NDN-MIDI: Real Time Musical Instrument Control using Named Data Networking

MS Capstone Project

Kelly Flickinger
University of California, Los Angeles
UCLA ID: 803178628
kflickin@gmail.com

ABSTRACT

Recent advancements in mobile technology and music software have given rise to the "mobile musician," empowered with the ability to compose, produce, and perform music on their own using only a laptop or mobile device and a musical instrument input device (if needed). The Musical Instrument Digital Interface (MIDI) protocol has been the glue that makes this possible, providing a "standardized and efficient means of conveying musical performance information as electronic data" [1]. Since its inception in 1982 [1], many MIDI transport specifications have been developed, including those for physical cables, RTP, and Bluetooth, each with its own advantages and limitations. For the mobile musician, the RTP and Bluetooth network MIDI specifications are particularly enticing, providing a wireless means of transmitting MIDI data. Using these MIDI network specifications as inspiration, this paper introduces "NDN-MIDI," a macOS MIDI transport application utilizing the recently introduced future internet architecture "Named Data Networking" [2]. A brief overview of the MIDI protocol and its current transport specifications will first be given, followed by the design choices and challenges for NDN-MIDI and its implementation. The paper concludes with a discussion of future work as it relates to NDN-MIDI.

KEYWORDS

Named Data Networking (NDN), Musical Instrument Digital Interface (MIDI)

ACM Reference Format:
Kelly Flickinger. 2018. NDN-MIDI: Real Time Musical Instrument Control using Named Data Networking: MS Capstone Project. In *Proceedings of (MS Capstone Project).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

1.1 MIDI Protocol

The MIDI protocol was created in 1982/83 and is currently the industry standard for electronic musical instrument performance and control [1]. According to the MIDI specification, MIDI is a "standardized and efficient means of conveying musical performance information as electronic data. MIDI information is transmitted in MIDI Messages, which can be thought of as instructions which tell a music synthesizer how to play a piece of music" [1]. Therefore, digital audio itself is not transmitted in MIDI, but rather a playback device uses the MIDI messages to produce the notes itself. MIDI messages can be thought of as sheet music for a playback device to use. This approach drastically reduces the amount of information transferred in MIDI. According to the original MIDI specification, a typical MIDI message sequence for a minute of music may be 10 Kbytes, whereas digital audio might be 10 MBytes [1] (this was most likely using lossless audio). A typical MIDI system consists of three main parts: MIDI controllers, MIDI messages, and MIDI modules. MIDI controllers produce the MIDI messages and include hardware or software piano-type keyboards, drum triggers, sequencers, wind controllers, and many other instrument types. The MIDI messages encode the music performance data, not the actual audio itself. The most common MIDI messages include note on/off messages, and are typically 3-bytes in length. MIDI messages are received and interpreted by MIDI modules such as: hardware or software synthesizer modules, music composition software, sequencers, and many others. These modules use sound libraries to produce the notes encoded in the MIDI messages sent from the MIDI controller. A simple MIDI setup can be seen in Figure 1.

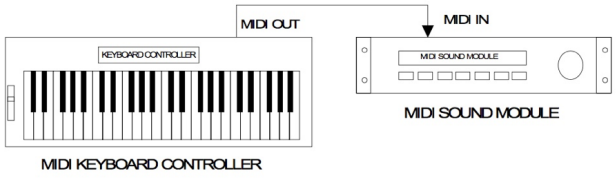

Figure 1: A simple MIDI setup example included in the original MIDI 1.0 specification. MIDI messages are sent from the "MIDI Out" port of the MIDI Keyboard controller to the "MIDI In" port on the MIDI Sound module. Currently, some/all of these components can be done in software.

Although MIDI was originally conceived as a protocol using a physical MIDI cable for connections between devices, there have been multiple attempts to provide wireless transports for MIDI connections using RTP and Bluetooth, each with their own limitations. In this paper, a new wireless MIDI transport application for macOS is introduced named "NDN-MIDI", which utilizes the recently introduced future Internet architecture "Named Data Networking" (NDN) [2]. It is assumed that the reader has a basic understanding of NDN, as described in [2]. This paper will discuss the topics below.

- Overview of the MIDI Protocol

MS Capstone Project, Winter 2018, UCLA
© 2018

- Evaluation of current MIDI transports
- Design choices and implementation of NDN-MIDI
- Future work and possibilities for NDN-MIDI

