Distributed Audio Processing

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

26. Oktober 2015

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.

Inhaltsverzeichnis

Kapitel 1

Introduction

1.1 Ausgangslage

Im digitalen Audio-Bereich übernimmt die CPU immer mehr Aufgaben, die früher durch spezialisierte Hardware gelöst wurden. Dies beinhaltet nicht nur das Aufnehmen und Abmischen von Tonspuren, sondern auch die Klangerzeugung selbst. In den letzten Jahren stiegen die Erwartungen der User bedeutend schneller als die Entwicklung der CPU-Performance. Von aufwändigen physikalischen Emulationen über Pianos und Gitarren bis hin zu virtuellen Analog-Synthesizern, alles wird mittlerweile erwartet, was schnell zur Folge haben kann, dass die CPU überlastet wird.

Ähnlich wie GPU-Karten, die Grafiken und 3D-Anwendungen beschleunigen, können Audio-DSP-Karten digitales Audio in Echtzeit verarbeiten und dabei die Last auf den CPU-Host-Computer verringern. Audio-DSP-Karten werden in der Regel über PCI, FireWire oder Thunderbolt mit dem Rechner verbunden. Die meisten Anbieter von DSP-Karten bieten die Möglichkeit, mehrere Karten parallel zu verbinden, um die Verarbeitungskapazität zu erhöhen.

Im Gegensatz zu GPU-Prozessoren hat sich jedoch kein offener Standard entwickelt, wie OpenGL oder OpenCL. 3D-Grafikanwendungen gewinnen enorm von der Interoperabilität, die OpenGL bietet. Kein solcher Vorteil existiert für digitale Audio-Anwendungen. Auch im Gegensatz zu OpenGL-Anwendungen, kann Audio-Software, die für eine Audio-DSP-Karte entwickelt wird, nicht auf dem CPU-Host oder auf anderen Audio-DSP-Karten ausgeführt werden. Dies führt zu "Vendor Lock-in". Der Verbraucher, der in eine Audio-DSP-Karte und Software investiert, muss beim gleichen Hersteller bleiben. Wenn ein anderer Anbieter von DSP-Hardware ein überlegenes Produkt herstellt, wird ein Verbraucher kaum die Plattform wechseln, wenn bereits eine Investition gemacht wurde.

Vor 10 Jahren war dies ein akzeptabler Kompromiss, weil über PCIe-verbundene DSP-Prozessoren eine deutliche Leistungssteigerung ermöglichten. Heute könnten jedoch, ARM-basierte kostengünstige CPUs, die über Standard-Gigabit-Ethernet

verbunden sind, eine wettbewerbsfähige Alternative bieten.

1.2 Ziel der Arbeit

Ziel dieser Studienarbeit ist es, die Möglichkeiten für die Verteilung von Audio-Verarbeitungsaufgaben für einen Software-basierten Musik-Synthesizer über ein Netzwerk von SBC-Geräten (Single Board Computer) zu untersuchen. Einschränkungen in der Polyphonie oder der Prozessorleistung sollen durch einfaches Hinzufügen eines neuen Geräts im Netzwerk gelöst werden. Ein besonderes Augenmerk soll auf günstige Raspberry-Pi und ähnliche SBC-Geräte gelegt werden, vor allem unter Berücksichtigung von Preis und Leistung. Es existieren viele Standards, damit Audio-Applikationen miteinander kommunizieren können. MIDI und OSC definieren Protokolle zur Steuerung von Audio- Geräten und Software. VST, AU und LW2 sind Standard-Schnittstellen für Plug-ins, die Echtzeitaudioverarbeitung bieten. AVB (IEEE 1722), Jacktrip und Dante sind Standards bzw. Software für die Übertragung von hochwertigem Audio über Ethernet mit minimaler Latenz. Diese Studienarbeit wird einige dieser Standards untersuchen, um zu eruieren, was notwendig ist, um einen skalierbaren und verteilbaren Musik-Synthesizer zu entwickeln, der mit anderer professioneller Audio-Software kompatibel ist. Eine Machbarkeits-Version der verteilbaren Synthesizer-Software wird entwickelt, welche es einem Benutzer erlauben wird, ein Musikstück in Echtzeit zu spielen.

1.3 Aufgabenstellung

- Anforderungsanalyse mit Prioritätsbewertung
- Vergleich von mehreren CPUs und Embedded Systems (Banana Pi, Adapteva, Odroid) hinsichtlich ihrer Nutzbarkeit als Echtzeit- Audioverarbeitungsmodule. Mit dem System, das die Anforderungen am besten erfüllt, wird die Implementierung vorgenommen.
- Entwicklung der Audioverarbeitungssoftware in C ++.
- Entwicklung eines VST-Plug-ins in C++, das als Schnittstelle zwischen gängiger Audio-Software und den Audioverarbeitungsmodulen (pkt 3) dient.
- 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?

Kapitel 2

Einführung in die Thematik

2.1 Hintergrund

Vor 20 Jahren musste man ins Tonstudio, um eine Musik- oder Audioaufnahme zu bearbeiten. Dort befanden sich große Ständer voller Geräte für verschiedene Signalverarbeitungsaufgaben. Zum Beispiel, Kompressoren und Limiters, um den Dynamikbereich zu verarbeiten, oder Hall- und Echogeräte, um einer Tonspur mehr Umgebung zu geben. Am Mischpult konnte man Lautstärke und Frequenzgang mehreren Tonspuren anpassen.

Heute werden alle diese Aufgaben von Software-Plug-ins auf der CPU berechnet.

1996 hatte Steinberg GmbH, die Entwickler von Cubase, einer populären Audio-Produktionssoftware (oder DAW, Digital Audio Workstation), die VST-Schnittstelle und SDK veröffentlicht [10]. Der VST-Plug-in-Standard war besonders, weil er Echtzeit-Verarbeitung von Audiodaten in der CPU ermöglichte. Dazu konnten andere Entwickler jetzt Plug-ins entwickeln, die innerhalb Cubase ausgeführt werden konnten. Der VST-Plug-in-Standard hatte schnell Akzeptanz in der Branche gefunden und wurde sogar von konkurrierenden DAWs implementiert. Obwohl alternative Standards heute existieren, ist VST immer noch der am meisten verbreitete Crossplatform-Standard.

Die Anzahl der Echtzeit-Plug-ins, die auf einer CPU laufen konnte, wurde durch mehrere Faktoren wie Festplattenzugriffsgeschwindigkeiten, Bus-Geschwindigkeiten, viel RAM und OS-Scheduler zum Beispiel beschränkt [3]. Damalige Benutzer erwarteten nicht, mehr als 10 Plug-ins gleichzeitig laufen lassen zu können. Schon die Wiedergabe mehrerer Spuren ohne Plug-ins konnte einen Rechner ins Stottern bringen.

Heute ist es jedoch möglich, hunderte von Tonspuren und Plug-ins in Echtzeit wiederzugeben. Während die Leistungsgrenze angestiegen ist, stiegen auch die Erwartungen der Benutzer. Die Algorithmen heutiger Plug-ins sind sehr viel komplexer als diejenigen von 1996. Es werden akustische Systeme detailgenau modelliert und die Schaltkreise alter 70er-Jahre-Synthesizer digital emuliert.

Auch wenn die CPU-Leistung sich deutlich erhöht hat, sind die Grenzen schnell erreicht.

Es existieren mehrere DSP-basierte Systeme, die, ähnlich wie GPU-Beschleunigungskarten, die CPU-Belastung mindern. Audio-Verarbeitungsaufgaben werden an externe Hardware via PCIe- oder Thunderbolt-Schnittstellen übertragen. Jedoch sind diese DSP-basierten Systeme geschlossen und teuer. Die Entwicklung von Software für einen DSP-Chip ist auch um einiges komplexer als für einen CPU.

2.2 Echtzeit-Audio-Plug-ins

Das Komponieren und Produzieren von Musik erfolgt in der Regel mit Hilfe einer Digital Audio Workstation(DAW-)-Software. MIDI-Events und Audio-Aufnahmen werden als Spuren arrangiert, gemischt, redigiert und verarbeitet. Um Änderungen rückgängig zu machen, werden Bearbeitungen in einem zerstörungsfreien Verfahren gemacht, das dynamisch in Echtzeit während der Wiedergabe berechnet wird. Die ursprünglichen Audiodaten bleiben immer im ursprünglichen Zustand erhalten. Der Benutzer kann die Parameter eines Effektes oder Prozesses in Echtzeit ändern und mit verschiedenen Einstellungen experimentieren, ohne zu fürchten, dass die ursprünglichen Audioaufzeichnungen geändert werden.

