# MIDI or Die

Oren Bell*
oren.bell@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

Dane Johnson*
dane@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

Marion Sudvarg*
msudvarg@wustl.edu
Washington University in St. Louis
St. Louis, Missouri, USA

## ABSTRACT

We present MIDI or Die, a soft real-time platform that converts analog audio into Musical Instrument Digital Interface (MIDI) streams. The target application is a music rhythm video game played with real instruments. One or more musical instruments are connected to USB sound cards on Raspberry Pi devices, which serve as video game controllers. Each controller performs a Fast Fourier Transform (FFT) in real-time against the digital audio signal, then sends that data over the local area network to a server. The server can multiplex several FFT streams in real-time, using a Deep Neural Network to extract individual tones. Tones are converted to MIDI notes, then forwarded to a custom video game, which uses the received notes to control a simulated guitar fretboard. This provides visual feedback to players as they play along with a song.

## KEYWORDS

fourier transform, midi, note extraction, soft real-time, video game development

## 1 INTRODUCTION

Electronic music composition is a discipline at the cutting edge of both artistic expression and computer science. Digital composers often find themselves in need of specialty playing and recording hardware, e.g. MIDI controllers, cables, and hardware or software synthesizers. Many MIDI controllers are implemented similarly to electric pianos or organs, consisting of a piano keyboard (often only a few octaves of keys), knobs, sliders, buttons, and other controllers. While immensely flexible, the suite of hardware required can be costly, and constrains the musician to the available media and interfaces. For example, a musician experienced with playing a guitar must learn a new operational mode (i.e. playing a piano keyboard) to successfully record electronic music.

As digital music technology has progressed, music rhythm video games have entered the market that provide players with the means to "play" along to popular songs. In the popular Guitar Hero game[8], the player is a musician put on a stage in front of a virtual audience. The screen shows a simulated guitar fretboard, and the player must play the indicated notes in time with a song. The specialized controllers for these games are drastically simplified and aren't interchangeable with an actual instrument.

To address both of these issues, we present MIDI or Die, a novel approach to controlling digital music software with analog instruments. With minimal hardware, MIDI or Die converts audio signals from one or more analog instruments into streams of MIDI notes. The MIDI stream can then be fed into music composition software (such as in [13]) or a video game that accepts MIDI notes as control events. The MIDI or Die package includes a simple video game that, like Guitar Hero, presents a guitar fretboard on the screen and prompts players to play along to a song.

Because music composition and performance requires auditory feedback to the musician (the composer must hear the notes they are playing, and the player must hear the song with which they are to play along), MIDI or Die involves time sensitive computation. Indeed, a trained musician is said to be able to detect audio latency greater than around 20ms[5, 11]. It is thus necessary that we employ the techniques of real-time scheduling in order to ensure that the musician can comfortably compose and record. To this end, MIDI or Die has been implemented as a soft real-time distributed system using a number of design patterns to ensure end-to-end deadline guarantees. In testing, we are able to demonstrate predictable latency under the bounds we impose.

The rest of this paper is organized as follows: In Section 2, we describe the goals and requirements for MIDI or Die and its accompanying video game. In Section 3, we describe our design process. In Section 4, we describe the details of our program implementation. In Section 5, we present experimental evaluation of MIDI or Die's accuracy in extracting individual notes from an audio signal, and results of several execution time and latency measurements. In Section 6, we discuss our conclusions, lessons learned, and ideas for future work.

## 2 GOALS AND REQUIREMENTS

MIDI or Die, at its core, is a system that allows musicians to generate MIDI notes by playing real musical instruments. The MIDI stream, then, should be valid input for any MIDI software, including MIDI transcription (e.g. [13]) and software synthesizers (e.g. Fluidsynth[1]).

Additionally, it should allow multiple instruments to be played simultaneously; therefore, MIDI or Die must be implemented as a distributed system. One or more controllers, acting as client devices, each receive audio signals from an individual instrument.

The overall design goal of this project leads to certain timing requirements. Each controller has to sample its audio signal fast enough to perform frequency analysis (e.g. a Fast Fourier Transform[7]) at the desired rate. The human ear can hear auditory latencies greater than 20ms, making this a natural latency target. However, due to physical constraints, we targeted a 40ms deadline, as detailed in Section 3.1.3. This defines the real-time requirements for our system: each controller represents an implicit-deadline periodic task with a period of 40 ms. If a job completes more than 40ms after it is released (i.e. after an audio sample becomes available), the latency will become auditory, degrading the performance of the

*All authors contributed equally to this research.

*CSE 520S SP21, Spring, 2021, Washington University in St. Louis*

[1]https://www.fluidsynth.org/

system. Because a single late result does not cause system failure, these are soft real-time tasks.