2 MIDI OVERVIEW

2.1 MIDI Messages

According to the MIDI 1.0 specification [1], a MIDI message "is made up of an eight-bit status byte which is generally followed by one or two data bytes." Messages can be divided into two main categories: Channel Messages and System Messages. Channel Messages apply to one of 16 specific MIDI channels, which could be used, for example, to distinguish among 16 different instruments in a MIDI system. Within the category of Channel Messages are Channel Voice Messages and Channel Mode Messages, Channel Voice Messages include Note On and Note Off Messages, as well are other musical performance information, and are the major types of messages used in NDN-MIDI. Due to the scope of this paper, System Messages, as well as many Channel Messages will not be discussed. The reader may see [1] for more information regarding MIDI messages. An example of a Note On MIDI Message can be seen in Figure 2. The first four bits of the Status Byte indicate that this is a Note On Message (a Note Off Message would start with "1000"). The last four bits of the Status Byte indicate which of the 16 possible channels this message is for. A playback module may, for example, have Channel 1 set to a piano sound and Channel 2 to a trumpet sound. The Data Bytes always start with a zero bit. The first Data Byte indicates which note (from 0-127) the message applies to. Although the note range is limited to 0-127, there are other means to expand the range that will not be discussed here. Data Byte 2 indicates the velocity (from 0-127) of the note to be played. The velocity can be thought of as the volume, with 0 being silent and 127 being the loudest. A Note Off Message can either have a Status Byte starting with "1000", or it can be a Note On Message with a velocity of zero (this is used for pipelining messages, which will not be discussed in this paper).

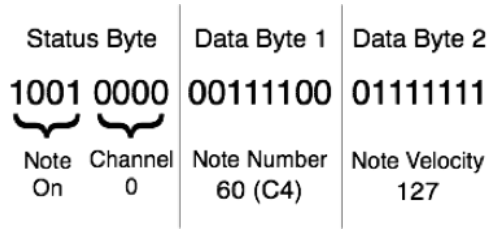

Figure 2: MIDI "Note On" message example.

2.2 MIDI Ports

The MIDI specification describes three logical port types: MIDI In, MIDI Out, and MIDI Thru. The original MIDI 1.0 spec defines the MIDI data stream as a "unidirectional asynchronous bit stream at 31.25 Kbits/sec. with 10 bits transmitted per byte (a start bit, 8 data bits, and one stop bit) [1]." The data stream is organized as follows: MIDI OUT -> MIDI THRU (optional for daisy chaining devices) -> MIDI IN. Although in the original MIDI 1.0 spec, the ports were physical 5 pin DIN ports on MIDI controllers and modules, these are now also logical software ports, with physical connections via MIDI Cable, USB, Firewire, RTP, or Bluetooth. MIDI support is built into macOS, Windows, and Linux with the OS specific APIs Core MIDI, ALSA and JACK, and Multimedia Library, respectively. An example of the physical MIDI In and MIDI Out ports can be seen in Figure 1. Software ports can be manually configured by the user, but may be automatically configured via software. On macOS 10.13, for example, software MIDI ports can be configured within "audio MIDI setup".

2.3 MIDI Transports

Five official MIDI transport specifications have been introduced since the inception of MIDI. The "MIDI 1.0" specification from 1982/83 was the first and most comprehensive specification, including not only message formats, but also specifications for a 5-pin DIN MIDI cable. A USB and Firewire spec were introduced in 1999, although the USB spec has been pulled from the MIDI Manufacturers Association (MMA) website. Two network MID transport specs, RTP-MIDI and Bluetooth LE MIDI provide the main inspiration for NDN-MIDI.

RTP-MIDI. RTP-MIDI uses Real-time Protocol to send MIDI messages within packets over Ethernet and Wi-Fi [3]. In addition to the specifications from MIDI 1.0, RTP-MIDI adds session management, device synchronization, and detection of lost packets. Although the specification has inexplicably been pulled from the MMA website, RFC 6295, titled "RTP Payload Format for MIDI", is available [4]. The only "official" version, of RTP-MIDI, called "AppleMIDI" is implemented by Apple and made specifically for Apple devices. It creates virtual MIDI ports to create sessions between devices using IP addresses and UDP ports or Bonjour. Although AppleMIDI is supported in iOS, iOS devices cannot initiate sessions. Although unofficial implementations have been created to connect Windows and Linux devices to Apple devices, no official cross-platform implementation exists.

Bluetooth LE MIDI. The MMA working group evaluated the possibilities of using Bluetooth LE for MIDI transport, and in 2015, an official spec was created. The spec is only 7 pages long and is based on Apple's implementation that was first introduced in iOS 8 and OSX 10.10. Potential performance issues for real time performance when using Bluetooth LE MIDI are addressed in the spec: "...delays in wireless packet delivery may occur unexpectedly at any time, temporarily increasing latency and jitter" [6]. In this case, jitter is the deviation from the expected temporal spacing between MIDI messages. Limitation of Bluetooth LE itself can also lead to performance issues. Bluetooth connection range is limited, although the introduction of Bluetooth 5.0 will extend the potential range. There is also a limitation of 7 connections to one device, which means only 7 MIDI controllers could connect to 1 playback module.