Eine DAW-Anwendung hat in der Regel mehrere Echtzeit-Effekte eingebaut, welche ein Benutzer auf Audiospuren einsetzen kann. Zusätzlich zu den eingebauten Effekten sind alle professionellen DAW-Anwendungen auch in der Lage, Dritthersteller-Plug-ins zu laden. In Abhängigkeit von der Plattform und dem Anbieter werden ein oder mehrere Plug-In-Standards implementiert. Steinbergs VST-Standard ist der meistverbreitete.

Alle Standards funktionieren in einer ähnlichen Weise. Die Host-DAW-Anwendung übergibt in regelmäßigen Abständen das Plug-in über eine Rückruffunktion an zu verarbeitende Audiodaten. Das Plug-in muss innerhalb eines definierten Zeitrahmens die Verarbeitung abgeschlossen haben und wartet dann auf den nächsten Abruf.

Audio-Plug-ins können auch eine grafische Oberfläche zur Verfügung stellen, über die ein Benutzerparameter geändert und gespeichert werden kann. Dies könnte z. B. die Grenzfrequenz eines Tiefpassfilters oder die Verzögerungszeit eines Hall-Effekts sein.

Aus der Sicht des Programmierers sind Plug-ins dynamisch ladbare Bibliotheken, die eine spezifizierte API implementieren. Die Host-DAW-Anwendung kann dann während der Laufzeit laden und Audiodaten durch sie verarbeiten lassen [6]. Auf der Windows-Plattform werden VST-Plug-ins als Dynamic Link Libraries (DLL) kompiliert, auf Mac OSX sind sie Mach-O-Bundles. Die ursprünglichen Apple-AudioUnit-Plug-ins sind auch als Mach-O-Bundles zusammengestellt, sie haben fast identische Funktionalität. Andere alternative Plug-in-Formate sind Avids RTAS und AAX-Plug-Formate, Microsofts DirectX-Architektur oder LADSPA, DSSI und GW2 auf Linux.

Echtzeit-Audio-Plug-ins, wie der Name schon sagt, müssen in der Lage sein,

ihre Aufgaben schnell genug zu erfüllen, um den Echtzeit-Audio-Anforderungen gerecht zu werden. Wie schnell ist schnell genug? Nun, das hängt davon ab, wie man "Echtzeit" definiert. Für Audio-Anwendungen wird Echtzeit in Bezug auf die Latenz des Audiosystems definiert. Die Gesamtverzögerung zwischen dem Eintreffen eines Audiosignals in das System (beim Analog-Digital-Wandler zum Beispiel), Verarbeitung, und dem Verlassen des Systems (beim Digital-Analog-Wandler oder Audio-Ausgang) ist die Latenz. Die maximal zulässige Latenzzeit für Audioanwendung ist ungefähr 10 ms [7]. Ist die Latenzzeit länger als 10ms, wird sie als störend empfunden und ist nicht mehr akzeptabel für Live-Performance-Anwendungen.

Jeder Verarbeitungsprozess wird eine gewisse Verzögerung hinzufügen. Doch innerhalb der Audio-Verarbeitungsfunktion muss der Programmierer darauf achten, keine unnötigen oder nicht berechenbaren Verzögerungen zu erzeugen. Es ist auch wichtig, zu verstehen, dass die Audio-Verarbeitungsfunktion im Kontext eines hoch-prioritären System-Threads berechnet wird. Nichts innerhalb dieser Funktion sollte auf Ressourcen warten, welche von anderen Threads mit niedrigerer Priorität berechnet werden. Beispiele von zu vermeidenden Dingen sind Speicherallokationen und Speicherdeallokationen oder das Warten auf bzw. Sperren eines Mutex [6], oder direkt Aktualisierungen der grafischen Oberfläche vorzunehmen.

2.3 Audio via Ethernet

Das Versenden von Audiodaten über Netzwerke ist nicht neu. Auch nicht das Senden von Audiodaten in Echtzeit. Das IETF (Internet Engineering Taskforce) RFC 3550 definiert das Real-time Transport Protocol für die Bereitstellung von Audio und Video in Echtzeit über IP-Netzwerke. RTP wird als Grundlage für die meisten Medien-Streaming und Video-Konferenz-Anwendungen eingesetzt.

Andere neue Spezifikationen wie AVB[1] und AES67[2] bauen auf RTP und ermöglichen präzises Timing und Synchronisieren für professionelle Audio-Anwendungen. Die Synchronisation ist in diesen Standards wichtig, weil sie sich mit der Steuerung von auf mehrere Hosts verteilter Audiohardware befassen.

Hardware-Synchronisation ist nicht für dieses Projekt relevant, weil es sich nicht mit externer Audio-Hardware befasst. Ziel dieses Projektes ist es, externe CPUs als Audio-Koprozessoren über Gigabit-Ethernet zu nutzen. Dennoch bieten die AVB- und AES67-Standards viele Erkenntnisse darüber, wie die Datenübertragung für Low-latency-Anwendungen optimiert werden kann und beweisen deren Machbarkeit. AES67 definiert Richtlinien, um Latenzzeiten deutlich unter 1 ms zu erreichen, auch bei Hunderten von parallel laufenden

[1] Audio Video Bridging ist einer Sammplug IEEE standards welche Zeit-Synchroniziertes niedrig Latenz Streaming Dienste ermöglicht

[2] AES67 , ein von der Audio Engineering Society definierte Standard, die bestehenden Low-Latency Streaming-Systeme Interoperabilität ermöglicht. AES67 definieret keine neuen Technologien, sondern versucht, einen kleinsten gemeinsamen Nenner zu setzten, durch die bestehenden Standards kompatibel sein können.

Audiostreams. Dies ist viel schneller als die in vielen DSP-basierten Systemen verwendeten Legacy-PCI- und Firewire-Verbindungen [1].

2.4 Single-Board-Computer

Die Popularität der Raspberry Pi hat eine ganze Industrie rund um Single-Board-Computer (SBC) generiert. Basierend auf Hardware, welche in Mobiltelefonen zu finden ist, sind diese kleinen Low-power-Geräte sehr beliebt, weil sie preiswert und einfach zu bedienen sind. Der größte Vorteil von SBCs im Vergleich zu anderen eingebetteten Geräten ist, dass sie mit Android- und Linux-Betriebssystemen laufen und somit mit den gleichen, auf Desktop-Computern verfügbaren, Tools programmiert werden können.

Aktuelle High-end-SBCs werden sogar mit Gigabit-Ethernet ausgestattet und Quad-Core-CPUs mit Geschwindigkeiten von weit über 1 GHz getaktet. Vergleicht man diese Systeme mit den 450 MHz G3 PPC-Systemen, auf dem die ersten VST- Software-Plug-ins berechnet wurden, sollten wir erwarten können, dass die neueren High-end-SBCs exzellente Audio-Koprozessoren sind.

Zwei SBCs sind es wert, in Betracht gezogen zu werden, da sie möglicherweise eine noch bessere Performance als Audio-Koprozessoren bieten. Das Parallella-Board hat einen 16 Core Epiphany-Koprozessor welcher verwendet werden kann, um die Audioverarbeitung parallel auszuführen. Standard-Frameworks wie OpenCL, MPI oder OpenMP können verwendet werden, um die Epiphany-Cores zu benützen. Der ODROID-XU4 SBC enthält einen Mali-T628-GPU-Koprozessor, der auch OpenCL- kompatibel ist. Beide sind für weniger als $ 100 erhältlich.

Die Programmierung von Audioverarbeitungsalgorithmen als OpenCL Kernels könnte einiges komplexer sein als in C++, aber OpenCL bietet herstellerunabhängigen Zugriff auf GPGPU-Computing und hat den zusätzlichen Vorteil, dass es auch auf einem CPU ohne GPU-Beschleunigung verwendet werden kann [5].

Die Untersuchung dieser und anderer OpenCL-fähiger SBCs könnte ein interessantes Folgeprojekt sein.

2.5 Virtuelle analoge Synthese

Der Begriff "virtuelle analoge Synthese"wird verwendet, um die Echtzeit-Emulation des analogen Synthesizers der 60er- und 70er-Jahre zu beschreiben. Die Komplexität und die Ziele einer Emulation können variieren. Einige Emulationen gehen so weit, dass sie die tatsächlichen elektronischen Komponenten der Vintage-Synthese-Schaltungen simulieren, andere modellieren nur grob den Signalfluss.

Unabhängig von der Art der Emulation hat die virtuell analoge Synthese zwei Probleme, mit denen es sich zu beschäftigen gilt: Latenzzeit und Aliasing. Das Problem der Latenzzeit wurde bereits oben beschrieben. Jede Verarbeitung erzeugt eine Verzögerung des Signals hervor; die Komplexität der Verarbeitung kann die Verzögerung erhöhen oder mehr CPU-Zyklen verbrauchen. Aliasing

ist eine hörbare Verzerrung des Signals, welche durch Frequenzen verursacht werden, die höher sind als die Abtastrate des Systems erlaubt.

