Audio Engineering Society

Convention Paper

Presented at the 117th Convention
2004 October 28–31 San Francisco, CA, USA

An RTP Payload Format for MIDI

John Lazzaro[1] and John Wawrzynek[1]

[1] *Computer Science Division, UC Berkeley, Berkeley, CA, 94720-1776, USA*

Correspondence should be addressed to John Lazzaro (lazzaro@cs.berkeley.edu)

ABSTRACT

The Real-Time Protocol (RTP) is an extensible transport for sending media streams over Internet Protocol packet networks. We describe a new payload format that extends RTP to transport MIDI (the Musical Instrument Digital Interface command language). The payload format encodes all commands that may legally appear on a MIDI 1.0 DIN cable. The format is suitable for interactive applications (such as the remote operation of musical instruments) and content-delivery applications (such as file streaming). The format defines tools for graceful recovery from packet loss, to support use over lossy unicast and multicast networks (including wireless networks). Stream behavior, including the MIDI rendering method, may be specified during session setup. Rendering methods are specified using the extensible Multipurpose Internet Mail Extensions (MIME) registry.

1. INTRODUCTION

The Real-Time Protocol (RTP, [1]) is widely used to transport media streams over Internet Protocol packet networks. RTP is suitable for both low-latency interactive applications (such as Internet telephony) and content-delivery applications (such as Internet radio). RTP itself does not provide service quality guarantees, but applications may use lower-level protocols to configure the quality of the network that RTP uses.

RTP is an extensible transport. Support for a new audio or video codec may be added to RTP by creating a new payload format. The Audio/Video Transport (AVT) working group of the Internet Engineering Task Force (IETF) coordinates payload format development.

At the 52nd IETF meeting (December 2001), we presented a proposal for an RTP payload format for MIDI (Musical Instrument Digital Interface, [2]). MIDI is a standard for coding the physical gestures that underly musical performance (playing piano keys, striking drum pads, pushing faders, etc). Our proposal was based on the network protocols used in the network musical performance system described in [3].

Shortly after the 52nd IETF meeting, RTP MIDI was accepted as an AVT standards-track working

group item. Many industrial and academic experts have participated in the working group process, with the goal of defining a format suitable for use in a wide range of present and future MIDI applications. The MIDI Manufacturers Association has participated in this effort, as have members the Motion Pictures Expert Group (MPEG), whose synthetic audio codecs use MIDI.

As of this writing, the RTP MIDI documents [4] [5] have reached stability, and an example protocol implementation (based on [6]) exists. The working group is preparing a "Last Call" request for objections to the documents from the IETF community. If the Last Call proceeds without objections, the documents would be handed off to other IETF committees for further vetting. If these committees approve, the MIDI payload format would become an IETF Proposed Standard.

In this 117th AES conference paper, we present an overview of the RTP MIDI payload format. RTP MIDI is a significant project to the AES community, because if MIDI performances can be sent as RTP streams, new types of network applications may emerge:

- Manufacturers of music and audio equipment may adopt RTP as the transport layer for local media networks. In a local media network, Internet Protocol (Layer 3) RTP streams for audio, video, and MIDI are mapped to a wired or wireless local area network (Layer 2). The Layer-2 network is chosen to match the application requirements (Ethernet, FireWire, and Wi-Fi are examples of Layer-2 networks). Local area networks have been used for performance control for over a decade, both in academia [7] [8] [9] and industry (for example, [10]).

- Conferencing applications may add support for network musical performance. In a network performance, musicians located in different physical locations interact over a network to perform as they would if located in the same room. Interest in network performances dates back to the early days of Internet media protocols [11] [12] [13], and the topic has recently seen renewed activity [14] [15] [16] [3].

- The Internet content-streaming community may begin to use MIDI for low-bitrate music coding, perhaps in conjunction with normative sound synthesis methods [17].

Technical highlights of the RTP MIDI payload format are listed below:

- The payload format uses a novel resiliency scheme to support MIDI transport over networks that occasionally lose packets. The resiliency scheme is feed-forward in nature, and does not use packet retransmission.

- The payload format defines flexible tools for encoding MIDI command timing. The tools support the accurate RTP transcoding of MIDI 1.0 DIN cables and Standard MIDI Files [2].

- Multiple synchronized RTP MIDI streams may be sent in a media session. The session may also include synchronized video and audio streams.

- Sessions may specify how a receiver renders a MIDI stream into audio. Alternatively, sessions may specify how an operating system presents a MIDI stream to application programs. These features are extensible to new renderers and operating systems, via the Multipurpose Internet Mail Extensions (MIME) registry [18] of the Internet Assigned Numbers Authority (IANA).

The organization of this conference paper proceeds as follows. We begin with an introduction to the RTP protocol (Section 2). Section 3 describes the design challenges of the RTP MIDI payload format design. Sections 4-6 describe, at a high level, the bitfield structure of the RTP MIDI payload format. Sections 7-8 describe example algorithms for sending and receiving RTP MIDI streams.