3 DESIGN CHOICES AND IMPLEMENTATION

3.1 NDN-MIDI Application Overview

The current implementation of NDN-MIDI consists of two macOS command-line applications running on separate machines: Controller and Playback Module. At a high level, the Controller is used to take MIDI input (primarily Note On/Off Messages) from a user-specified MIDI port and transmit it over the network using NDN NFD [8]. All available MIDI ports for the computer on which the Controller is running will appear as options in the Controller UI when the application is started. These ports may correspond to devices physically connected to the computer, such as with USB, or connected via other software. The Playback Module runs on a different computer. It uses an existing MIDI port or creates a MIDI port named "NDN" and accepts MIDI data over the network through NDN NFD. This MIDI port can then be selected as the input in any macOS application that uses MIDI ports. Therefore, the Controller and Playback Module can be used as a network MIDI transport for any macOS supported MIDI controllers and playback modules. For the Playback Module, NDN-MIDI was tested with the applications SimpleSynth, Apple's MainStage 3, and Finale music composition software. For the controller, it was tested with an AKAI Professional Mini II physical MIDI piano keyboard and the software synthesizer Virtual MIDI Piano Keyboard (VMPK).

3.2 Setup and Configuration

In the following NDN-MIDI configuration example, we will use two macOS machines. One machine will be running the NDN-MIDI Controller application, and will have a USB connected MIDI piano keyboard connected to it. The other machine will be running the NDN-MIDI Playback Module application, as well as SimpleSynth, a simple MIDI software synthesizer playback module. The NDN-MIDI Playback Module will not produce any sounds itself, but will pass the received MIDI messages sent through the network to the SimpleSynth application. The basic flow of MIDI messages can be seen in Figure 3.

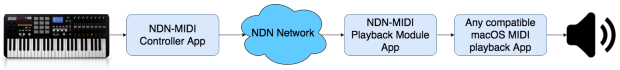

Figure 3: Simple NDN-MIDI configuration

Upon launching the NDN-MIDI Playback Module, the user will encounter the prompts in Figure 4 to setup the initial configuration of the performance session.

Arguments. The first command-line argument given by the user when starting the Playback Module is the name the user wishes to give the playback module. This name will also need to be used by any instances of the Controller application that wishes to connect to this Playback Module. The second command-line argument is optional, and it is the name the user wishes to give to the performance session. If used, this performance session or "project name" will need to be used by any connected Controller application instances. If a project name is not given, a default name will be used.

Allowed Devices Prompt. Device names entered here will be the only devices allowed to connect to the Playback Module. This is optional.

Prohibited Devices Prompt. Device names entered here will be explicitly prohibited from connecting to the Playback Module, while all other devices will be allowed to connect. This is optional.

MIDI Ports Prompt. If desired, the user may choose to set up a virtual midi port with the name "NDN" for the performance session. Otherwise, a list of available ports will be given for the user to select from. The user will then have to select the chosen port as "source" or "MIDI In" within their desired playback software. When using the MIDI playback software "SimpleSynth", for example, if the user wishes to use port "IAC Driver Bus 1", then they would select "IAC Driver Bus 1" as the port within the NDN-MIDI Playback Module, and then select "IAC Driver Bus 1" as the "MIDI Source" within SimpleSynth (see Figure 5). Once this port is configured, any MIDI messages received by the NDN-MIDI Playback Module will be sent through the port "IAC Driver Bus 1" to SimpleSynth, and SimpleSynth will play back the received messages using the instrument sound selected under "instrument" (see Figure 5). Most software playback devices, like SimpleSynth, allow port selection in a similar manner.

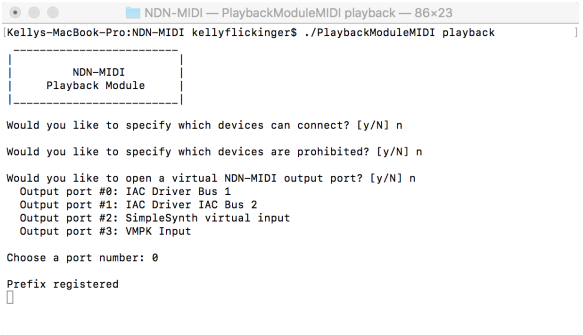

Figure 4: Initial NDN-MIDI Playback Module Configuration

. Upon launching the NDN-MIDI Controller application, the user will encounter the prompts in Figure 6 to setup the initial configuration of the performance session.

Arguments. The first command-line argument given by the user when starting the Controller Module is the name of the Playback module the user wishes to connect to. The third command-line argument is optional, and is the name the user wishes to give to the performance session used by the Playback Module. If the user of the Playback Module did not specify a project name, then the user of the Controller application must also not specify the project name in order to use the default project name.