Analog-Synthesizer verwenden üblicherweise subtraktive Synthese. Ein oder mehrere Klangerzeuger (Oszillatoren) erzeugen Signale mit besonderen harmonischen Qualitäten. Diese Signale werden durch Filter geschickt, welche die Frequenzen aus dem Signal ßubtrahieren". Die Oszillatoren, Filter und Amplitude können moduliert werden.Abbildung 2.1 ist ein einfaches Blockschaltbild einer typischen subtraktiven Synthesizer-Stimme.

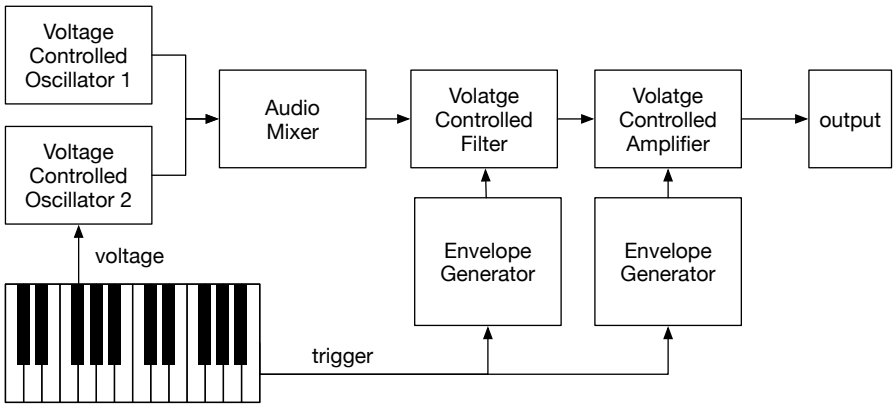

Abbildung 2.1: Blockschaltbild einer subtraktiven Synthesizer-Stimme

Die spannungsgesteuerte Oszillatoren (voltage controlled oscillators, VCO) erzeugen einfache Wellenformen in der Tonhöhe, die auf der Tastatur gespielt wird. Der Benutzer kann in der Regel zwischen einer Kombination aus Sägezahn-, Rechteck- oder Dreieck-Wellenform wählen. Die Frequenzen oder die Klangfarbe der Wellenformen können dann durch die folgenden Filter und Amplituden-Blöcke moduliert werden.

Der erste Eindruck könnte sein, dass die Modellierung des Oszillators einfach ist. Eine digital erzeugte Rechteck- oder Sägezahn-Wellenform sollte leicht zu implementieren sein. Die 5-kHz-Sägezahn-Wellenform beispielsweise würde sich alle 8,82 Samples wiederholen bei einer Taktrate von 44,1 kHz. So würde die Wellenform sich linear von -1,0 auf 1,0 erhöhen, dann zurückspringen auf -1.0 und wieder von vorn beginnen. Was bedeutet aber 0,82 Samples in einem diskreten digitalen System? Abbildung 2.2 veranschaulicht das Problem. Die linke Spalte zeigt einen Abschnitt einer idealisierten 5kHz-Sägezahnwellenform und den entsprechenden Frequenzinhalt. Oberhalb der Grundfrequenz 5 kHz sind Oberwellen, die weit über 100 kHz hörbar sein werden. Die rechte Spalte zeigt den gleichen Abschnitt einer 5kHz-Sägezahnwellenform in einer mit 44,1kHz getakteten diskreten Umgebung. Die Wellenform selbst ist verzerrt und die höheren Harmonien sichtbar von der 22,05 kHz Nyquist-Grenze nach unten reflektiert.

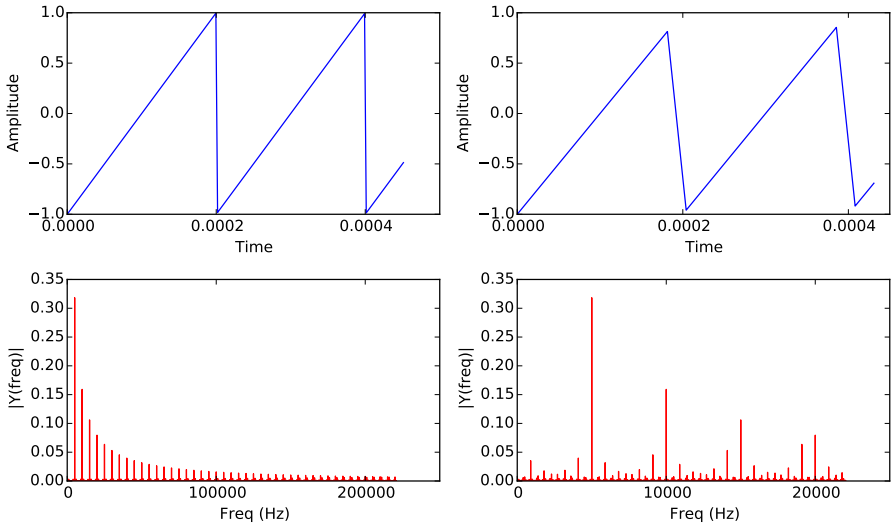

Abbildung 2.2: Ideale und Aliased 5kHz-Sägezahnwellenform

Es gibt verschiedene Verfahren zur Beseitigung oder Verminderung von Aliasing. Die effektivste ist es, die Oberwellen mit einer Reihe von Sinus-Wellen bis zu der Nyquist-Grenze oder die Hälfte der Abtastrate des Systems zu konstruieren, aber dies ist sehr rechenintensiv. Weniger CPU-intensive Strategien sind Oversampling, Bandbegrenzung oder andere Aliasing-Unterdrückungsmethoden [9].

Kapitel 3

Anforderungsanalyse

3.1 Allgemeine Anforderungen

Die Anwendung besteht aus zwei Komponenten, einem Audio Plug-in, das auf der Haupt-CPU-Maschine läuft, und den Rechenknoten, die auf vernetzten SBC-Geräten laufen. Das Audio-Plug-in leitet MIDI-Steuerung und Audio-Daten zu den Rechenknoten. Die Rechenknoten senden die verarbeiteten Audiodaten zurück an das Audio-Plug-in, das wiederum die Daten an die Host DAW-Anwendung übergibt. Die Gesamtumlaufzeit, einschließlich der Verarbeitung, sollte 10 ms nicht überschreiten. Dies ist die maximal erlaubte Latenzzeit für Live-Anwendungen [7].

Die Rechenknoten und das Audio Plug-in müssen in sich abgeschlossen sein und funktionieren, ohne dass der Benutzer externe Bibliotheken, Frameworks oder Server installieren muss[1].

3.2 Verteilte Datenverarbeitung

Um die Belastung der Host-CPU zu reduzieren, sollen Audio-Verarbeitungsaufgaben an externe SBCs über Gigabit-Ethernet verteilt werden. Auf der Host-CPU arbeitet das Audio Plug-in als Master-Knoten. Für den Host-DAW sollte die Verteilung der Audio-Verarbeitungsaufgaben vollkommen unsichtbar sein. Der Master-Knoten empfängt MIDI- und Audiodaten vom Host-DAW und gibt die Ergebnisse zurück wie jedes andere Audio-Plug-in.

Im Gegensatz zu anderen dezentralen Verarbeitungssystemen, wo große Datenmengen an Rechenknoten in Paketen verteilt werden, bekommt das Plug-in in sehr kurzen regelmäßigen Abständen Zugriff auf einen kleinen Pufferspeicher an Audiodaten. Das Plug-in muss in kurzer Zeit die Daten verarbeiten und zurückgeben an die Host DAW-Anwendung, bevor der nächste Puffer bereit-

[1] Die einzige Ausnahme könnte ZeroConf / Bonjour unter Linux oder Windows sein. Siehe Anhang

steht. Dies schränkt die Möglichkeiten der Verteilung von Bearbeitungsaufträgen ein.

Es gibt verschiedene Arten, Verarbeitungsaufgaben zu parallelisieren. Jede Plug-in-Instanz könnte den gesamten Auftrag an einen Rechenknoten delegieren wie in Abbildung 3.1 dargestellt. Besteht eine Verarbeitungsaufgabe aus mehreren, in Serie geschalteten, Verarbeitungsblöcken, dann könnte jeder Block einem Rechenknoten zugeteilt werden, wie in Abbildung3.2 dargestellt. Speziell bei einem virtuellen Synthesizer-Plug-in könnte jede gespielte Stimme seinem eigenen Rechenknoten zugeteilt werden, siehe Abbildung 3.2.

