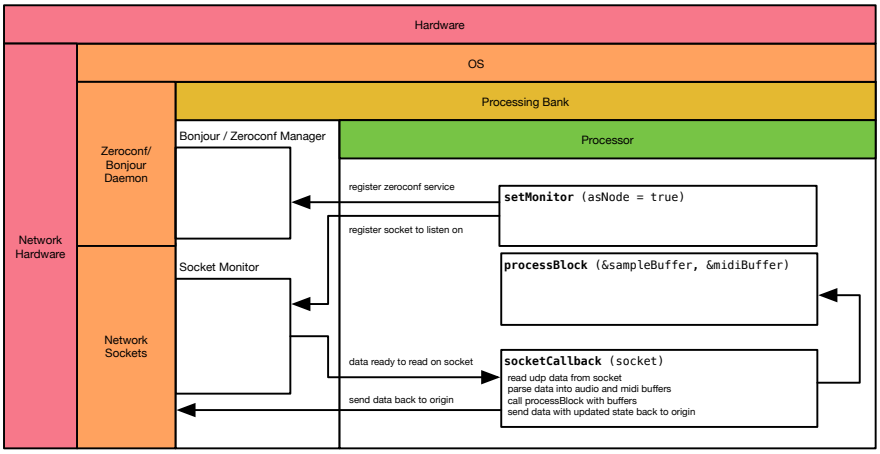

Figure 4.3: SBC Processor Overview

The processor also registers an open socket and itself as that socket's listener at the appliation's socket monitor module. The socket monitor holds an array of sockets and

a reference to each listener. It performs a select function on the array of sockets and waits. When data arrives at any of the sockets, the select function is unblocked and the socket monitor notifies the corresponding listener via callback that data is available.

The corresponding processor is notified via the callback function, it reads the data, parses the audio and midi buffers, then perform a processBlock function on the buffers. The results and updatad state information is that passed back to the origin.

4.2 Software Components

The application is divided into modules that each manage their own responsibilities. Each module was developed in a pseudo test-driven methodology. The tests were not necessarily developed before the corresponding code was written, but definitely in conjunction thereof. This resulted in stable code that could be tested in isolation from other components at every code iteration.

Where applicable, interaction with the class is defined in an associated ”listener” class with specific callback methods that a client is expected to implement. A client class can then inherit from and override the listener class.

4.2.1 Socket Monitor

The Monitor class was created to handle sockets efficiently. Client classes can implement the FileDescriptorListener class can be extended by clients that wish to be notified when data is available to read from on a specific socket.

The Monitor class has a thread that blocks on a system select() call. The select is given an array of sockets as file descriptors. When one of the sockets is ready to read from the select call unblocks. The thread loops through the array to find the socket that is ready, and notifies the corresponding listener.

One of the sockets passed to the select() call is the Monitor's own control_listener socket. The Monitor class can send a signal to this socket when ever it need to wake the blocked thread from the select() call and update it's state. The Monitor class does this whenever a new client registers or removes itself, and when the Monitor class needs to shut down.

4.2.2 ZeroconfManager

Zeroconf (short for Zero Configuration Network and also know as Bonjour on OSX) is a specification that allows services to broadcast their availability and location on a network. Printer and Multimedia devices use Zeroconf in order to allow networked computers to easily find and use their services.

Bonjour on OSX and the compatibile features implemented by Avahi on Linux define a callback based API that interfaces to a bonjour or avahi daemon running on the OS. The Bonjour API is in C. The ZeroConfManager class encapsulates communication to the bonjour or avahi deamon in an object oriented manner. Clients that want to interface with this class must extend and override the ZeroConfListener class, they register themselves with the ZeroConfManager..

When new services are registered on the network, or removed from the network, the corresponding listeners are notified with a list of all active services. If they are currently connected with a service that is no longer available, it is the clienst responsibility to disconnect from that service.

In order to resolve a service to a specific IP address and port number several asynchronous calls must be made to the Bonjour daemon, saving and updating the state of each result between calls. The ZeroConfManager hides this complexity from it's clients and only notifies them when all the information is complete.