MIDI Ports Prompt. Port selection for the NDN-MIDI Controller app is similar to port selection in the NDN-MIDI Playback Module app. The user will be given a list of ports to select from. In Figure, 6, "MPKmini2" is the name of the port that was automatically created when the user connected their Akai MPK mini 2 USB piano MIDI keyboard to their computer. Selecting "MPKmini2" as the

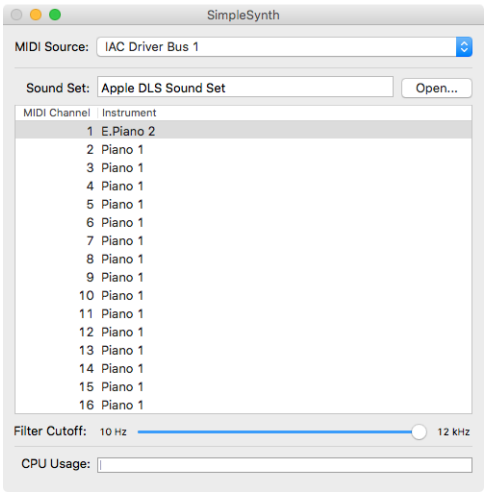

Figure 5: SimpleSynth MIDI Playback Module

port within the Controller application will cause all notes played on the Akai MPK mini 2 to be sent to the Controller application. After selecting a port, the Controller application will attempt to connect to the Playback Module over the network. Users will be notified if the connection is accepted or denied. The details of the NDN namespace for NDN-MIDI will be discussed later.

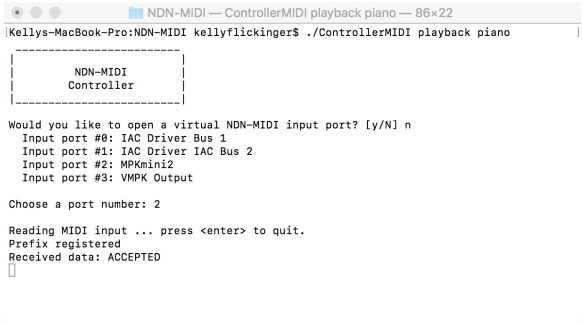

Figure 6: Initial NDN-MIDI Controller Configuration

4 PLAYBACK

Once a connection is established, the user of the Controller may begin playing notes on their connected device to send MIDI messages through the NDN network. The Playback Module will receive the MIDI messages and send them to any configured playback devices, such as SimpleSynth. Figure 7 shows the stream of MIDI messages sent within the Controller application. Each line represents one NDN data packet, which may contain one or more MIDI messages separated by brackets. MIDI messages sent within the same data packet generally represent notes that are played at the same time, although they could also be notes played at very close time intervals. The three numbers within each bracket represents the note type, note number, and note velocity, respectively. The MIDI message in Figure 2, for example, would be represented by "[9 60 127]". The channel value is not used here for reasons discussed later, as it is set by the Playback Module.

The stream of received notes for the Playback Module can be seen in Figure 8, and is mostly similar to the output of the Controller, with some additions. Each received MIDI Message also includes a channel number. This is assigned by the Playback Module automatically at the time of connection in order to support multiple Controllers connected to a single Playback Module. The MIDI protocol limits the number of channels to sixteen and thus NDN-MIDI theoretically supports up to sixteen Controllers connected to a single Playback Module. This has not been tested due to the need of seventeen macOS computers: sixteen running the Controller application and one running the Playback Module application.

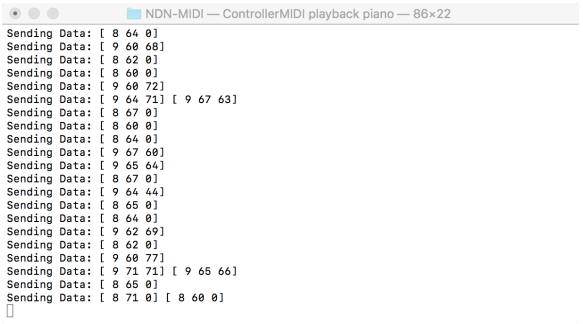

Figure 7: Notes sent from Controller

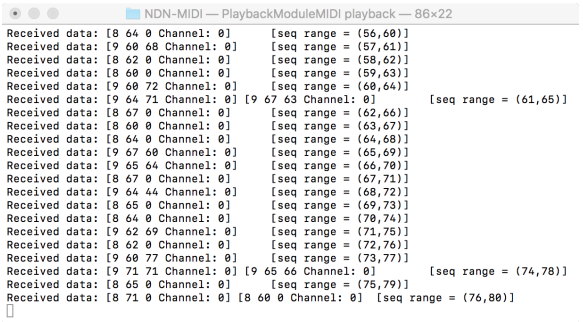

Figure 8: Notes received from Controller at Playback Module

4.1 Groups

One of the main goals of NDN-MIDI was to provide an easy, wireless network MIDI solution for group performance. Therefore by default, the Playback Module will automatically allow connections from any Controller attempting to connect, until it reaches the maximum number of sixteen connections. The Controllers will need to know the correct Playback Module name and project name in order to connect. For additional security, the Playback Module may explicitly allow or prohibit certain devices from connecting.