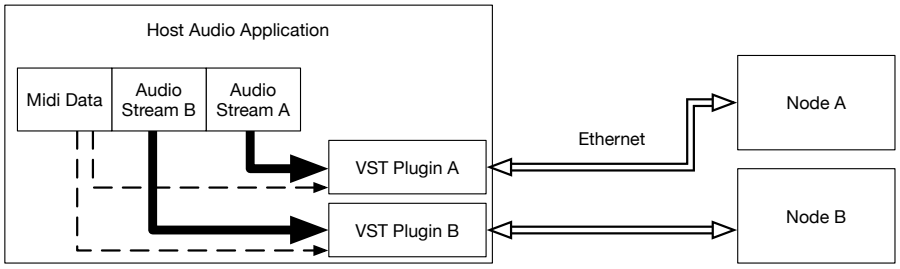

Abbildung 3.1: Jedes Plug-in an einen Knoten verteilt

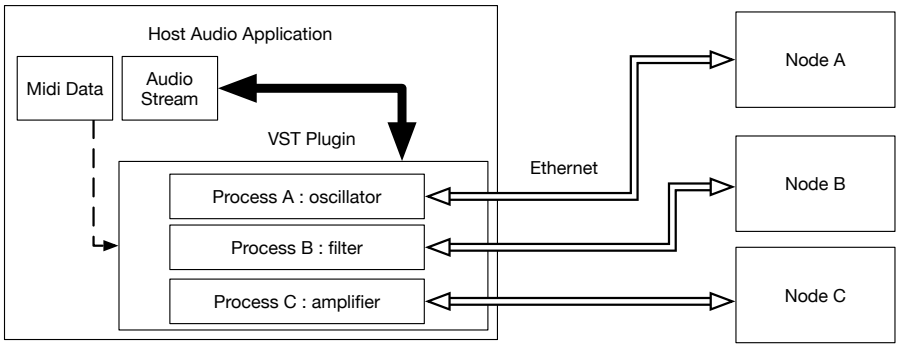

Abbildung 3.2: Jeder Verarbeitungsblock an einen Knoten verteilt

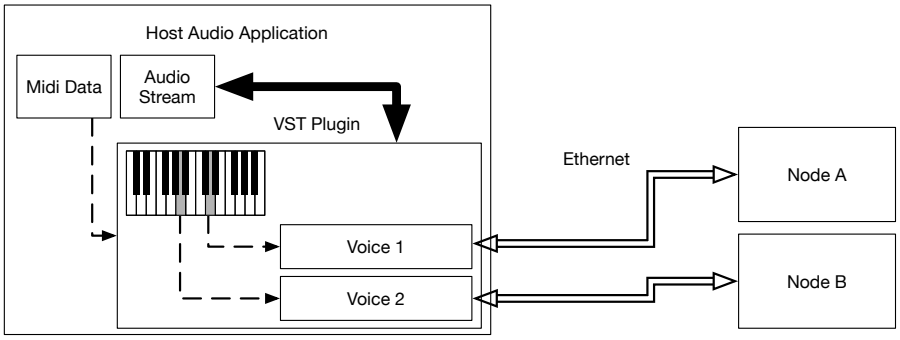

Abbildung 3.3: Jede Synth-Stimme an einen Knoten verteilt

Für dieses Projekt wird die zweite Option implementiert aus Gründen der Testbarkeit, auch wenn dies vielleicht nicht die effizienteste Implementierung ist.

3.3 Audio-Plug-in

Aus Sicht der Host-DAW-Anwendung muss das Plug-in wie ein ganz normales VST-Plug-in aussehen. Das Verteilen der Aufgaben auf vernetzte Rechenknoten muss transparent funktionieren. Das Plug-in wird nicht in einer isolierten Umgebung ausgeführt. Es teilt die Rechenressourcen mit dem Host-DAW so wie eine beliebige Anzahl anderer Plug-ins. Daher sollte die Belastung auf der CPU sorgfältig auf das Nötigste reduziert werden.

Audio-Plug-ins haben typischerweise einen Zustand, der Parametereinstellungen enthält, die vom Audio-Host oder vom Benutzer gesteuert werden können. Der aktuelle Zustand muss den vernetzten SBC-Modulen auch mitgeteilt werden. Dies ist notwendig, damit ein Audio Plug-in den Zustand eines fortlaufenden Prozesses nahtlos an einen vernetzten Rechenknoten weitergeben kann. Es könnte auch verwendet werden, um einen vernetzten Rechenknoten durch Umschalten des aktiven Zustandes die Verarbeitung für mehrere Audio-Plug-ins übernehmen zu lassen.

Das Audio-Plug-in stellt die folgenden Anforderungen:

- lauffähig als Echtzeit-VST-Audio-Plug-in in einer Standard DAW-Anwendung
- Lokalisieren und verbinden mit einem oder mehr Rechenknoten im Netzwerk
- Übermitteln von MIDI- und Audio-Daten von der Host-DAW-Anwendung an die vernetzten Rechenknoten

- Audiodaten von den Rechenknoten empfangen und diese zurück an die DAW-Anwendung geben
- ist kein entsprechender Knoten im Netzwerk vorhanden, muss die Audioverarbeitung lokal stattfinden
- das Audio-Plug-in muss seinen aktuellen Zustand an den Rechenknoten übermitteln können

3.4 Externer Rechenknoten

Die Rechenknoten sind Anwendungen, die auf den vernetzten SBC-Geräten laufen. Je nach Art der implementierten Parallellisierung im Audio-Plug-in bekommt der Rechenknoten die gesamte Verarbeitungsaufgabe oder einfach nur einen Teil davon. Für dieses Projekt werden die Rechenknoten als Reihe von Prozessoren in eine übergeordnete Anwendung umgesetzt. Die übergeordnete Anwendung implementiert einen Socket-Listener, welche den zugeordneten Rechenknoten aufruft sobald eine Audio-Verarbeitungsaufgabe eintrifft. Die Verfügbarkeit, Art und Lage der Rechenknoten und das entsprechende Socket wird über das Netzwerk via Bonjour / ZeroConf übertragen.

Die Rechenknoten sollten zustandslos sein. Jeder Zyklus des Audio-Verarbeitungsalgorithmus sollte nur die Zustandsdaten des entsprechenden Pakets berücksichtigen müssen. Werden die Zustandsdaten durch den Verarbeitungsalgorithmus geändert, müssen sie an das Audio-Plug-in zurückgesendet werden. Damit soll sichergestellt werden, dass ein Rechenknoten zu jeder Zeit in eine laufende Session einspringen kann, ohne etwas über vorherige Events wissen zu müssen. Es hat auch den zusätzlichen Vorteil, dass ein Verarbeitungsknoten in der Lage wäre, Aufträge für mehrere Instanzen eines bestimmten Plug-ins verarbeiten zu können.

Die Rechenknoten müssen die folgenden Anforderungen erfüllen:

- ihre Verfügbarkeit und ihren Standort im Netzwerk via Bonjour / ZeroConf bekanntgeben
- Steuerdaten vom Audio-Plug-in akzeptieren
- eingehende Audiodaten und MIDI-Daten vom Audio-Plug-in verarbeiten
- die verarbeiteten Daten sofort an das Audio-Plug-in zurücksenden

3.5 Software-Anforderungen

In Echtzeit-Audio-Anwendungen ist das Einhalten zeitlicher Vorgaben kritisch. Dies mag selbstverständlich klingen, aber für einen Programmierer bedeutet es den Verzicht auf viele Annehmlichkeiten der modernen Programmierung, besonders das Arbeiten mit High-Level-Programmier-Sprachen wie Java oder Python. Echtzeit-Audio ist eine "Hard Realtime"-Aufgabe und dafür eignen sich nur

Low-Level-Sprachen wie C oder C++. Auch die meisten Audio-Anwendungsschnittstellen und Bibliotheken wie die VST SDK sind für C/C ++.

Professionelle Audio-Anwendungen laufen in der Regel auf Mac OSX oder Windows-Betriebssystemen, daher muss das Audio-Plug-in auf diesen Systemen kompatibel sein. Die Rechenknoten, welche auf den SBC-Geräte laufen, müssen Linux-kompatibel sein. Doch beide Anwendungen sollten einen Großteil ihres Quellcodes teilen können, da ihre Aufgaben sich größtenteils überschneiden.

Es gibt viele C ++ Bibliotheken und Frameworks, welche Cross-Plattform-Kompatibilität ermöglichen und nebenbei auch den Zugriff des Programmierers auf High-Level-Konstrukte, wie automatische Speicherbereinigung mittels intelligentem Zeiger und Referenzzählung, die das Programmieren in C++ einfacher machen.

3.5.1 Evaluierte C++ Frameworks

Boost ist das beliebteste plattformübergreifende C ++ Framework. Viele seiner Funktionalitäten wurden sogar der C ++ 11-Standard-Bibliothek hinzugefügt. Andere Frameworks wie Cinder und Open Frameworks bieten viele High-Level-Funktionen, um schnell interaktive medienreiche Anwendungen zu programmieren. Zwei Bibliotheken die hervorzuheben sind, JUCE und WDL, bieten besonders für Audio-Plug-ins und DAW-Anwendungen zugeschnittene Funktionen und Klassen. Von diesen beiden hat JUCE eine viel größere Benutzergemeinschaft (einschließlich Anbieter von DSP-basierten Audio-Koprozessoren).