4.2.3 DiauproMessage

The DiauproMessage class manages the serialisation and deserialisation of data to datagram packets. The datagrams themselves are comprised of a fixed length header, and a varible length payload that contains the audio, midi, and state information.

The header is defined as follows:

```
struct diapro_header {
    uint16 sequenceNumber;
    uint16 numSamples;
    uint16 numChannels;
    double sampleRate;
    uint16 audioDataSize;
    uint16 midiDataSize;
    uint16 stateDataSize;
    double cpuUsage;
    double totalTime;
    double processTime;
    uint32 tagNr;
};
```

The DiaproMessage class makes an effort to use existing allocated memory when possible. Instances of DiauproMessage should not be created or allocated in the thread that calls the audio processing routines. Instead an instance of the DiauproMessage should be preallocated with enough memory to store the largest UPD Datagram possible (64 kilobytes). The instance can be reused for each cycle and memory will not have to be reallocated during timing critical processes.

4.2.4 DiauproProcessor

The DiauproProcessor class is the base class that other Processors should inherit from. The DiauproProcessor inherits from the JUCE AudioProcessor class which encapsulates all the functionality of various audio plugin formats. Classes that inherit from AudioProcessor can be wrapped into a VST plugin directly.

A class that inherits from DiauproProcessor must extend the localProcess and getServiceTag functions. The localProcess function performs the audio processing. getServiceTag returns a string that will be used to broadcast and browse for the specific service on the network.

Classes that inherit from DiauproProcessor can be instantiated and setup to run in the audio plugin or in the processing node. When run as plugin code the will browse for corresponding services on the network. If they dont find one then the localProcess function will be used locally. It they do find a networked service on all incoming audio and midi data will be forwarded to that service.

When run in a processing node, classes that inherit from DiauproProcessor will register themselves on the network and wait to incoming processing requests.

Examples of classes that inherit from DiauproProcessor are DiauproVCOProcessor and DiauproVCAProcessor, described below.

4.2.5 DiauproVCOProcessor

The DiauproVCOProcessor extends the base DiauproProcessor and implements the oscillator block of a virtual synthsizer. The only functions from DiauproProcessor that need to be overridden are localProcess, getState, getStateSize, and getServiceTag.

localProcess is implemented as follows:

```
1  void DiauproVCOProcessor::localProcess(AudioSampleBuffer &buffer,
2                                         MidiBuffer &midiMessages,
3                                         void* state)
4  {
5    processState = *(vco_state*)state;
6    int sampleNr;
7    int nextEventCount = -1;
8    MidiBuffer::Iterator midiEventIterator(midiMessages);
9    MidiMessage nextEvent;
10   bool hasEvent;
11
12   for(sampleNr = 0; sampleNr < buffer.getNumSamples(); sampleNr++)
13   {
14     if(nextMidiEventCount < sampleNr)
15     {
16       hasEvent = midiEventIterator.getNextEvent(nextEvent,nextEventCount )
17     }
18     if(hasEvent && nextEventCount == sampleNr)
19     {
20       if(nextEvent.isNoteOn())
21       {
22         processState.voice_count++;
23         processState.frequency = MidiMessage::getMidiNoteInHertz(nextEvent.getNoteNumber());
24         double cyclesPerSample = processState.frequency / getSampleRate();
25         processState.step = cyclesPerSample * 2.0 * double_Pi;
26
27       } else{
28         processState.voice_count--;
29       }
30     }
31     if(processState.voice_count > 0)
32     {
33       const float currentSample = (float) (sin (processState.phase) * processState.level);
34       processState.phase += processState.step;
35       for(int i = 0; i < buffer.getNumChannels(); i++)
36       {
37         float oldSample = buffer.getSample(i, sampleNr);
```

```
38              buffer.setSample(i, sampleNr, (currentSample + oldSample)*0.5);
39          }
40      }
41    }
42  }
```

When the function is called it is passed a reference to the current audio sample buffer to be filled, a buffer of midi events for the corresponding time frame, and a pointer to a block of data that can be used to hold and state that needs to be persisted between calls.