When setting up a connection to a Controller, the Playback Module will assign the controller one of sixteen available channels. The Playback Module will then mask the first byte of each MIDI message

from that controller with the assigned channel before outputting the message to the playback software. The user of the Playback Module can then assign each channel to a particular instrument playback sound in their playback software of choice. MainStage 3, a popular live performance MIDI software application for macOS, allows such routing of channels to particular instrument sounds. In Figure 9, there are MIDI messages coming from two channels on "Bus 1 IAC Driver" through NDN-MIDI. Channel 1 is assigned to an "Electric Piano" sound and Channel 2 is assigned to a "Synth" sound. Therefore the MIDI messages for Channel 1 are coming from one user using the NDN-MIDI Controller application, and the MIDI messages for Channel 2 are coming from another user doing the same thing. As described before, this could be expanded to up to sixteen channels per MIDI port. Mainstage 3 contains a seemingly endless number of instrument sounds and configurations, as well as an intuitive user interface, allowing for easy and powerful integration with NDN-MIDI.

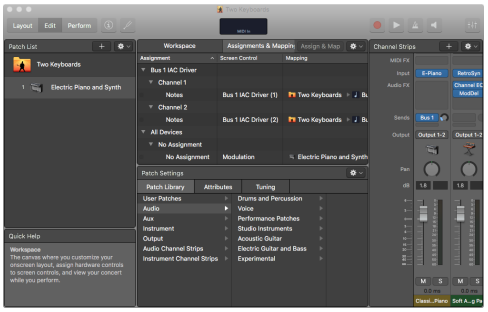

Figure 9: MainStage 3 MIDI Playback Module

4.2 The Main Menu

Any time after the initial configuration of the Playback Module, the user may type "menu" in the command line to view the NDN-MIDI Main Menu in Figure 10. While viewing the Main Menu, communication between Controllers and the playback of notes will continue, although "Received Data" messages will not be displayed. The Main Menu provides the following options:

- View Connections: Current connections to external Controllers are shown here.
- Clear Connections: Connections to all external Controllers will be ended and shutdown messages will be sent to the connected Controllers.
- Allowed Devices: List of the explicitly allowed connections declared when first running the application.
- Prohibited Devices: List of the explicitly prohibited device connections declared when first running the application.
- Toggle Verbose Mode: Turns on/off verbose output to the command line. The default is off. Additional output in verbose mode include connection status, heartbeat messages, timeout messages, out-of-order packet messages, and nack messages.

The user may exit the menu at any time to return to the normal stream of MIDI messages.

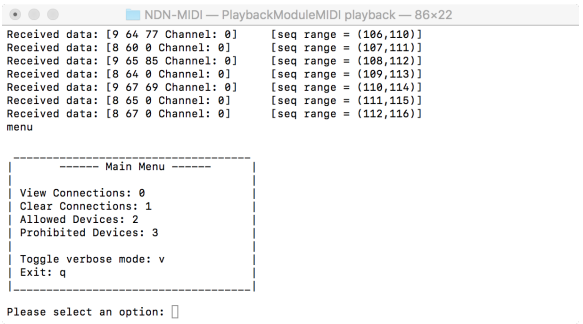

Figure 10: Playback Module Main Menu

4.3 NDN Namespace Design

The requirements for MIDI live performance have significantly impacted our design for NDN-MIDI. To understand how, let's first have an overview of NDN-MIDI's namespace design. The name prefix of NDN-MIDI is as follows:

/<topo-prefix>/<dev-name>/ndn-midi/<proj-name>

- <topo-prefix> : We assume a routable, topological prefix, as NDN assumes routing to be achieved by hierarchical naming. For example, this component could be the address of a subnetwork.
- <dev-name> : Name of the device, which can be either the Controller or Playback Module. This component should uniquely identify each device under the same topological prefix (and preferably be globally unique). For example, it could be a MAC address, or some human-readable name such as "Kellys-YAMAHA-Keyboard".
- ndn-midi : Name of the application.
- <proj-name> : Project name, used as an identifier for the performance session. This component enables the users to have multiple performance sessions within the same topological prefix.

Currently, our design has two sub-namespaces:

- /<seq-no> : Sequence number used to identify each packet, and inform both the Controller and Playback Module about the order each packet is generated. In the current implementation, the sequence number is primarily used as a means of sending multiple, non-collapsing interests at a time in an effort to reduce latency and jitter. Out-of-order packets are simply dropped. In the future, the sequence number could be used to ensure reliable delivery if desired.
- /<controller-name>/heartbeat : Heartbeat messages are sent as interest packets by the Controller, and replied to as data packets by the Playback Module. Controllers use this to detect if the specified Playback Module is up in the network, and Playback Modules use this to learn which controllers would like to use them for playback. Upon receiving a heartbeat interest packet, the Playback Module gets the Controller's name and looks up if it is allowed and if so, if it is already connected. If the Controller is not allowed to connect, the Playback Module will send a message to the

Controller indicating that the connection is denied. If the device is allowed, but it is not currently connected, the Playback Module allocates a block to maintain information related to that Controller. Heartbeat messages may in some way sound like the idea of a "connection" in IP, but our true intention in doing so is that it increases the Playback Module's efficiency by eliminating the sending of interest packets for nonexistent Controllers, and enables the Playback Module and Controller to detect if either is down.