A key part of an RTP payload design is support for the popular IETF session management protocols: the Session Initiation Protocol (SIP, [19]) for interactive applications such as telephony, and the Real Time Streaming Protocol (RTSP, [20]) for content-streaming applications. In Section 9, we introduce SIP as an example of a session management protocol. In Section 10, we introduce the Session Description Protocol (SDP, [21]) and outline the session description features of RTP MIDI.

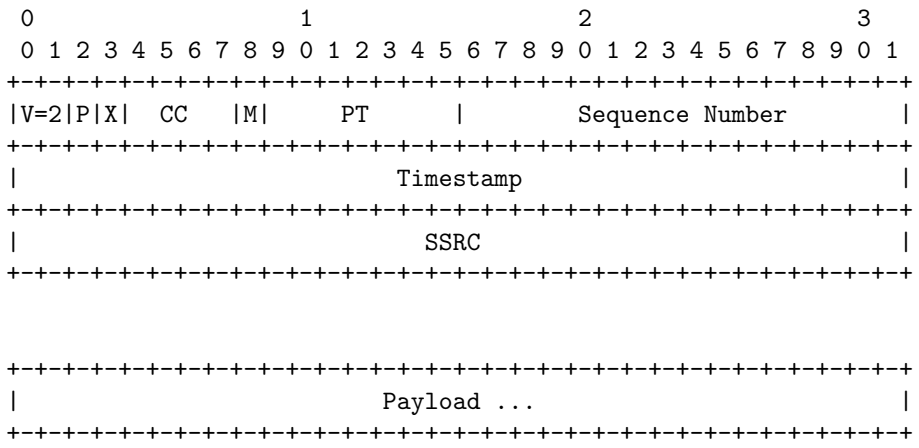

Fig. 1: RTP packet format (space delineates the header and payload)

Note that the purpose of this AES conference paper is to provide an overview of RTP MIDI. Although we consider the RTP MIDI design to be stable, and the details in this paper reflect this stable design, the protocol is still an IETF work in progress, whose details may change at any time. Implementors should use [4] as a reference for preliminary software development, not the details contained in this paper.

2. THE REAL TIME PROTOCOL

This section provides a brief introduction to the Real Time Protocol (RTP, [1], [22]). We recommend [24] for a comprehensive introduction to RTP.

An RTP media stream is a sequence of logical packets that share a common format. Each packet consists of two parts: the RTP header and a payload. Figure 1 shows the structure of an RTP packet.

We describe RTP packets as "logical" packets to highlight the fact that RTP itself is not a network-layer protocol. Instead, RTP packets are mapped onto network protocols by an application.

RTP was designed to be used with network protocols whose abstraction exposes the underlying nature of the network, such as the User Datagram Protocol (UDP, [23]). UDP lets a host send a single packet to another host (unicast UDP) or to multiple hosts that are listening to the same address (multicast UDP).

UDP packet delivery is best-effort. The network strives to deliver packets in-order at the underlying latency of the network. However, packets may be lost in route to some or all receivers, packets may arrive out of order, and packets may be subject to variable delays.

When RTP is mapped to UDP, a single RTP packet is mapped to a single UDP packet. Thus, an application that uses RTP is directly exposed to the best-effort nature of the network: a lost UDP packet results in a lost RTP packet, a delayed UDP packet results in a delayed RTP packet.

This exposure may seem to be an RTP shortcoming. In fact, it is the prime feature of RTP. By presenting an unfiltered view of network behavior, RTP lets a media application handle lost and late packets in a graceful way [25].

For example, if a video RTP packet codes a small patch of the image, and this packet is lost, the application may be able to use nearby image data (in time and space) to "fill in the hole in the image" in an unobjectionable way.

Whereas, if the application used a transport that provided reliable in-order delivery, a packet loss might result in a long delay in data delivery, due to the time it takes to retransmit the lost packet. The long delay might force the application to interrupt video playback. Note that this rationale for RTP also applies to lossless strategies for image repair, such as the use of forward error correction (FEC) codes in the RTP payload [26].

With this background, we can see why RTP is aptly named, even though it is not a protocol that offers real-time service quality guarantees: RTP offers a set of tools real-time applications may use to cope with the best-effort nature of the underlying network.

The RTP header fields encode several of these real-time tools (Figure 1). For example, the header has a 16-bit sequence number. The sequence number is incremented by one (modulo 2^{16}) for each packet sent in the stream. By noting breaks in the sequence number series, receivers may detect lost packets.

The header also has a 32-bit timestamp value. The timestamp value indicates the earliest moment in time that is coded by the packet. Timestamps are interpreted modulo 2^{32}, and have units of Hz. A timestamp scaling factor (the *clock rate*) is defined during session setup. For example, if the clock rate is 44,100 Hz, timestamps that differ by 2 seconds have values that differ by 88,200.

RTP has separate time and sequence numbers to maximize payload format design flexibility. For example, in telephony, packet transmission may be stopped if a user stops talking. The packet that ends the silent period would have a discontinuous timestamp jump, but the unbroken sequence number series would indicate that no packets were lost.