If this function is called from a node processor then the audio, midi, and state data has been extracted from a DiauproMessage sent as a UDP packet from the master audio plugin.

In the example above a sin wave is create by calculating the sin value for each element in the audio sample buffer.

4.2.6 DiauproVCAProcessor

The DiauproVCAProcessor modulates the amplitude of a signal over time. VCA is short for Voltage Controlled Amplifier, this referes to the processing block in a virtual synthsizer that is responsible for the ”shape” of a sound. A bell sound, for instance, will have a fast and loud initial tone that gradually decays to silence over a long period of time. In contrast, a bowed string sound, like a violin, might have a slow rising initial tone that suddenly ends. The VCA is responsible for the shaping of the amplitude of a sound.

4.2.7 DiauproPlugin

The DiauproPlugin is the plugin that runs in the host DAW application. It passes audio data from the host DAW application to three internal processors in series. The first processor is a ”Null Process Block” that does no actual audio processing but is usefull for gathering data regarding timing. The second processor is the VCO block that generates a tone based on recieved midi data. The third block is the VCA that shapes the amplitude of the tone over time.

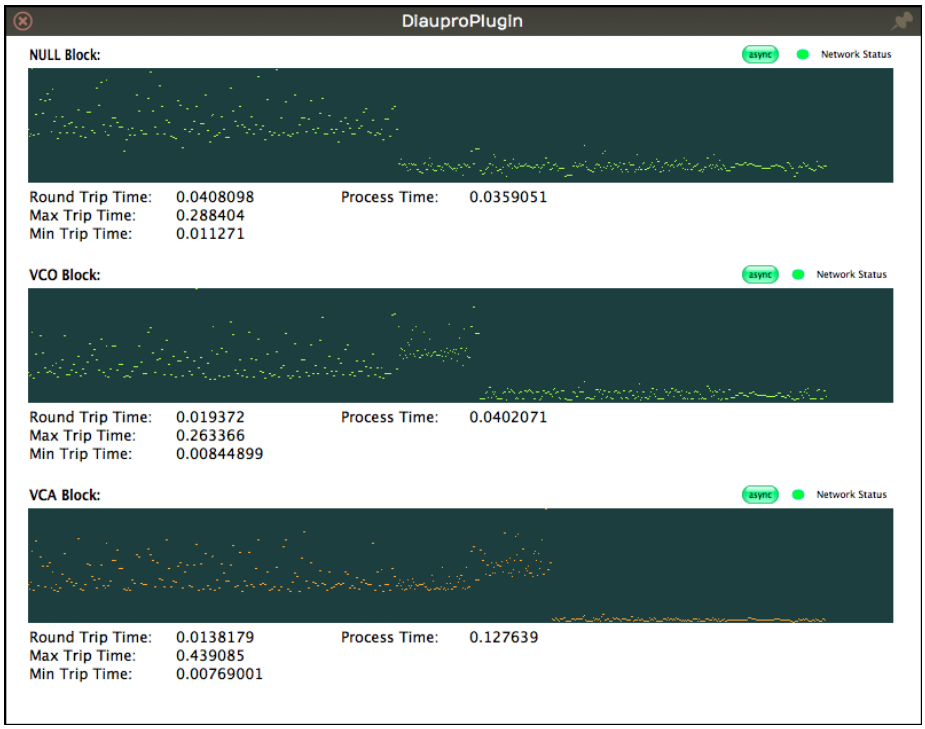

Figure 4.4: Screenshot of Plugin GUI

The audio plugin's user interface provides feedback to the user regarding the time and operation of the internal processors. Each processor can perform the calculations locally on the CPU or send it's data to a networked node to be processed. A "Network Status" indicator shows if a networked node is performing the processing. The total time that a processor requires to perform it's operations is displayed as the "Round Trip Time". The "Process Time" displays the amount of time that a remote SBC required to perform the processing.