4.4 Addressing Real-time Requirements of Live Performance

Packet Lifetime. Because of the nature of music performance, all data packets should have a short freshness period, and interest packets should have a short lifetime and must fetch fresh data, including the heartbeat packets. In our current implementation, the freshness period of data packets and the lifetime of interest packets are both set to be 1 second. This applies to heartbeat packets as well.

Latency and Jitter. Due to the real-time transport requirements of live performance, latency and jitter are the most prioritized performance concerns in NDN-MIDI. Perceivable latency could make performance difficult to impossible for a musician, while perceivable jitter could result in music that is not rhythmically correct. A model using a sequence of single interest/data at a time between the Controller and Playback Module proves to be problematic. When performing a piece of music, two notes can be temporally, arbitrarily close to each other. If only one interest packet is sent from the Playback Module to fetch one note at a time, each note will at best be separated by a single round trip time, potentially introducing perceptible latency to the performer. Additionally, if the round trip time is greater than the spacing between two performed notes, the playback will be rhythmically incorrect.

These issues are somewhat addressed in the original MIDI spec, which describes "running status" [1], in which consecutive MIDI messages of the same type can share a single status byte, reducing the size of individual MIDI message from 3 bytes to 2 bytes. The MIDI spec, however, was written before MIDI messages were being sent as packets in a network, where data loss was not considered. Lost packets when using running status could wreak havoc on real-time musical performance, and therefore "running status" is not practical to satisfy the needs to NDN-MIDI. /par In order to more fully solve the latency issues of a sequence of single round trips for every data packet, it would make sense to send multiple interests at a time so that data packets that are ready to be sent could be transmitted immediately. Unfortunately, in NDN, multiple interests with the same name cannot be sent simultaneously from the Playback Module to the Controller because they would be collapsed into one interest in the pending interest table. Therefore, we used sequence numbers to differentiate the interests. When a Playback Module first receives a heartbeat message from a Controller, the playback module will preemptively "prewarm" the network with multiple interest packets with sequential sequence numbers. In the current implementation, the prewarm amount is 10, so the playback module will immediately send interest packets with sequence numbers from 0 to 9, even before receiving one data packet. As long as the prewarm amount is high enough, the Controller can send data packets whenever needed because there will be pending interest packets in the network from the Controller. For example, if a performer starts off a performance by playing a fast burst of notes, the "prewarm" interests can be satisfied immediately. In practice, the prewarm amount of 10 interest packets resulted in no playback delay, whereas smaller prewarm amounts resulted in stuttered playback.

Although prewarming the network lessens the impact of latency in rapidly played successive single notes, it cannot address the issue of multiple notes being played simultaneously, such as when a pianist plays a chord consisting of multiple notes. Initial implementations of NDN-MIDI suffered from audibly stuttered playback when playing chords. When a performer played a chord (multiple notes played simultaneously) on a MIDI piano keyboard controller, instead of all of the notes of the chord being played back at the same time, they were played sequentially. To address this issue, the current version of NDN-MIDI places simultaneously created MIDI messages into a single data packet. As MIDI messages are created, they are placed into an input queue (usually for a very short amount of time). When the Controller receives interest packets, they are placed into an interest queue. If both the input queue and interest queue are not empty, the entire contents of the input queue is immediately sent in a single data packet to satisfy a single interest packet. This implementation has successfully eliminated any perceptible delay in notes that are played at the same time.

Of course, the consequence of putting multiple MIDI messages into one packet is if the data packet is lost in network, all note messages in that packet are lost. Fortunately, this does not create an issue from a musical standpoint. Since all of the notes in a single message are intended to be played back at the same moment in time, the result of such packet loss is simply a dropped chord instead of a single note. It could actually be argued that it is better to drop the entire chord than to drop only some of the notes.

Efficiency. While the current "connection-based" implementation of NDN-MIDI may sound more suited for IP than NDN, a bigger concern is efficiency. Although NDN-MIDI could adopt a simple producer/consumer model, where the Playback Module is the consumer, and Controllers are producers, such a model would not be as efficient. Since the playback module has to prewarm the network with multiple interest packets to solve the latency issue, and it may not know how many Controllers there are in the network, or who they are, the Playback Module would have to periodically send a great amount of interest packets to check, which would be inefficient.

With the Playback Module and Controller both being producer and consumer at the same time (sending both data and interest packets), the Playback Module can wait and listen once it is up in the network, and only prewarm the network when necessary, i.e. when a Controller sends it a heartbeat interest packet to request for use of the Playback Module. This also always greater flexibility in accepting or denying certain Controllers.