Software-Framework-Kriterien:

- Plattformübergreifend für OSX, Linux und Windows
- Bietet High-Level-Konstrukte wie intelligente Zeiger
- Unterstützung für plattformübergreifende Audiointegration
- Sollte gut dokumentiert sein und eine aktive Benutzergemeinschaft haben
- Bietet plattformübergreifenden Netzwerkzugriff

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

Auf Grundlage des Kriterienvergleiches und früherer Erfahrungen mit anderen Projekten wurde für die Implementierung dieses Projektes JUCE ausgewählt.

[2] einfache Socket-Verwaltungs-Klasses
[3] durch optionale ïplug"Libraries ermöglicht
[4] ofxNetwork-addon ermöglicht einfaches verwalten von TCP und UDP Sockets

Kapitel 4

Implementation

4.1 Architecture

Abbildung 4.1 zeigt einen Überblick über die Audioverarbeitungsumgebung. Der Benutzer des Systems arbeitet mit der DAW-Anwendung welches auf dem Host-CPU läuft, und kann einen Audio Plug-in auf eine bestimmte Audiospur aktivieren. Ein einzelner Audio Plug-in kann einer oder mehrere Audio Prozessoren enthalten. Prozessoren, die fähig sind ihre Aufgaben zu Verteilen suchen nach einem entsprechenden Rechenknoten auf einem Vernetztes SBC-Gerät.

Die Geräte werden über Gigabit-Ethernet vernetzt. Es wird angenommen, dass das Netzwerk nicht für sonstiges signifikanten Verkehr benützt werden.

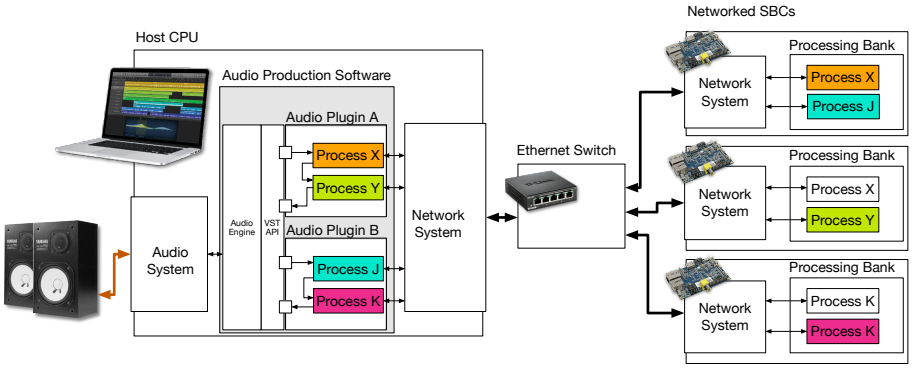

Abbildung 4.1: Architectural Überblick

4.1.1 Host CPU

Abbildung 4.2 zeigt die Komponenten welche für Echtzeit-Audio auf der CPU benötigt werden. Audio-Hardware benötigt, einen kontinuierliches genau-getaktetes

Datenstrom um es den Digital-Analog-Wandlern bereitzustellen. Dies geschieht durch periodisches Abfragen des Betriebssystems über die Hardware-Interrupts. Die angeforderte Puffergrße kann wenige Abtastwerte beinhalten und die Abfrageintervall weniger als 1 ms. Dies ist abhängig von der benützten Hardware und der Audio-Treiber.

Das Betriebssystem bietet eine Abstraktionsschicht in Form einer API für die Anwendungssoftware. Dies gibt der Anwendungs-Software einer einheitlichen Schnittstelle, unabhängig von der Marke der Audio-Hardware und Treiber.

Das VST-API ist eine weitere Abstraktionsschicht, die weiderum eine einheitliche Schnittstelle für Plugin-Anbieter anbietet, unabhängig von der OS. Aber VST ist nicht die einzige Plugin API. Die JUCE Bibliothek bietet einen eigenen Plugin-API, die einfacher ist und abstrahiert die Unterschiede zwischen verschiedene Plugin-APIs.

Die Datenanabfrage, ausgelöst duch Interrupts von der Audio-Hardware, werden durch das Betriebssystems an die DAW-Anwendung durch Rückruffunktionen weitergeleitet. Das DAW hat zuvor, beim aufstarten, die entsprechende Rückrufffunktionen an das Betriebssystems registriert. Die DAW-Anwendung wiederum lietet die Datenabfrage, ebenfalls duch die vom VST Spzifizierten Rückruffunktionen an alle aktiven Plug-ins weiter.

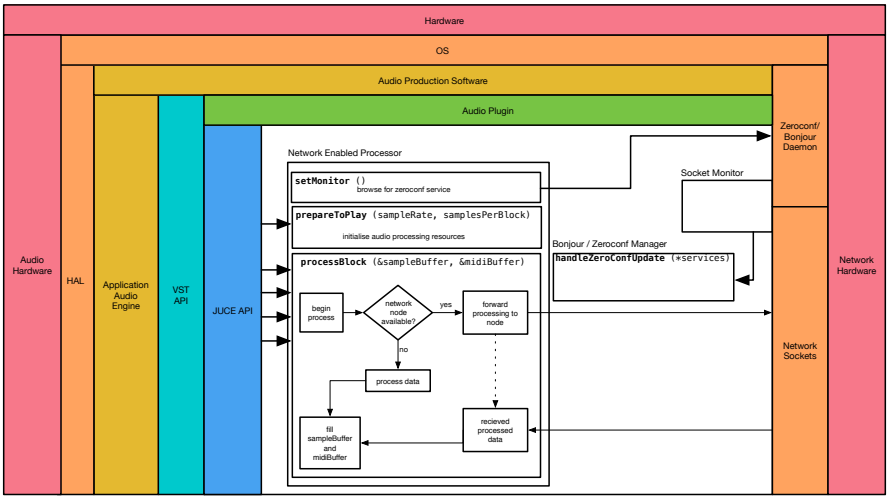

Abbildung 4.2: Host CPU Überblick

Die für dieses Projekt umgesetztes Audio Plug-in hat mehrere Netzwerkprozessoren aktiviert, in Abbildung 4.2 wird nur eine dargestellt als Beispiel. Wenn das Plug-in von der Host-Software instanziiert wird, instanziiert es weiderum jeder seiner internen Prozessoren. Die Prozessoren rufen je den Bonjour / Zeroconf-Daemon des Betriebssystems auf, um eine passende im Netzwerk regis-

triertes Rechenknoten zu finden. Der Aufruf übergibt das Daemon eine Socket über eine benachrichtigung zurück schicken kann wenn es einen passendes Service im Netzwerk gefunden hat.

Die Bonjour / Zeroconf-Daemon übergibt dem Audio Plug-in mittels der Socket einer Liste der gefundenen Services. Die Audio Plug-in durchsucht die Liste nach einer freien Rechenkonten. Das ausgewählter Rechenknoten wird in das ächtiveNode"Variable gespeichert.

Wenn eine Anfrage für Audio-Daten von der DAW-Anwendung an das Plugin übergeben wird, dann wird der "processBlock"Funktion des Plugins aufgerufen. Das Plugin ruft wiederum die "processBlock"Funktionen jedes seiner internen Audio Prozessoren nacheinander. In Abbildung 4.2 wird dies vereinfacht dargestellt, mit der "processBlock"Funktion von einem einzigen Audio Prozessor. Der "processBlock"Funktion wird einen Referenz auf die aktuelle zu verarbeiten Audiopuffer und MIDI-Puffer übergegeben. Der Audiopuffer enthält die einzelnen Audioabtastungen für jeden Kanal als Float-Werte. Die MIDI-Puffer enthält Performance-Daten wie der Eintrittszeit und Tonhöhe der gespielten Noten.

Innerhalb der "processBlock"Funktion prüft der Prozessor, ob ein Rechenknoten im ächtiveNode"gespeichert ist. Wenn ja, dann leitet er sofort die MIDI- und Audiodaten sowie die eigene Zustandsdaten an die Rechenknoten und wartet auf die Antwort. Wenn die Antwort eintrifft werden die Daten zurück in die entsprechende Puffer kopiert. Die DAW-Anwendung geht dann weiter die nächsten Plug-ins abfragen, bis die gesamte Verarbeitungskette abgeschlossen ist. Die resultierenden Puffer wird an die Audiohardware über die HAL API geschickt.

4.1.2 Vernetzte SBCs

Abbildung 4.3 zeigt die Komponenten der vernetzten SBC-Geräte. Eine Verarbeitungsanwendung kann eine beliebige Anzahl von Prozessoren erhalten, die jeweils ihre Dienste über den Zeroconf/Bonjour-Daemon im Netzwerk anbieten. Das Zeroconf/Bonjou-Daemon wird als teil des Betriebsystems hier dargestellt. Die Zeroconf/Bonjour-Daemon sendet die Verfügbarkeit, der Verarbeitung-Art und die Portnummern von jedem der verfügbaren Prozessoren.