A button is provided that allows the user to switch to an asynchronous processing method. When this is activated each process checks for the availability of the SBC's response, but if it is not yet available is does not wait. Instead the process simply returns empty data back to the plugin. By the time the next buffer is requested by the DAW application the first response is available and the processor can immediately return it without waiting. This method creates a delay in the signal of one buffer cycle, but has the advantage the it does not block the CPU while waiting. The delay can usually be compensated for in the DAW application.

In addition to providing a graphical user interface the plugin also saves timing data to a text file.

Chapter 5

Conclusion

5.1 Performance Evaluation

An Audio Plugin runs in an environment with many other components, sharing CPU resources. Processing of audio data must be completed within discrete time intervals dictated by the audio hardware. Typical audio sampling frequencies are 44.1kHz, 48kHz, 88.2kHz and 96kHz. Using 44.1kHz as an example, this means that all the calculations required for a single sample must be completed within 0,023 ms. Interrupts would be received from the audio hardware at intervals of 0.023ms as well, and this would put too much of a strain on the Operating System of the CPU though. Instead, requests for new audio data are bundled into buffers of samples. The size of the buffers is a parameter that the user can modify, it's usually set at 512 samples, but can be as low as 16 samples.

Increasing the buffer size, increases the amount of time that the CPU has to process audio data. This introduces latency into the system though. A buffer size of 512 samples equates to a latency of 11.60ms. A 16 sample buffer size equates to 0.36ms.

If a single plugin requires 1.0ms to complete its processing then it will not finish in time if the buffer size is set too low. If the buffer size is just high enough, then there still might not be enough CPU resources left for other plugins to complete their tasks.

5.1.1 Synchronous Performance

This project offloads the processing to external SBC devices. However, no resources are saved if the audio plugin is blocked while it waits for the results from the SBC devices. Since the external SBC devices are slower than the main CPU, the processing time will take longer. Adding the time it takes to serialise and deserialise the Datagram packet at each end will degrade performance even more.

Figure 5.1 illustrates the problem in more detail.

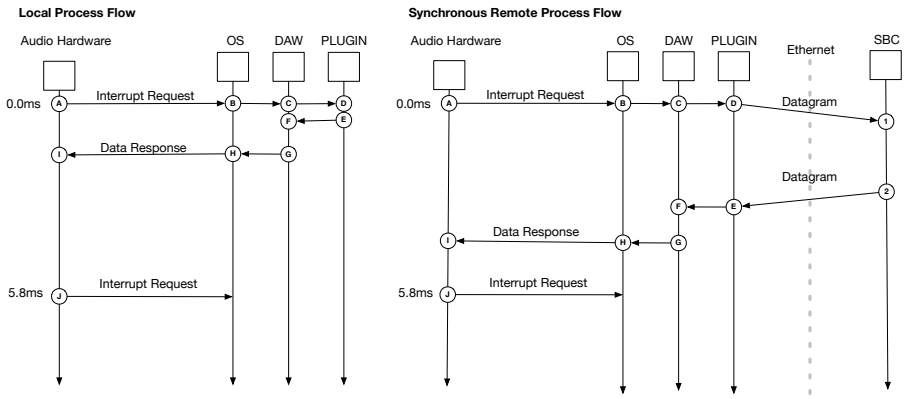

Figure 5.1: Local vs Remote Synchronous Processing

The example shows the audio hardware polling the operating system in intervals of 5.8 ms. This corresponds to a buffer size of 256 samples. The time interval between states D and E represents the time it takes for a plugin to process a buffer of 256 sample. The DAW performs other necessary audio processing functions in the interval between F and G. After states G and H the DAW and OS are free to be able to do other things, like update the GUI.