We also define a number of functional requirements for our system. First, the controller should ideally support any non-percussive instrument. However, due to time constraints, we focused on support for the guitar (both acoustic and electric). For the target instruments, MIDI or Die should be able to extract notes and simple chords from the audio signal. The goal is to support chords having notes as high as a $C_6$, which is around 1kHz, and represents the highest note playable on many guitars.

Second, besides supporting existing MIDI software, MIDI or Die also includes a music rhythm video game. This game should allow players to play along with real songs, providing visual feedback according to how well the player matches the given notes.

## 3 DESIGN

MIDI or Die has been designed in a modular fashion using object-oriented techniques to ensure that functionality could be easily swapped out, and different techniques compared. The server and controller use concurrent design patterns to provide a platform for this modularity, and to guarantee robust execution with predictable timing.

### 3.1 Software Components

The primary software components of our system include: (1) a sound server library to interface with the audio input driver, (2) an FFT module, (3) software to extract individual tones or chords from a DFT, and (4) tone and chord to MIDI note conversion.

*3.1.1 Sound Server Library.* MIDI or Die aims to support a variety of musical instruments and a wide range of hardware devices (e.g. analog audio outputs, USB-based analog-to-digital converters and preamplifiers, microphones, etc.). To this end, MIDI or Die uses the PulseAudio[2] sound server library to interface with common Linux audio drivers, allowing it to sample audio data coming in via any supported input device.

PulseAudio provides a hook in the form of a callback function signature. The callback function we implement collects audio samples at our hardware's recommended rate of 44.1kHz, then passes sampled audio data to the FFT module.

*3.1.2 FFT.* To ascertain the notes being played, the audio signal is passed through an FFT algorithm, generating a Discrete Fourier Transform (DFT) representation of our audio. These acronyms – FFT and DFT – will be used interchangeably.

The FFT algorithm is provided by the FFTW3 library[4]. The audio segment is preprocessed through application of a Hamming window[3], then the FFT is computed. The result is an array of complex numbers containing phase information.

The FFT module then determines the magnitude of each complex value. The resulting real-valued array is a histogram of amplitude values over the frequency domain, with each bin representing a 25Hz range. Amplitude values are then expressed in decibels.

Due to the hardware sampling rate, the resulting DFT histogram will incorporate frequencies as high as 22050Hz. Because we only

[2]https://www.freedesktop.org/wiki/Software/PulseAudio/

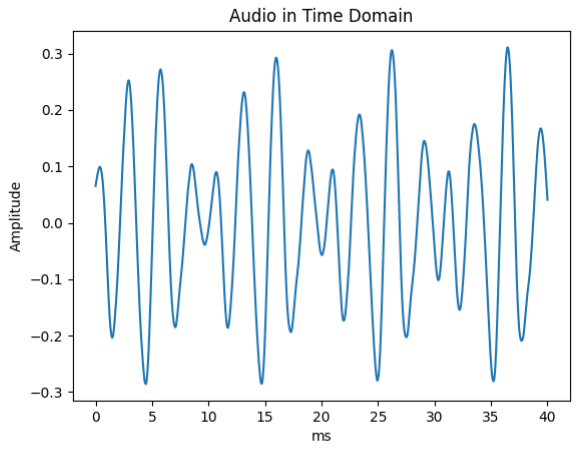

**Figure 1: Audio of G-D interval**

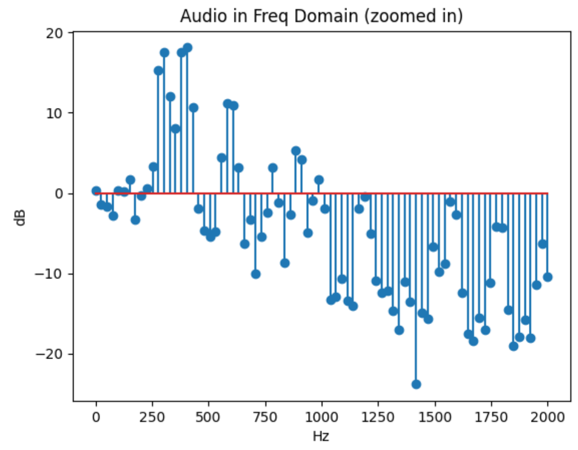

**Figure 2: Frequency domain representation, in decibels**

target notes up to a $C_6$, we then crop the histogram to 2kHz, approximately at $C_6$'s second harmonic.

See Figs 1 and 2 for an end-to-end visualization of this process.

*3.1.3 AI Model for Tone and Chord Extraction.* Given an array of DFT values, a naïve way to extract the note being played is to find the most prominent frequency peak. But this runs into a fundamental problem of mathematics. The finest resolution of the frequency transform is the reciprocal of the duration of the audio window used to calculate it[7]:.

$$\Delta_t N = 1/\Delta_f \quad (1)$$

Here, $\delta_t$ and $\delta_f$ are the width of a sample, in time and frequency domain, respectively; and N is the number of samples, which is identical for a signal and its frequency transform.