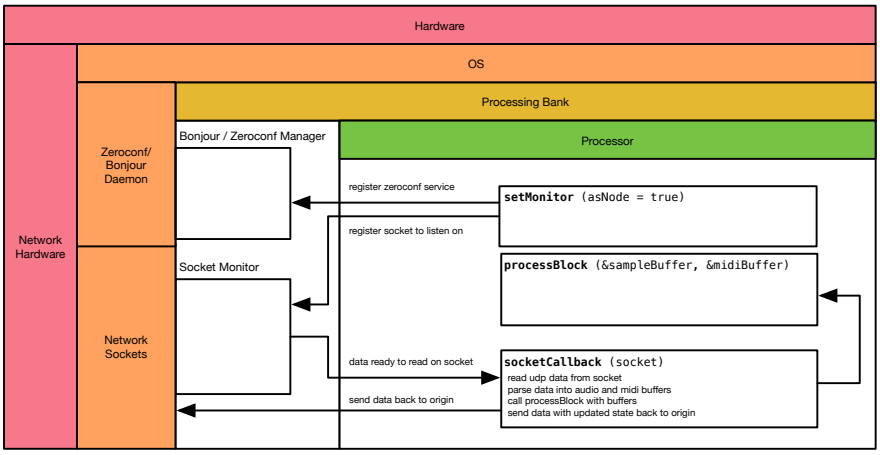

Abbildung 4.3: SBC Processor Überblick

Der Prozessor übergibt eine offene Socket an einen Socket-Monitor und registriert sich als dazugehörende Socket-Zuhörer-Objekt. Die Socket-Monitor besitzt eine Reihe von Sockets und einen Verweis auf jeden Zuhörer. Es führt eine ßelect"Funktion auf der Reihe von Socket und wartet. Wenn Daten an einem der Sockets eintreffen, erwacht der ßelect"Funktion und der Socket Monitor benachrichtigt den entsprechenden Socket-Zuhörer-Objekte über eine Rückruffunktion, das Daten zu verfügung stehen.

Der entsprechende Prozessor wird über die Rückruffunktion benachrichtigt, liest die Daten, entpackt die Audio-, Zustands- und MIDI-Daten, und führt den processBlock Funktion aus. Die Ergebnisse und aktualisierte Zustandsinformation werden am Sender zurück geschickt.

4.2 Software Komponenten

Die Anwendung ist in Komponenten (Klassen) aufgebaut welche jeweils die Verantwortung für einen Teilbereich entkapselt. Jedes Modul wurde in einem "pseudo Test Driven"Methodik entwickelt. Die Tests wurden nicht zwingenderweise entwickelt, bevor der entsprechende Code geschrieben wurde, aber auf jeden Fall kurz danach. Dies führte zu stabilen Code, dass in jeder Iteration geprüft werden konnte, isoliert von anderen Komponenten.

Die Interaktion mit einer Klasse wird in einer dazugehörenden SZuhöhrer"Klasse definiert. In der SZuhöhrer"Klasse sind Rückruffunkionen spezifiziert, die einer Client-Klasse zu implementieren hat. Ein Client-Klasse kann von der Listener-Klasse erben und die Rückruffunktionen überladen.

4.2.1 Socket Monitor

Die Monitor-Klasse wurde entwickelt, um Sockets effizient zu verwalten. Client-Klassen erben von der FileDescriptorListener Klasse, um benachrichtet zu werden, wenn Daten verfügbar sind auf einer dazugehörendes Socket.

Die Monitor-Klasse hat einen Thread, welches eine select-Funktion aufruft. Die select-funktion wird einen Array von Sockets als Filedescriptors übergeben, dann blockiert es. Sobald Daten auf einer der Sockets eintreffen, fird die select-Funktion wieter fahren. Der Thread geht prüfft in einer Schleife alle Filedescriptors bis er das richtige gefunden hat und benachrichtigt das entsprechende FileDescriptorListener Objekt.

Eine der Sockets welches an die select-Funktion übergeben wird der Monitor-Klasse eigenen control_listener Socket. Die Monitor-Klasse kann ein Signal an dieser Socket schicken, um den blockierten Thread aus dem select-Funktion zu wecken und den Zustand aktualisieren. Die Monitor-Klasse tut dies, wenn eine neue Client sich registriert bzw. sich entfernt, und wenn der Monitor-Klasse heruntergefahren werden muss.

4.2.2 ZeroconfManager

Zeroconf (kurz für Zero Configuration Netzwerking, auch als Bonjour auf OSX bekannt) ist einer Technologie, die es einen Softwaredienstleistung erlaubt ihre Verfügbarkeit und Standort über einen Netzwerk bekannt zu machen. Drucker und Multimedia-Geräte verwenden Zeroconf z.B., damit sie für andere Netzwerkteilnehmer leicht zu finden und nutzen sind.

Bonjour auf OSX und den compatibile Funktionen von Avahi unter Linux definieren eine Rückruff-basiertes API zu der Bonjour oder Avahi Daemon-Prozess. Die API ist in C. Die ZeroConfManager-Klasse kapselt die API in einer objektorientierten Weise. Andere Klassen, welche die Zeroconfigdienste ansprechen müssen können von der ZeroConfListenere-Klasse erben, und können sich bei der ZeroConfManager-Klasse als Zuhörer registrieren.

Wird ein neuer Software-Dienst im Netzwerk registriert, oder verschwindet ein Bestehender aus dem Netzwerk, werden die entsprechenden Zuhörer einer Liste aller aktiven Services mitgeteilt. Wenn sie noch mit einem Software-Dienst, der nicht mehr verfügbar ist, verbunden sind, muss es sich von diesem Dienst trennen.

Um einen Dienst an eine bestimmte IP-Adresse und Port-Nummer aufzulösen, müssen mehrere asynchrone Aufrufe an der Bonjour-Daemon vorgenommen werden. Zwischen den Aufrufen müssen die Ergebnisse gespeichert werden und den Zustand aktuallisiert. Die ZeroConfManager verbirgt diese Komplexität von den Zuhörern und benachrichtigt sie nur, wenn alle Informationen vorhanden sind.

4.2.3 DiauproMessage

Die DiauproMessage Klasse verwaltet die Serialisierung und Deserialisierung von Daten zu, beziehungsweise von, Datagramm-Pakete. Die Datagramme selbst bestehen aus einer Header mit festen Länge, und Nutzdaten mit variable Länge, welche die Audio-, MIDI- und Zustanddaten enthält.

Der Header wird wie folgt definiert:

```
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;
};
```

Die DiaproMessage Klasse bemüht sich, bestehende zugewiesenen Speicher zu verwenden, wenn möglich. Instanzen DiauproMessage sollte nicht alloziert werden in den Thread, der die Audio-Verarbeitungsroutinen aufruft. Sie sollten lieber voralloziert werden mit genügend Speicher, um einen UPD Datagram mit maximale Größe (64 KB) zu speichern. Dieser Instanz soll dann für jeden Aufruf-Zyklus wiederverwendet werden damit Speicher nicht neu alloziert werden muss während der Zeit kritische Prozess.

4.2.4 DiauproProcessor

Die DiauproProcessor-Klasse ist die Basisklasse, von der anderen Prozessoren erben sollen. Die DiauproProcessor-Klasse erbt wiederum von der JUCE Audioprocessor-Klasse, die alle Funktionen der verschiedenen Audio-Plugin-Formate kapselt. Klassen, die von Audioprocessor erben können direkt in ein VST-Plugin gewickelt (wrapped) werden.

Eine Klasse, die von DiauproProcessor erbt muss die Funktionen ”localProcessünd ”getServiceTag”überladen. Die ”localProcess”Funktion führt die Audio-Verarbeitung aus. ”getServiceTag”gibt einer Zeichenfolge zurück, die verwendet wird, um seine Dienste im Netzwerk bekannt zu machen oder danach zu suchen.

Klassen, die von DiauproProcessor erben können instanziert und eingerichtet werden so, dass sie entweder in der Audio Plug-in oder in der Rechenknoten laufen. Im Plug-in ausführt wird es ein entsprechendes Rechenknoten im Netzwerk suchen. Falls nichts passendes gefunden wird, wird das ”local-Process”Funktion lokal verwendet. Es sie finden eine vernetzte Service für alle ankommenden Audio- und MIDI-Daten werden auf diesen Dienst weitergeleitet.

Wenn in einem Rechenknoten ausgeführt, werden Klassen, die von DiauproProcessor erben sich im Netzwerk anmelden und warten, um ankommende

Verarbeitungsanforderungen.