With synchronous remote processing the state E is blocked until states 1 and 2 have completed. If that time is significant then the ability of the DAW and the OS to perform other critical tasks is impaired.

buffer size	buffer time (ms)	rtTime (ms)	pTime (ms)	tTime (ms)	% of buffer time
64	1.451247	0.574672	0.015857	0.558815	39.59
96	2.176870	0.575419	0.015851	0.559568	26.43
128	2.902494	0.59001	0.016519	0.573491	20.32
192	4.353741	0.67902	0.019013	0.660007	15.59
256	5.804988	0.707267	0.020352	0.686915	12.18
512	11.60997	0.707905	0.026348	0.681557	6.097

Table 5.1: Measured Times for Synchronous Processing

Table 5.1 shows the measured times for various buffer sizes for the synchronous implementation that does no actually audio processing. Only the preparation and transport times are taken into account. An audio system running with a buffer size of 64 samples has 1.451ms to complete all tasks, the ”buffer time”. ”rtTime” is the meassured total round trip time. This corresponds to the time between states D and E of the Synchronous Remote Process Flow diagram in Figure 5.1. ”pTime” is time interval be-

tween states 1 and 2 on the SBC device, the time spent processing the data. In this case there was no processing so this is only the measures the time spent deserialiseing and serialising the datagrams into audio and midi data. ”tTime” is the ”rtTime” subtracted by the ”pTime” and is the packaging overhead required to send and receive data.

The last column, ”% of buffer time”, shows what percent of the available time was used for the plugin. At a buffer size of 64 samples 39.59% percent of the total time available was spent by the plugin. This does not leave much time for other processes on the CPU. For a buffer size of 512 samples the percentage is much lower. The total system latency is slightly higher that the stated 10ms maximum though. The synchronous implementation has not reduced the processing load on the CPU at all.

5.1.2 Asynchronous Performance

Instead of waiting for a response from the SBC device, as in the synchronous method above, the asynchronous method checks if a response is available. If so, return that data, if not immediately return without data. The initial request for audio data would return empty data, but by the time the second buffer is requested, the first respose from the SBC device might have returned. In both cases the only time spent by the audio plugin itself is the time is takes to serialise, deserialise, and transport the data. These are the intervals between the states D and E and M and N represented in figure 5.2.

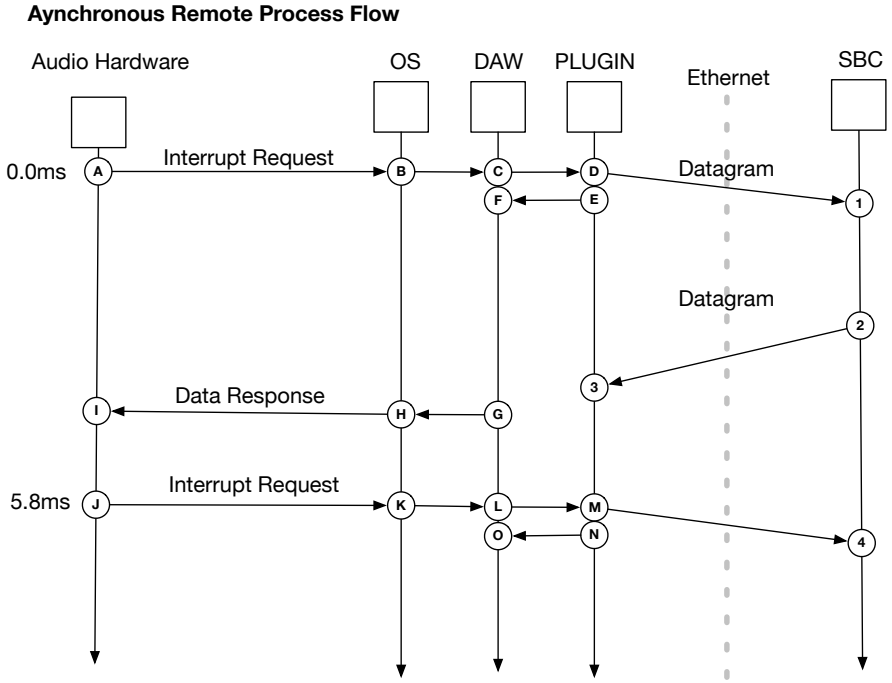

Figure 5.2: Asynchronous Processing

By the time the second cycle begins (J in figure 5.2) processed data from the first cycle has returned. The time between events M and N is no longer proportional to the amount of time needed to actually process the data, only to the time it takes for sending and receiving. Table 5.2 shows the measured times.