Incidentally, the two lowest notes on the guitar are E2 and F2, which are 5Hz apart. In order to distinguish these, the audio sampling window must be at least 200ms wide, an unacceptable delay for audio applications. Since this project has a video game aspect, and the refresh rate of monitors would constrain us anyways, we compromise on a 40ms window.

To target a 40ms window, corresponding to a 25Hz resolution, we modified our approach to DFT analysis. Instead of just searching for fundamental frequencies, we consider the entire DFT, including overtones. The $6^{th}$ harmonics of $E_2$ and $F_2$ are 30Hz apart, enough to distinguish with a 40ms audio window.

We implemented a Deep Neural Network in TensorFlow[1] to perform the DFT analysis and infer the notes played. To train the network, we wrote a script to generate thousands of different guitar chords, record them, and store the resulting DFT data with the corresponding notes as labels. Each chord was generated using dozens of unique sound fonts. This approach resulted in a sufficiently large quantity of data to allow for robust training of the network.

*3.1.4 MIDI Note Conversion.* Notes extracted from DFT data are sent to the target MIDI application as MIDI messages. We focus on sending note-on and note-off events. To generate these events, the server maintains parameters in memory that track the most recent notes that were played. When a new note is detected, it publishes a corresponding “Note On” event to the MIDI stream. When a note is no longer being played, it generates a “Note Off” event. No action is taken for detected notes that were also present in the previous DFT. Note that, for now, the server does not infer volume; any notes detected are sent with a constant value for velocity.

## 3.2 Network Architecture

MIDI or Die also provides a custom library, built on top of BSD stream sockets, for network communication between the client and server. The library implements Server and Client classes, both deriving from a base Socket class. The Server implements the Acceptor as part of the Acceptor-Connector design pattern[10], having a dedicated thread to listen for client connections on the socket. It constructs an object of the Connection class for each client it accepts. This class implements the Active Object pattern[10], using a dedicated thread per connection to read and write across the socket. The Connector’s thread function is a parameter of the Socket constructor. This allows the library to be highly flexible: custom communication protocols and functionality can be developed separately of the socket communication itself, then simply passed to a single Server object which then handles all communication.

The Client class, unlike the server, only maintains a single Connection object, which is constructed upon successful connection to the server. This allows all socket communication on the client side to be handled in its own thread. If the Client object cannot connect to the Server, it throws an exception, with a unique type for each cause of failure. This allows the library user to set up custom exception handling. In our case, for example, the client attempts to reconnect every second if there is no listening Server. For other errors where an attempt to reconnect would be futile (e.g. an error in the socket subsystem, an invalid IP address, etc.), the program is terminated gracefully with an appropriate error message.

The network architecture is illustrated in Figs 3 and 4.

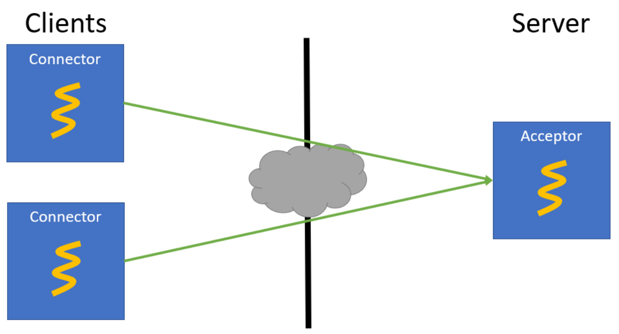

**Figure 3: First, the Client establishes a connection with the Server’s Acceptor thread.**

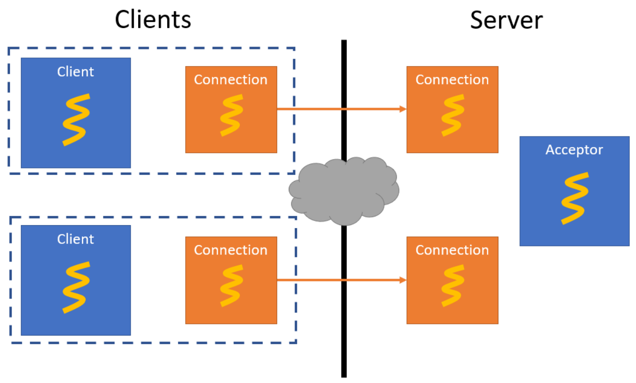

**Figure 4: Once established both Client and Server create dedicated Connection Active Objects to handle bidirectional socket communication.**

## 4 IMPLEMENTATION

MIDI or Die is implemented as a distributed soft real-time system with two basic components: one or more client devices acting as video game or MIDI controllers, and a server hosting the target MIDI application or video game.