Packet Ordering. In NDN-MIDI, interest and data packets are processed as soon as they are received, and do not wait for out-of-order packets. If out-of-order packets do arrive later, they are dropped. Therefore, if packet loss or reordering happens, lost or reordered

packets are simply skipped by the Playback Module. This is an effort to reduce latency as much as possible in real-time performance, but also has the potential to create some problems. Remember that the Playback Module will only know to stop playing a note if it receives a Note Off Message. If a Note On Message is received, but the corresponding Note Off Message is lost or dropped, the Playback Module could potentially play forever. Initially NDN-MIDI attempted to solve this problem simply by sending two consecutive Note Off Messages for every Note On Message sent. This approach worked because at that time each data packet only contained a single MIDI message. Once the implementation was changed to contain multiple MIDI messages, this solution was no longer viable since both Note Off Messages would be in the same packet and thus both be lost if the packet was dropped. Additionally, a solution in which a second Note Off Message is sent in the next packet is not a viable solution since the first packet might contain additional Note On Messages for the same note. In practice, this has not been a significant issue. In the rare case that a note does get stuck due to a lost or dropped Note Off Message, playing the same note again will usually solve the issue since a Note Off Message will stop the note from being played regardless of how many Note On Messages were received.

Crashing and Recovery. Heartbeat packets enable the Controller and Playback Module to tell each other whether they are still running in the network. If either has not heard from the other for a predetermined length of time, it will assume the other has crashed. Either the Controller will reset the sequence number, or the Playback Module will deallocate the block for the Controller. Heartbeat messages are sent periodically and it does not matter whether the Controller or Playback Module is running in the network first. Once either one has detected the presence of the other, the communication can begin.

4.5 Security

Our application development has not focused extensively on security (though it is not completely devoid of it, as you will see below). For our future plans on improving security in NDN-MIDI, please refer to the "security" section in "Future Work".

One security issue that has been a focus is the mitigation of DoS attack against the Playback Module. Normally, it is harder to DoS attack a particular device in NDN than it is in IP if the data name focuses on content [2]. Unfortunately, because our application requires the Playback Module to put its name in the name prefix in order provide service to controllers for music performance, the possibility of a DoS attack against the Playback Module is increased.

As stated earlier, the user of the Playback Module can explicitly allow or prohibit devices with specific names. Although this provides some level of safety against attack, these names can still be spoofed. An attacker can rapidly send heartbeat interest packets with randomly-generated bogus Controller names or guess currently allowed names to a Playback Module to request for a connection. Since a Playback Module keeps a table for all Controllers interested in using it for music performance, this table can be quickly filled up. However, to keep using a Playback Module, the Controller is required to periodically send the same heartbeat interest packet (with a different nonce) to the Playback Module to prove that it is still up in the network, otherwise if the Playback Module hasn't heard from a Controller for a long enough period, it will deallocate the block associated with it. Since the Playback Module scans all allocated blocks to check how long it hasn't heard from the Controller, it can essentially deallocate at the same speed the attacker is sending bogus heartbeat interest packets after that "long enough period" has passed. If an attacker cannot manage to fill up a Playback Module's table within this "long enough period", the DoS attack is not effective. Meanwhile, a legitimate Controller can still make a "connection" with the Playback Module and just sit there, not producing any data, and its allocated block in the Playback Module will not be lost because the Playback Module is receiving its periodic heartbeat. Additionally, the user of the Playback Module can always clear all of its connections if anything goes awry.

4.6 An Illustration of Communication Between Playback Module and Controller

An example of packet exchange in an NDN-MIDI connection can be seen in Figure 11. A "'guitar" Controller initiates a connection with a Playback Module by sending a heartbeat interest. In this case "guitar" is an allowed device and the Playback Module responds to accept the connection and sends 10 "prewarm" interests to the guitar Controller. The guitar then sends MIDI messages back to the Playback Module in data packets. Meanwhile, another device, a piano, makes a connection to the Playback Module and begins to send data. Although in this this example, the two Controllers do not send MIDI messages at the same time, they could.