buffer size	buffer time (ms)	rtTime (ms)	pTime (ms)	tTime (ms)	% of buffer time
64	1.451247	0.031683	0.015633	0.016050	2.18
96	2.176870	0.046585	0.016076	0.030509	2.14
128	2.902494	0.034745	0.016712	0.018032	1.19
192	4.353741	0.052279	0.018800	0.033478	1.20
256	5.804988	0.065752	0.019646	0.046105	1.13
512	11.60997	0.062939	0.019708	0.043230	0.54

Table 5.2: Measured Times for Asynchronous Processing

In the table above ”rtTime” is no longer the actual round trip total processing time.

Since data is already available at the time the interrupt request is triggered, the plugin can return it's data immediately. This comes at the cost of a delay equal to one interrupt cycle, but the user can set the audio system's buffer size low enough that this is well below the 10ms limit, without needing to use more than 1.2% of the processing time available.

The asynchronous method brings a real benefit to distributed processing in terms of lighting the load on the CPU. The price of the benefit though is an increase in the latency of the plugin. The processed audio data is always one delayed by one buffer cycle. Another disadvantage is that if several distributed processors are chained in series, as is the case in the demo application, then the latency is cumulative. This results is a total latency that is the "buffer time" multiplied by the number of processors in the plugin.

5.1.3 Potential Optimisations

There are two areas of the application where unnecessary operations are performed and could be optimized. The first is the deserializsation of data when it is returned from the SBC. If serveral processes in the Plugin are chained in series, then the returned DiauproMessage could be sent directly to the following process without unnecessarily deserialising and the reserialising the data in between steps. This would be a relatively simple optimization to implement.

The second optimisation is similar but more complicated to implement. In a chain to processes the data would not need to be returned to the master plugin between each step, instead it could be sent to the next node directly. If the next node is located on the same SBC device then an additional hop over ethernet could be skipped as well. In addition serialisation steps between nodes could also be skipped if the nodes performed the operations directly on the data in the DiauproMessage. For this the DiauproMessage would need to be extended to include routing information so that each node knows the next destination point.

Although more complex, the second optimization would have the added benefit that the latency would no longer be a multiple of the full cycle time between interrupt calls. The final data could be gathered after a single cycle.

5.2 Summary

The asynchronous implementation offers definite potential. The latency seems to be identical to commercial DSP based systems [6]. This is interesting as it suggests that the DSP and SBC based systems share some limitations and that these cannot simply be overcome by simply connecting a more powerful DSP or SBC device. The real barrier is getting the data back from the device as fast as possible without blocking the audio thread. If the audio thread is blocked for 10% or 30% of the time is irrelevant. By resigning to a method that skips a full cycle of the audio thread there is no need to block it. The difference in bandwidth between Thunderbolt, PCIe, or Gigabit-Ethernet connectivity only effects that number of plugins that can be run simultaneously but not the latency.

This is encouraging, especially considering the expandability of the SBC based system, and the compatibilty of the codebase. Two definite advantages that the SBC system has over DSP based systems.

Bibliography

[1] Nicolas Bouillot, Elizabeth Cohen, Jeremy R. Cooperstock, Andreas Floros, Nuno Fonseca, Richard Foss, Michael Goodman, John Grant, Kevin Gross, Steven Harris, Brent Harshbarger, Joffrey Heyraud, Lars Jonsson, John Narus, Michael Page, Tom Snook, Atau Tanaka, Justin Trieger, and Umberto Zanghieri. Aes white paper: Best practices in network audio. *J. Audio Eng. Soc*, 57(9):729–741, 2009.

[2] Eli Brandt and Roger B Dannenberg. Low-latency music software using off-the-shelf operating systems. 1998.

[3] Wolfgang Fohl and Julia Dessecker. Realtime computation of a vst audio effect plugin on the graphics processor. In *CONTENT 2011, The Third International Conference on Creative Content Technologies,*, pages 58–62, 2011.

[4] Vincent Goudard and Remy Muller. Real-time audio plugin architectures, a comparative study. pages 10, 22, 9 2003.