## 4.1 The Controller

The client device acts as both a video game and MIDI controller. Its target platform is meant to be lightweight, inexpensive, and headless. We target the Raspberry Pi platform, since it provides an inexpensive, easily accessible multicore system on a chip. Additionally, the Raspberry Pi OS has the necessary audio drivers to run many off-the-shelf plug and play USB audio adapters. As implemented, the controller runs the sound server library and FFT module, then sends DFT data over a local area network (LAN) to the server.

The controller constructs an object that encapsulates the FFT functionality and implements the callback function to PulseAudio. It also constructs a Client object to connect to the server. The function passed to the Client first waits for the server to be ready to receive data (indicated by a single byte sent from the server to the controller), then sends DFT data to the server every 40ms.

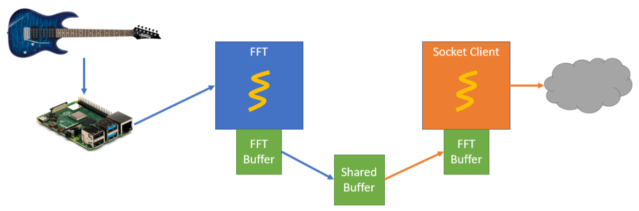

**Figure 5: Controller Architecture**

Because DFT data is processed by the main thread of execution, then sent by the socket thread, we have to ensure that data is shared between the two threads in a consistent state. As such, we implement a custom Shared_Array class, which creates a buffer for which reads and writes are guaranteed to be consistent. Consistency is maintained in two ways, allowing for flexibility of use: the act of access can be locked by a mutex, using the C++ Standard Library mutex object. It can also be maintained using a mechanism similar to Linux's seqlock[6].

The described controller architecture is illustrated in Fig 5.

### 4.2 The Server

The server requires more computational power than the controller, and is intended to be installed on a Debian-based desktop or server machine with the necessary peripherals (keyboard, mouse, monitor, speakers) to interact with the target MIDI application. It receives streams of DFT data from one or more controllers on the LAN, performs tone and chord extraction for each one, and converts the extracted tones to MIDI notes. MIDI notes are then forwarded to the target MIDI application or video game, which runs on the same machine.

The server software constructs a Server object to accept new client connections. Each resulting Connection object thread function executes a loop. Every 40 ms, it reads DFT data from the socket into a Desynthesizer object that encapsulates the DNN for tone and chord extraction and the MIDI note conversion. The Desynthesizer then sends messages to a MIDI port specified as a command-line argument. On a system running one or more MIDI applications, each application binds to a unique port number.

If multiple controllers are connected, notes from each connection will be sent to a unique MIDI channel number. This is consistent with the semantics of traditional MIDI recording. To allow for automatic channel selection, we implement two classes: a Channel and a ChannelBroker. As new controllers connect, the ChannelBroker assigns each one a Channel object corresponding to the next free MIDI channel. MIDI provides 16 channels, and when no more channels are available, subsequent controllers fail to connect. The ChannelBroker uses appropriate locking semantics for thread safety.

### 4.3 MIDI or Die (The Game)

MIDI or Die comes with a custom music rhythm game, aptly named "MIDI or Die (The Game)." The game was implemented in the Godot game engine.[3] We modified the source code to use the RTMidi