Beispiele für Klassen, die von DiauproProcessor erben sind DiauproVCO-Processor und DiauproVCAProcessor. Sie werden unten in Detail beschrieben.

4.2.5 DiauproVCOProcessor

Die DiauproVCOProcessor erbt von DiauproProcessor und implementiert die Oszillatorblock eines virtuellen synthsizer. Es erzeugt einen Klang in der Tonhöhe einen gespielten MIDI-Note. Die einzigen Funktionen aus DiauproProcessor, die überladen werden müssen, sind "localProcess", "getState", "getStateSizeünd "getServiceTag".

"localProcess"wird wie folgt implementiert:

```
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 }
```

Wenn die Funktion aufgerufen wird, wird die aktuelle Audio-Puffer übergeben, ein Puffer von MIDI-Events für den korespondierende Zeitrahmen, und ein Zeiger auf einen Block von Daten, die verwendet werden, um den Zustand zu speichern, damit es zwischen Aufrufen beibehalten werden.

Wird diese Funktion innerhalb eines Rechenknotens ausgeführt dann werden Audio-, MIDI- und Zustandsdaten aus einem DiauproMessage, als UDP-Paket von der Master-Audio-Plugin gesendet, entnommen.

Im obigen Beispiel wird einer Sinuswelle generiert, indem die Sinus Wert für jedes Element in der Audio-Puffer berechnet wird.

4.2.6 DiauproVCAProcessor

Die DiauproVCAProcessor moduliert die Lautstärke des Signals über Zeit. VCA ist die Abkürzung für Voltage Controlled Amplifier, dies bezieht sich auf den Verarbeitungsblock in einer virtuellen Synthesizer, der für die "Formëines Klangs zuständig ist. Eine Glocke, zum Beispiel, hat einen schnellen und lauten Anfangs-Ton, der über langen Zeitraum abklingt. Im Gegensatz dazu ist ein Streicher Klang, wie eine Geige, hat, wenn sanft gespielt, eine langsame Anfangs-Ton, der plötzlich endet. Die VCA ist für die Gestaltung der Amplitude eines Klang zuständig.

Kapitel 5

Conclusion

5.1 Performance Evaluation

Eine Audio-Plugin läuft in einer Umgebung mit vielen anderen Komponenten mit dem es CPU-Ressourcen teilt. Die Verarbeitung von Audiodaten muss innerhalb diskreten, von der Audio-Hardware gesteuerten, Zeitintervallen abgeschlossen werden. Typische Audio-Sampling-Frequenzen sind 44,1 kHz, 48 kHz, 88,2 kHz und 96 kHz. Mit 44,1 kHz als Beispiel bedeutet dies, daß die für eine einzelne Abtastung erforderlichen Berechnungen innerhalb von 0,023 ms durchgeführt werden muss. Interrupts müssten von der Audio-Hardware in Abständen von 0.023ms getätigt werden, und dies würde das Betriebssystem der CPU überlasten. Stattdessen werden Anfragen für neue Audiodaten in Puffern gebündelt. Die Größe der Puffer ist ein Parameter, der vom Benutzer gestzt werden kann. In der Regel beträgt es 512 Abtastungen, kann aber bis 16 Abtastungen klein sein.

Erhöht man der Puffergröße steigt auch die Zeit, die die CPU hat, um die Audiodaten zu verarbeiten. Dies führt aber auch dazu das die Latenz das System steigt. Eine Puffergröße von 512 Abtastungen entspricht einer Latenzzeit von 11.60ms. Ein 16-Probenpuffer Größe entspricht 0.36ms.

Braucht ein einzelnes Plugin 1,0 ms auf seine Verarbeitung abzushliessen, dann wird es nicht rechtzeigtig beendet, wenn die Puffergröße zu niedrig eingestellt ist. Wenn die Puffergröße gerade hoch genug ist, dann bleibe nicht genügend CPU-Ressourcen übrig für andere Plugins, ihre Berechnungen zu auszuführen.

5.1.1 Synchronous Performance

Dieses Projekt verteilt die Verarbeitung an externen SBC-Geräte. Es gibt jedoch keine Entlastung , wenn die Audio-Plugin blockiert ist, während es für die Ergebnisse aus dem SBC-Gerät wartet. Da die externe SBC-Geräte langsamer sind als der Haupt-CPU wird die Bearbeitungszeit sogar länger. Fügt man der

Zeit hinzu, die zur Serialisierung und Deserialisierung das Datagramm-Paket an jedem Ende benötigt wird, wird die Leistung sogar noch verschlechtern.

Abbildung 5.1 zeigt das Problem genauer.

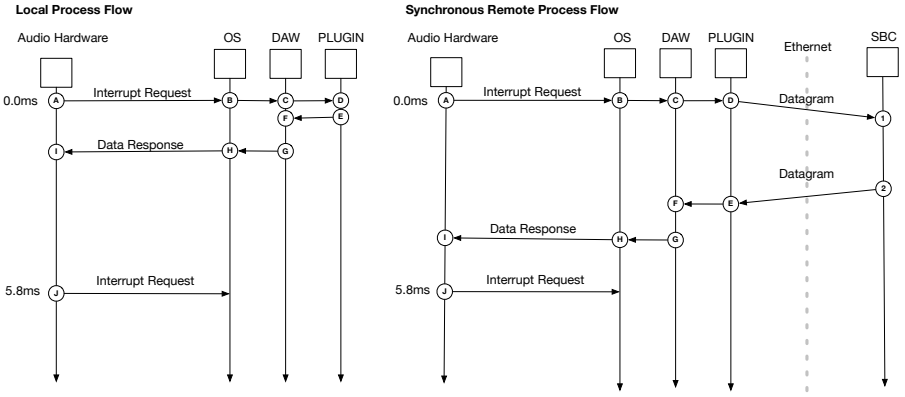

Abbildung 5.1: Local vs Remote Synchronous Processing

Das Beispiel zeigt die Audio-Hardware Abfragen an das Betriebssystems, in Zeitintervallen von 5,8 ms getacktet. Dies entspricht einer Puffergröße von 256 Abtastwerten. Das Zeitintervall zwischen den Staaten D und E repräsentiert die Zeit, die ein Plugin braucht, um einen Puffer von 256 Proben zu verarbeiten. Die DAW-Anwendung führt andere notwendige Audio-Verarbeitungsfunktionen in dem Intervall zwischen F und G. Nach Zustände G und H sind der DAW-Anweung und Betriebssystems wieder in der Lage andere Dinge zu tun, wie z.B. die GUI zu aktualisieren.

Mit der synchronen verteilte Verarbeitung wird der Zustand E blockiert bis Zuständen 1 und 2 abgeschlossen sind. Wenn diese Zeit signifikant ist dann hindert dies die DAW-Anwendung und Betriebssystem andere wichtige Aufgaben durchzuführen.

buffer size	limit (ms)	rtTime (ms)	pTime (ms)	tTime (ms)	% of limit
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

Tabelle 5.1: Measured Times for Synchronous Processing

Tabelle 5.1 zeigt die gemessenen Zeiten für verschiedene Puffergrößen in der synchrone Ausführung, in der keine tatsächlich Audioverarbeitung statfindet. Nur die Vorbereitungs- und Tranport-Zeiten werden berücksichtigt. Ein Audiosystem läuft mit einer Puffergröße von 64 Proben hat 1.451ms, um alle Aufgaben zu erledigen. "rtTimeïst die gemessene Gesamtumlaufzeit. Dies entspricht der Zeit zwischen den Staaten D und E des synchronen Verfahrens in Abbildung 5.1. "pTimeïst Zeitintervall zwischen den Zuständen 1 und 2 auf der SBC-Gerät, den Zeitaufwand für die Verarbeitung der Daten. In diesem Fall gab es keine Verarbeitung, so ist dies nur die Zeitaufwand für die Deserialisierung und Serialisierung der Datagramme zu Audio- und MIDI-Daten. "tTimeïst der "ptimeßubtrahiert von der "rtTime"welche das Verpackungsaufwand entspricht, um Daten zu senden und empfangen.

Die letzte Spalte, "% of limit", zeigt wieviel Zeit insgesamt als prozentualer Anteil der "limit"verbraucht wurde. Bei einer Puffergröße von 64 Abtastungen wird 39.59 % Prozent der Gesamtzeit für einer einzigen Plugin verbraucht. Es bleibt nicht viel Zeit übrig für andere Prozesse auf der CPU. Für eine Puffergröße von 512 Samples ist der "% der Grenze"Wert viel niedriger. Dafür ist aber die Gesamtsystemlatenz geringfügig höher als die angestrebten maximal von 10ms. Durch das synchrone implementation wird die Verarbeitungslast der CPU überhaupt nicht verringert .