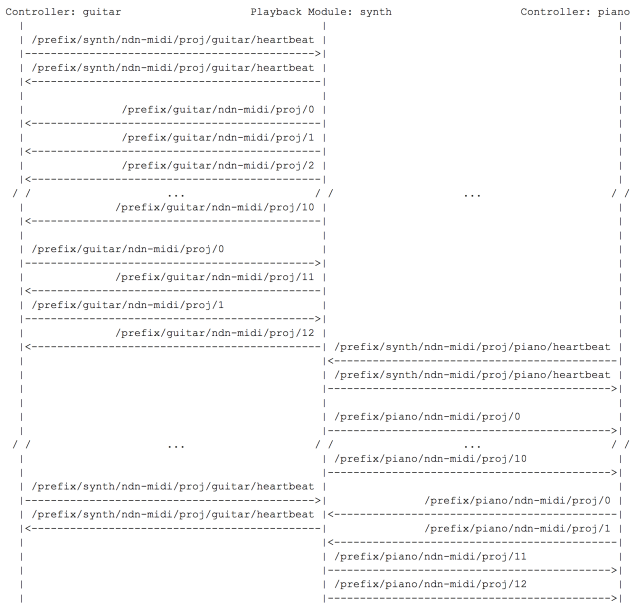

Figure 11

4.7 MIDI APIs

Although native MIDI APIs exist for macOS, Windows, and Linux, it was preferable to use a common API that could be used across all of the platforms. Fortunately, RtMidi is an excellent set of C++ classes that "provides a common API for real-time MIDI input/output across Linux (ALSA and JACK), Mactintosh OS X (CoreMIDI and JACK), and Windows (Multimedia Library)" [7]. All of the MIDI programming in NDN-MIDI utilities RtMidi, which will make it easier to extend the application to other platforms in the future. The code can be compiled for a specific OS and API with little to no changes to the code itself. RtMidi separates MIDI input and output into the classes RtMidiIn and RtMidiOut respectively, enabling creation of virtual MIDI ports and sending and receiving of MIDI messages. The Controller utilizes the RtMidiIn class and the Playback Module utilizes the RtMidiOut class.

5 FUTURE WORK

5.1 Security

It is the belief of the author that in the initial stages of development, attacks or hijacked performance sessions will result in little more than failed MIDI connections or annoying sounds coming from a playback module if unwanted MIDI data was introduced into the performance. Further work must be done to first decide on the level of security necessary for specific performance situations, and then how to implement it.

5.2 Wi-Fi Direct

A mobile musician can become even more mobile if they can use Wi-Fi Direct to directly connect their MIDI devices. NDN-MIDI was initially conceived with Wi-Fi direct in mind as the primary network connection. Due to the one-hop nature of these networks, it is hypothesized that such connections are likely to decrease latency and jitter of MIDI messages. The next implementations of NDN-MIDI will take advantage of newly completed implementations of Wi-Fi direct for NDN and/or NDN Control Center on macOS.

5.3 Timestamp

When extending NDN-MIDI for use cases when real-time playback is not needed (such as playing back a MIDI file) it may be desirable to add reliable in-order data delivery to ensure all notes are played back in order. Ensuring in-order delivery, however, will cause the playback of out-of-order notes to be delayed, guaranteeing rhythmic inaccuracy. Therefore, a combination of buffering and time stamping would be needed.

5.4 Name Uniqueness

NDN-MIDI has three name components that are variable: topological prefix, device name, and project name. While the topological prefix is determined for us, the uniqueness of device and project name is important. If there is a data name collision, then a device may receive data that is not meant for it. In the current implementation, the device name and project name is decided by the user, which could be problematic depending on the performance use case. It is the author's intention to take some unique name for the device automatically in the future, while leaving the uniqueness of project name up to the user. Possibilities include using MAC addresses, host names (if the device have one), some sort of barcode scanning method similar to the case in [5]., or public/private keys.

5.5 Beyond Note On/Off Messages

The current implementation of NDN-MIDI focuses primarily on transmitting MIDI On/Off Messages only. While some other MIDI message types do work, not all messages are guaranteed to be supported. If a full MIDI implementation is desired, changes to NDN-MIDI will need to be made to support all necessary message types. If only a lightweight live performance implementation is desired, however, the the current implementation will be sufficient.

5.6 Beyond macOS

Although the Playback Module is required to do some heavy lifting by managing multiple connections and running playback software, the requirements of a machine running the Controller are much less. Any machine with basic MIDI support that has a USB connection and can run NFD would be sufficient. A Raspberry Pi running Linux specially configured to run NDN-MIDI and NFD with little user interaction could provide an easy and inexpensive way to provide network connected MIDI devices for live performance. Given the cross-platform capabilities of the RtMidi library used, little modification would be needed to create a Linux version of NDN-MIDI for such an application.

ACKNOWLEDGMENTS

This work was made possible with the effort of Wenrui Wu and the advisement of Lixia Zhang, Alex Afanasyev, and Gordon Henderson.

REFERENCES

[1] "The Complete MIDI 1.0 Specification'. https://www.midi.org/specifications/item/the-midi-1-0-specification
[2] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang, "Named Data Networking," ACM SIGCOMM Computer Communication Review (CCR), July 2014.
[3] "RTP-MIDI," https://en.wikipedia.org/w/index.php?title=RTP-MIDI&oldid=768373999.
[4] J. Lazzaro, J. Wawrzynek, "RTP Payload Format for MIDI", RFC 6295, DOI 10.17487/RFC6295, June 2011, <http://www.rfc-editor.org/info/rfc6295>.
[5] J. Burke, P. Gasti, N. Nathan, G. Tsudik, "Securing Instrumented Environments over Content-Centric Networking: the Case of Lighting Control," Proceedings of IEEE INFOCOMM 2013 NOMEN Workshop, April 2013.
[6] "Bluetooth LE MIDI Specification," https://www.midi.org/specifications/item/bluetooth-le-midi.
[7] "The RtMidi Tutorial," https://www.music.mcgill.ca/ gary/rtmidi/index.html.
[8] "NFD - Named Data Networking Forwarding Daemon," http://named-data.net/doc/NFD/current/.