[3]https://godotengine.org/

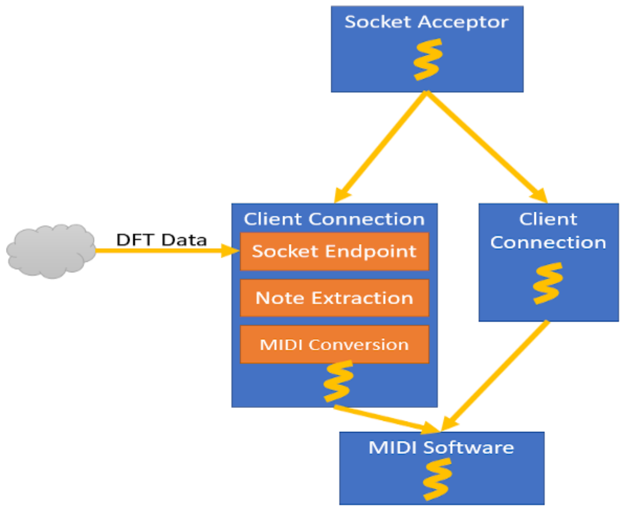

**Figure 6: Server Architecture**

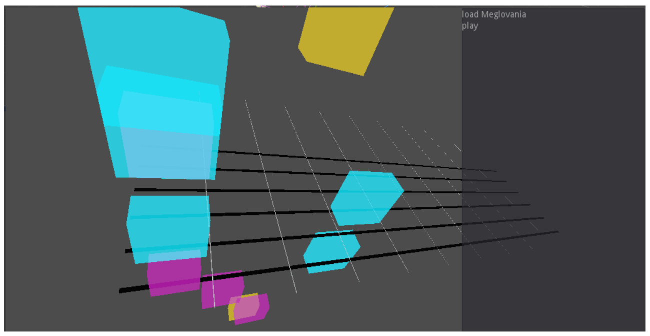

**Figure 7: MIDI or Die (The Game)**

library as an input mechanism. We added a C++ module to read Midi input from files exported by MuseScore, and we create physics objects in time with the tempo of the Midi file. Midi input creates a collision area which causes the physics objects to explode when they come in contact with it, and then removes the area after a third of a second.

## 5 EVALUATION

To evaluate our success in meeting the design goals of MIDI or Die, we tested both the execution time and end-to-end latency of the system, as well as the accuracy of our AI model at correctly identifying notes.

### 5.1 Execution Time and Latency

We evaluated execution times for different subtasks of our program, as well as testing end-to-end latency and polling jitter. For these tests, we used a single Raspberry Pi 3 Model B+, running the Raspbian GNU/Linux 10 operating system, as a controller. The server was installed on a system with 2 Intel Xeon Gold 6130 16-core CPUs running at 2.10GHz with Hyperthreading enabled and 32GB of RAM. Both the controller and server were connected to a gigabit Ethernet switch. All timing measurements were taken using the C++ Standard Library's High Resolution Clock.

*5.1.1 Execution Times.* We measured the time it took for the controller to perform an FFT for 500 runs of the algorithm. Results are illustrated in Fig 8. Average execution time was 558$\mu$s with a standard deviation of 57$\mu$s and a maximum execution time of 731$\mu$s. Compared to our 40ms deadline, this is very fast. It demonstrates that even the Raspberry Pi platform has sufficient computational power to perform an FFT well within our real-time constraints.

On the server, we measured the time to run the DNN to extract notes and convert them to MIDI. For 100 trials, we observed an average execution time of 569$\mu$s with a standard deviation of 111$\mu$s and a maximum execution time of 1.44ms. Results are illustrated in Fig 9. Even with the relatively high standard deviation, we expect this to remain well under the 40ms constraint, even if multiple controllers are connected or running on a more constrained host machine.

We also modified the Godot game engine to allow our game to receive custom MIDI events with timing information. This allowed us to measure the elapsed time between our server sending a MIDI note to the game, and the game receiving and processing the note. For 25 trials[4], we observed an average latency of 214$\mu$s with a standard deviation of 39$\mu$s and a maximum of 598$\mu$s. Results are illustrated in Fig 10. Again, these times are well within our deadline.

*5.1.2 End-to-End Latency.* We next measured end-to-end latency from when our controller receives a 40ms audio sample from PulseAudio, to when the corresponding MIDI note is sent to the game, taking into account the network latency. To do so, we modified the server to send a single byte back to the controller after it sends each MIDI event to the game. While we expect this to be an inflated measurement (since it includes the transit time of this status byte, which is not otherwise sent by the application), we do not expect the transit time of a single byte on the network to take long (indeed, ICMP packet roundtrip times were measured to be consistently under 1ms between the controller and server).

The results were surprising: for 500 jobs, roundtrip latency measurements started at only 664$\mu$s, well under our 40ms deadline. However, they grew in a linear fashion to around 18.5ms for the last measurement. Results are illustrated in Fig 11, which plots roundtrip latency measurements against the corresponding job number to illustrate the growth. This means that roundtrip latency increased by about 35.8$\mu$s. We assume, given this trend, that we will begin to miss the 40$\mu$s after around 1100 polling periods (or 45 seconds of playing time). We discuss possible reasons for this phenomenon in Section 6.

*5.1.3 Polling Jitter.* The controller is designed to send DFT data to the server every 40ms. The server, meanwhile, must receive the data and process it with the same polling period. For both the controller and server, we measured the elapsed time between 200 subsequent polling periods. The results of these tests are illustrated in Figs 12 and 13.

[4]This number comes from real gameplay, and corresponds to the number of notes actually detected and sent.