[5] AES Standards Committee Gross, Kevin. *AES standard for audio applications of networks - High-performance streaming audio-over-IP interoperability*. Audio Engineering Society, Inc., 60 East 42nd Street, New York, NY., US., 2013.

[6] Hugh Robjohns. Review : Universal audio apollo, 2012. [Online; accessed 06-August-2015].

[7] Vesa Välimäki and Antti Huovilainen. Oscillator and filter algorithms for virtual analog synthesis. *Computer Music Journal*, 30(2):pp. 19–31, 2006.

[8] Wikipedia. Virtual studio technology — Wikipedia, the free encyclopedia, 2015. [Online; accessed 05-August-2015].

Chapter 6

Appendix

6.1 Compiling the Source Code

6.1.1 Download and Build the DiauproPlugin on OSX

Download the JUCE Library:

```
> cd $HOME
> git clone https://github.com/julianstorer/JUCE.git
```

Download and install the DrowAudio Juce Module Extensions:

```
> cd $HOME/JUCE/modules/
> git clone github.com/drowaudio/drowaudio.git
```

Install the VST SDK Library:

```
> cd $HOME/
> mkdir SDKs
> cd SDKs/
> curl -O http://www.steinberg.net/sdk_downloads/vstsdk365_28_08_2015_build_66.zip
> unzip vstsdk365_28_08_2015_build_66.zip
```

Download the DiauproProject:

```
> cd $HOME/Documents
> git clone https://github.com/alexgustafson/DiauproProject.git
```

open the xCode Project File located "DiauproProject/DiauproPlugin/Builds/MaxOSX/-DiauproPlugin.xcodeproj" directory and build the project in the xCode IDE.

6.1.2 Download and Build the DiauproProject AudioProcessorNode on Ubuntu

Download the JUCE Library:

```
> cd $HOME
> git clone https://github.com/julianstorer/JUCE.git
```

Install JUCE requirements on Linux:

```
sudo apt-get -y install g++
sudo apt-get -y install libfreetype6-dev
sudo apt-get -y install libx11-dev
sudo apt-get -y install libxinerama-dev
sudo apt-get -y install libxrandr-dev
sudo apt-get -y install libxcursor-dev
sudo apt-get -y install mesa-common-dev
sudo apt-get -y install libasound2-dev
sudo apt-get -y install freeglut3-dev
sudo apt-get -y install libxcomposite-dev
```

Build on Linux:

```
> cd $HOME/Documents/
> git clone https://github.com/alexgustafson/DiauproProject.git
> cd DiauproProject/AudioProcessorNode/Build/LinuxMakeFile/
> make CONFIG=Release
```

6.1.3 Install Avahi with Bonjour Compatibility Mode on Ubuntu

Install the Avahi requirements:

```
sudo apt-get install intltool
sudo apt-get install libperl-dev
sudo apt-get install libgtk2.0-dev
sudo apt-get install libgtk3.0-dev
sudo apt-get install libgdbm-dev
sudo apt-get install libdaemon-dev
```

Download Avahi:

```
> curl -O http://www.avahi.org/download/avahi-0.6.31.tar.gz
> tar -xvzf avahi-0.6.31.tar.gz
> cd avahi-0.6.31/
```

Configure and Install Avahi:

```
> ./configure --prefix=/usr  --enable-compat-libdns\_sd  --sysconfdir=/etc  --
localstatedir=/var  --disable-static  --disable-mono  --disable-monodoc  --disable-python
--disable-qt3  --disable-qt4   --disable-gtk  --disable-gtk3  --enable-core-docs  --with-
distro=none --with-systemdsystemunitdir=no
> make
> sudo make install
```

6.2 Evaluated Frameworks

WDL : http://www.cockos.com/wdl/ (+iplug library)
Juce : http://www.juce.com
Open Frameworks : http://openframeworks.cc
Boost : http://www.boost.org
Cinder : http://libcinder.org
LibSourcey : http://sourcey.com/libsourcey/
Qt : http://www.qt.io