5.1.2 Asynchronous Performance

Anstatt auf eine Antwort von dem SBC zu Warten wie im obigen synchrone Methode, prüfft die asynchrone Methode zuerst ob Daten vorhanden sind. Falls doch, gibt es diese zurück, falls nicht gibt es leere Daten zurück. Die erste Anfrage für Audiodaten würde leere Daten zurück geben, aber in der Zeit bis die zweite Puffer angefordert wird, kommen der Erste von der SBC-Gerät zurück. In beiden Fällen ist die einzige Zeit, die der Audio-Plugin selbst verbraucht, nur für die Serialisierung, Deserialisierung, und Transport der Daten. Das sind die Zeitabstände zwischen den Zuständen D und E und M und N dargestellt in Abbildung 5.2.

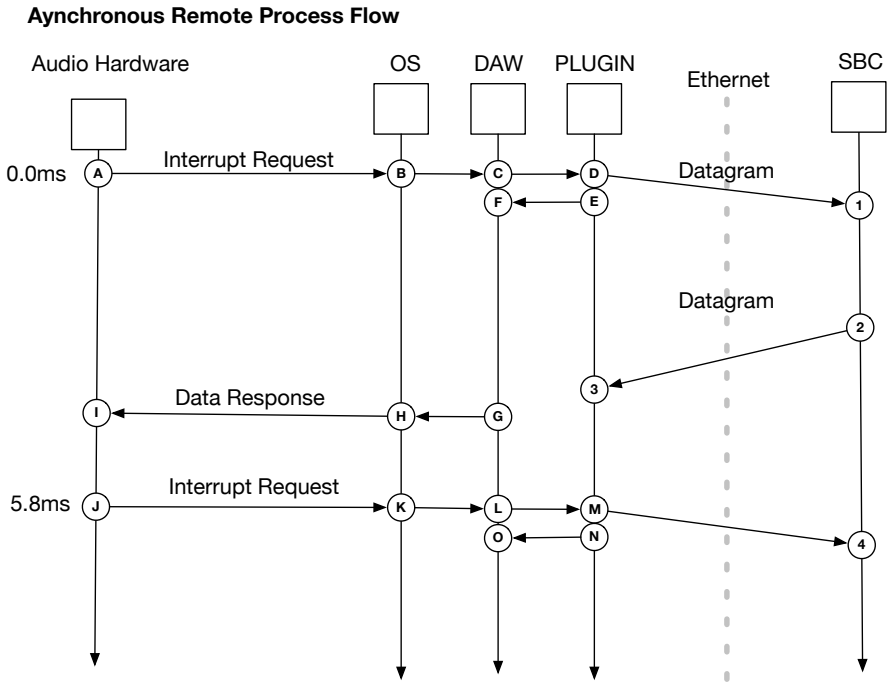

Abbildung 5.2: Local vs Remote Synchronous Processing

Bis der zweite Zyklus beginnt (J in Abbildung 5.2), sind die verarbeiteten Daten aus dem ersten Zyklus zurückgekehrt. Die Zeit zwischen den Ereignissen M und N ist nicht mehr proportional zur Zeit benötigt, um tatsächlich die Daten zu verarbeiten, sondern um die Zeit, für das Senden und Empfangen der Daten. Tabelle B zeigt die gemessenen Zeiten.

buffer size	limit (ms)	rtTime (ms)	pTime (ms)	tTime (ms)	% of limit
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

Tabelle 5.2: Measured Times for Asynchronous Processing

In der obigen Tabelle entspricht ”rtTime”nicht mehr der tatsächlichen Ge-

samtverarbeitungszeit. Da die Daten zum Zeitpunkt der Interrupt bereits vorhanden sind, kann das Plugin diese Daten sofort zurückzugeben. Dies geht auf Kosten der Latenzzeit welche jetzt genau einer Interrupt-Zyklus entspricht. Der Benutzer kann die Puffergröße des Audiosystems einstellen, dass dies weit unter der 10 ms Grenze ist, ohne dass mehr als 1,2 % der Bearbeitungszeit verwendet wird.

Die asynchrone Verfahren bringt einen echten Nutzen für die verteilte Verarbeitung in Bezug auf der Belastung der CPU. Der Preis ist jedoch eine Erhöhung der Latenzzeit des Plugins. Die verarbeiteten Audiodaten sind immer einen Puffer-Zyklus verzögert. Ein weiterer Nachteil ist, dass, wenn mehrere verteilte Prozessoren sind in einer Reihe verkettet sind, wie es in der Demo-Anwendung der Fall ist, dann ist die Latenzzeit ist kumulativ. Dies resultiert in einer Gesamtlatenz, der die "limitZeit multipliziert mit der Anzahl der Prozessoren im Plugin gleich ist.

5.1.3 Mögliche Optimierungen

Es gibt zwei Bereiche, wo unnötige Operationen durchgeführt werden und optimiert werden könnten. Die erste ist die Serialisierung und Deserialisierung von Daten, wenn sie von der SBC-Gerät zurückgegeben werden. Sind in der Plug-in mehrere Prozesse nacheinander verkettet, wird der zurück geschickte DiauproMessage zwischen jeder Prozess-Schritt unnötig deserialisiet und reserialisiert. Dies wäre eine relativ einfache Optimierung zu implementieren.

Die zweite Optimierung ist ähnlich, aber komplizierter zu implementieren. In einer Kette, nacheinandergeschalteten Prozessen müssten die Daten nicht nach jedem Schritt an das Master Plug-in zurückgeschickt werden, sondern direkt an den nächsten Knoten. Wenn der nächste Knoten auf dem gleichen SBC-Geräts befindet wird eine zusätzliche schritt über Ethernet ersparrt. Außerdem könnte auch die Serialisierungs Schritten zwischen den Knoten übersprungen werden, wenn die Knoten ihre Berechnungen direkt auf die Daten in dem DiauproMessage ausführen. Hierzu müsste die DiauproMessage erweitert werden, um Routing-Informationen zu erfassen, so dass jeder Rechennoten weiß, wo die Daten als nächstes geschickt werden müssen.

Obwohl komplexer, die zweite Optimierung hat den zusätzlichen Vorteil, dass die Latenzzeit sich nicht um ein Vielfaches der vollen Zykluszeit zwischen Interrupt-Aufrufe multipliziert wird. Die endgültigen Daten könnten nach einem einzigen Interrupt-Zyklus zurück gegeben werden.

5.2 Summary

Die asynchrone Umsetzung bietet deutliche Potenzial. Die Latenz scheint der von kommerziellen DSP-basierten Systemen zu entsprechen [8] . Dies ist ermutigend, insbesondere unter Berücksichtigung der Erweiterbarkeit des SBC-basiertes System, und die Kompatibilität der Codebasis. Zwei entscheidende Vorteile, die der SBC-System über DSP-basierten Systemen hat.

Kapitel 6

Glossary

MIDI

The Musical Instrument Digital Interface specification, first introduced in 1983 defines an 8-bit standard for encoding and transmitting music notes. It's original purpose was to allow one keyboard based synthesizer to controll other music devices. [2] Allthough the specification also describes the hardware and wiring for daisy chaining instruments in a midi "network"most midi communication today transmitted via usb or virtually between audio software.

Zeroconf

Zeroconf is a network service discovery protocl. It is also known as Bonjour, and occasionally refered to as Rendezvous, from the Apple implementations. Howl and Avahi are alternative open source Zeroconf implementations for Linux. Application can use Zeroconf to register or browse for service on a network without the need for a user to provide a specific IP Adress or port number. [4]

AES67

Latency

VST

SBC

AVB

Literaturverzeichnis

[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] Richard Boulanger and Victor Lazzarini. *The Audio Programming Book*. The MIT Press, 2011.

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

[4] Stuart Cheshire and Daniel H. Steinberg. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., first edition edition, 2006.

[5] 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.

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

[7] 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.

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

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

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

Kapitel 7

Appendix A

7.1 Compiling the Source Code

Download the Juce C++ Library from GitHub

github.com/julianstorer/JUCE

Additionally download and install the DrowAudio Juce Module Extensions

github.com/drowaudio/drowaudio.git

7.2 Bonjour

Instructions for installing bonjour:

If bonjour / zeroconfig is not installed on the SBC device you will get a "fatal error: dns_sd.h: No such file or directoryërror. You can fix this by installing the Avahi library

http://www.avahi.org/download/avahi-0.6.31.tar.gz

Requirements: sudo apt-get install intltool sudo apt-get install libperl-dev sudo apt-get install libgdbm-dev sudo apt-get install libdaemon-dev sudo apt-get install pkg-config sudo apt-get install libgtk2.0-0 sudo apt-get install libdbus-1-dev

sudo apt-get install intltool libperl-dev libgdbm-dev libdaemon-dev pkg-config libgtk2.0-0 libdbus-1-dev

./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 make install

sudo cp /usr/include/avahi-compat-libdns_sd/dns_sd.h /usr/include/

7.3 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