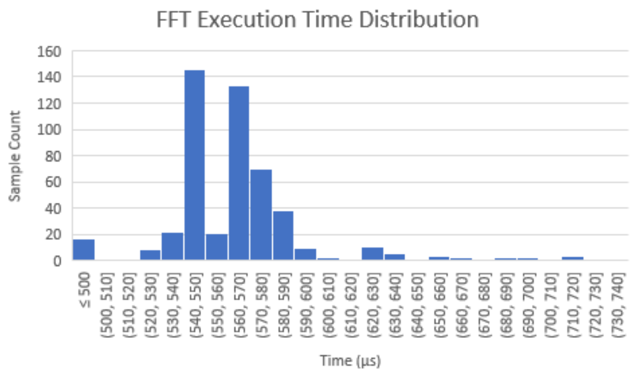

**Figure 8: FFT Execution Times**

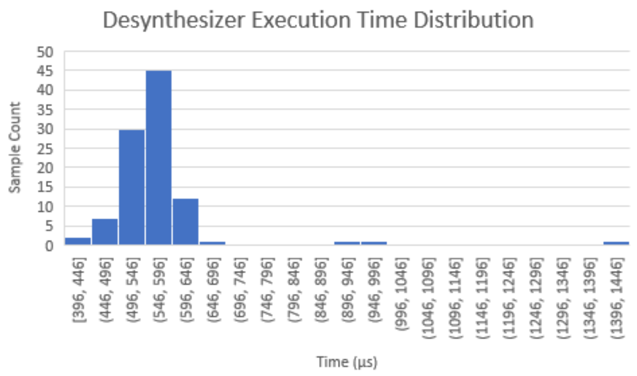

**Figure 9: DNN Note Extraction and MIDI Conversion Execution Times**

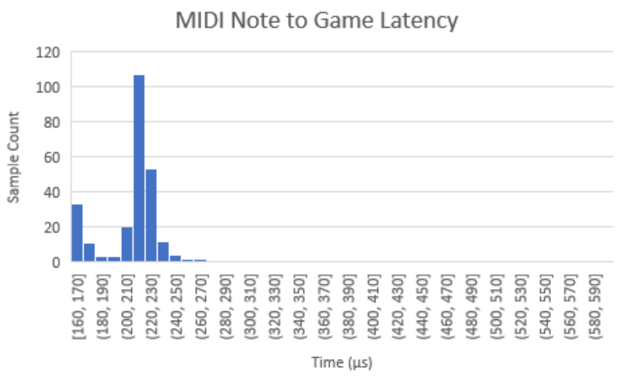

**Figure 10: MIDI Note to Game Latencies**

For the controller, the mean polling period was 40.12ms with a standard deviation of 0.11ms. For the server, the mean polling period was 40.11ms with a standard deviation of 0.20ms. While these values are not far from our target 40ms, they will begin to introduce skew against the desired period, and might contribute to the increase in round-trip latency that we observed.

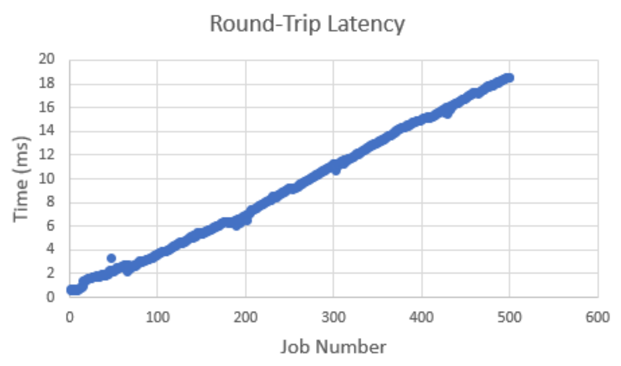

**Figure 11: Rountrip Latency Measurements. Notice that these are plotted against job number to illustrate an increasing trend.**

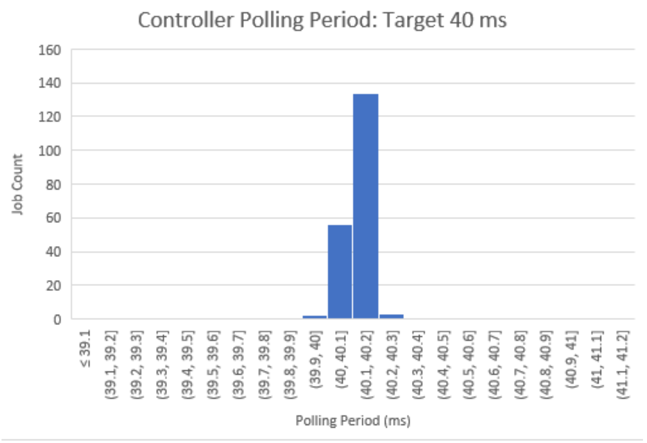

**Figure 12: Controller Polling Times**

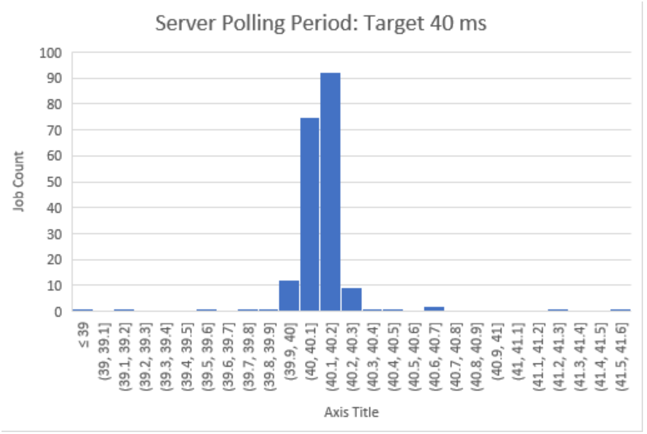

**Figure 13: Server Polling Times**

## 5.2 DNN Note Extraction Accuracy

The AI model, which was created to discern low notes, is ironically bad at identifying low notes. When we record silence, the sub-100Hz range of sound will appear 10-20dB louder than other frequencies, due to ambiance conforming to the curve of pink noise[2].

Incidentally, the AI model uses layer normalization. In layman's terms, this means that the AI looks at the shape of the input rather than the absolute values. This result is that that silence at -40dB will be interpreted similarly to E2 playing at 110dB, even though it's 1 quadrillion times louder. The result is that after training, the AI model would discard most notes played on the 6$^{th}$ string as silence.

Potential fixes for this issue would likely be representing our FFT in linear terms, rather than decibels.

Below is the result of testing our AI model. False positive rate is the percentage of incorrect notes detected out of all notes detected (smaller is better). Detection rate is notes that were detected out of all correct notes expected (larger is better).

| | |
|---|---|
| False Positive Rate | 15% |
| Detection Rate | 73% |

## 6 CONCLUSIONS AND FUTURE WORK

This paper presents MIDI or Die, a platform for using real musical instruments as MIDI controllers. Its goal was to support a broad range of instruments, and to guarantee sufficiently low end-to-end latency to provide players with a smooth experience. While it meets several of its goals, MIDI or Die is still a work in progress.

MIDI or Die's deep learning model supports both electric and acoustic guitars, but has not yet been adequately trained to support a broader range of instruments (e.g. pianos, synthesizers, etc.). Additionally, there is still room for improvement in both the model, and the corpus of training data provided to it.

We also identified a timing issue that skews the polling periods used in MIDI or Die, causing round-trip latencies to increase over time. The way we currently enforce polling is with a custom class that forces a loop to delay a specified duration of time from when it begin its current iteration, to when it begins its next iteration. The way this class is implemented could be improved: the delay enforcement does not take into account context-switching and loop branching times. We would need to improve this for a production release.

Given that this project is an interdisciplinary work of software engineering and musical composition, it is lacking the critical perspective of a trained musician. The authors, while capable, are only hobbyists. In the future, we would like to include a trained musician and music producer, who would be able to bridge the gap between the technical and artistic aspects of the project and the video game.

In a similar vein, the authors are only hobbyist game developers, and including artists from that sphere could much improve the aesthetic and entertainment value of "MIDI or Die (The Game)." We would also like to expand "MIDI or Die (The Game)" to support multiple players, since the MIDI or Die server software supports multiple controllers. In doing so, we would need to perform more timing measurements to determine the maximum number of controllers supported before execution times exceed the 40ms deadline.

## ACKNOWLEDGMENTS

Authors would like to thank Phil Ring for his insight into the acoustics of the guitar.

## REFERENCES

[1] M. Abadi, et al. "TensorFlow: A system for large-scale machine learning," 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association (2016), pp. 265-283.
[2] P. Bak, C. Tang, K. Wiesenfeld (1987). "Self-Organized Criticality: An Explanation of $1/f$ Noise," in *Physical Review Letters.* 59 (4): pp. 381–384.
[3] E. Douglas. *Handbook of Digital Signal Processing.* Academic Press, Cambridge, MA, USA, 1987. ISBN 978-0-08-050780-4.
[4] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3," in textitProceedings of the IEEE, vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.840301.
[5] H. Haas. "The Influence of a Single Echo on the Audibility of Speech," in *J. Audio Eng. Soc.*, vol. 20, no. 2, pp. 146-159, (1972 March.).
[6] C. Lameter. "Effective synchronization on Linux/NUMA systems." Gelato Conference, 2005.
[7] A. Oppenheim, *Signals and Systems, Second Edition.* Prentice Hall, Hoboken, NJ, USA, 1997. ISBN 0-13-814757-4.
[8] C. Roper. "Guitar Hero," IGN. Written Nov 2, 2005. Updated Nov 24, 2018. Retrieved May 7, 2021 from https://www.ign.com/articles/2005/11/03/guitar-hero
[9] G. Scavone and P.R. Cook (2005). "RtMidi, RtAudio, and a Synthesis ToolKit (STK) Update," In *Proceedings of the 2005 International Computer Music Conference*, Barcelona, Spain, pp. 327-330.
[10] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Volume 2. Wiley & Sons, New York, 2000.
[11] N. Schuett. "The effects of latency on ensemble performance," Doctoral Thesis, 2002, Stanford University.
[12] "Standard MIDI-File Format Spec. 1.1." The International MIDI Association, 1999.
[13] M. Watson. "MuseScore," in *Journal of the Musical Arts in Africa*, 15:1-2, pp. 143-147, 2018, DOI: 10.2989/18121004.2018.1534342

## A COMPILING AND RUNNING SUBMITTED CODE

### A.1 Platform Requirements

The system was built and tested using Ubuntu for the server machine, and a Raspberry Pi running Raspbian for the controller. We suggest testing it on only Debian-based systems, to maintain compatibility with the TensorFlow libraries.

### A.2 Retrieving MIDI or Die

MIDI or Die is available at https://github.com/msudvarg/MIDIOrDie. To retreive it, run the following commands:

```
git clone https://github.com/msudvarg/MIDIOrDie
cd MIDIOrDie
git submodule init
git submodule update
```

### A.3 Compiling Controller and Dependencies

The controller depends on PortAudio and the FFTW3 library. Install them and then build the repo with:

```
sudo apt install -y build-essential cmake \
        libfftw3-dev portaudio19-dev
mkdir build && cd build
cmake ..
make
```

### A.4 Compiling Server and Dependencies

The server also requires PortAudio, in addition to several TensorFlow libraries. MIDI libraries are bundled with the repo. If you built the controller on a Pi, switch to your server machine. Then install these dependencies with:

```
  # Install portaudio and co again,
  # if you're running on a different machine
  sudo apt install -y build-essential cmake \
          libfftw3-dev librtmidi-dev portaudio19-dev
  mkdir build && cd build

  # Install prereqs and bazel
sudo apt-get install -y cmake curl g++-7 git python3-dev \
        python3-numpy sudo wget libprotobuf-lite10 \
          libprotobuf10 libprotoc-dev
curl -fsSL https://bazel.build/bazel-release.pub.gpg | \
          gpg --dearmor > bazel.gpg
  sudo mv bazel.gpg /etc/apt/trusted.gpg.d/
  echo "deb [arch=amd64] \
          https://storage.googleapis.com/bazel-apt \
          stable jdk1.8" | sudo tee \
          /etc/apt/sources.list.d/bazel.list
  sudo apt-get update
  sudo apt-get install -y bazel-3.1.0
  sudo ln -s /usr/bin/bazel-3.1.0 /usr/bin/bazel

 # Clone a helpful repo from FloopCZ to install TF_cc
git clone https://github.com/FloopCZ/tensorflow_cc.git
  cd tensorflow_cc/tensorflow_cc
  mkdir build && cd build
  cmake -DALLOW_CUDA=off ..
  make
  sudo make install
```

With all those dependencies out of the way, build the repo with a flag to trigger building of the server

```
cd path/to/MIDIOrDie/
mkdir build
cd build
cmake -DBUILD_SERVER=true ..
make
```

### A.5 Running

The order of running is: host application, server, then controller. First start the host application (MuseScore, our MidiOrDie game, etc).

Run the server by navigating to the build directory and executing

```
./server -p 1 -a
```

This will start the server and publish to MIDI port 1. The server lists all available MIDI ports on launch, so if port 1 is not correct, simply kill the server with Ctrl+C and modify the argument given to -p.

[TODO: Detail all argument parameters?]

Next, run the controller. If it's running on a separate Pi, use:

```
./controller -f -i 192.168.x.x
```

Replacing 192.168.x.x with the IP address of your host machine. If the controller is also running on the host machine, omit the -i flag entirely, as it'll default to the local address.

[TODO: Detail all argument parameters?]

### A.6 Compiling and Running "MIDI or Die (The Game)"

"MIDI or Die (The Game)" runs on a modified version of the Godot game engine. This version must be built from source, available at https://github.com/dane-johnson/godot/tree/rtmidi. Instructions for compilation are available at https://docs.godotengine.org/en/stable/development/compiling/compiling_for_x11.html.

Furthermore, the game uses a C++ extension library that must also be compiled by hand. SCons must be run in the `midiordiethegame/MidiCPP/godot-cpp` directory, and then Make in the `midiordiethegame/MidiCPP` directory.

Once the game is loaded in the engine, it can be run simply by pressing the "play" button, the in-game console is loaded by pressing the "~" key, loading a song (Meglovania and Layla are shipped) is done with the "load <song>" command, and playback begins with the "play" command.