The header defines a marker bit (M), that payload formats may use to indicate special packet handling. For example, telephony voice payloads use the M bit to indicate the end of a silent period.

The RTP standard also defines a low-bandwidth back-channel protocol, the RTP Control Protocol (RTCP). Senders and receivers use RTCP to share reception statistics and other types of information. Of particular interest in this paper is the RTCP Extended Highest Sequence Number Received (EHSNR) field. The 32-bit EHSNR codes the highest sequence number that a receiver has observed from a sender. The EHSNR also codes the number of sequence number rollovers seen by the receiver.

Applications that send multiple RTP streams code synchronization information in RTCP packets, by specifying the wall-clock time that corresponds to an RTP timestamp value. In videoconferencing applications, receivers use these RTCP fields for lip-sync.

Because RTP supports multicast UDP (many parties sending and receiving to the same network address), RTCP is carefully designed to scale with the number of session participants. As the number of parties in a session increase, the rate of RTCP transmission by each party decreases, so that total RTCP session bandwidth does not exceed a fixed fraction of the total session bandwidth.

Finally, we note that although RTP was designed for use with unreliable network protocols, it is possible to map RTP packets to reliable byte-stream protocols, such as the Transmission Control Protocol (TCP, [27]).

3. RTP MIDI DESIGN GOALS

In this section, we describe the design goals of the RTP MIDI payload format.

At a minimum, a good payload format partitions media data into packets in a way that minimizes the impact of lost and late packets. In addition, a payload format may provide tools that receivers use to cope with lost and late packets.

What sort of resiliency tools are appropriate for RTP MIDI? Consider a simple payload format that codes one complete MIDI command in each packet. Assume that each packet either arrives with a fixed transmission latency, or is lost during transmission. In the latter case, a lost packet may cause an audio artifact. We classify artifacts into two categories:

- Transient artifacts. Transient artifacts produce immediate but short-term glitches. For example, a lost MIDI NoteOn (`0x9`) command produces a transient artifact: one note fails to play, but the artifact does not extend beyond the end of that note.
- Indefinite artifacts. Indefinite artifacts produce long-lasting errors. For example, a lost MIDI NoteOff (`0x8`) command may produce an indefinite artifact: the note that should have been ended by the lost NoteOff command may sustain indefinitely. As a second example, the loss of a MIDI Control Change (`0xB`) command for controller number 7 (Channel Volume) may produce an indefinite artifact: after the loss, all notes on the channel may play too softly or too loudly.

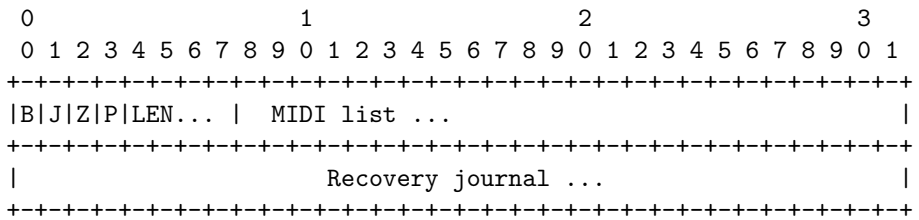

Fig. 2: RTP MIDI payload format ("..." denotes variable-length fields)

Transient artifacts are clearly undesirable. However, indefinite artifacts are intolerable. One design goal for RTP MIDI was the elimination of indefinite artifacts in a performance rendered from a lossy RTP MIDI stream. A second design goal was the minimization of the frequency and severity of transient artifacts. The payload format realizes these goals via a resiliency tool, the *recovery journal system*.

A third design goal was generality. The range of potential RTP MIDI applications is wide, and each application class has its own requirements. Whenever possible, we made design choices that were inclusive to this application pool.

4. PAYLOAD FORMAT OVERVIEW

The RTP MIDI payload format maps a MIDI command stream (16 voice channels + systems commands) onto an RTP stream. To send more than 16 voice channels over RTP, applications send several synchronized RTP MIDI streams.

Figure 2 shows the structure of the payload. The MIDI command section begins with a variable-length header. If the `LEN` header field is non-zero, a `MIDI list` field follows the header. The `MIDI list` codes timestamped MIDI commands, and provides the essential service of the payload format. The `MIDI list` has a size of `LEN` octets (octet is an IETF term for 8-bit bytes).

If the `J` header bit is set to 1, a variable-length `Recovery journal` field appears at the end of the packet. The recovery journal system uses this field to provide resiliency to lost packets. We describe the recovery journal system in Sections 5 and 6 of this paper. In this remainder of this section, we describe the `MIDI list`.

Two `MIDI list` sizes are supported. If the `B` header bit is set to 0, the `LEN` field is 4 bits long, supporting a maximum MIDI list size of 15 octets. This mode is intended for interactive applications, that usually send one MIDI command per packet to minimize encoding latency.

If the `B` header bit is set to 1, the `LEN` field is 12 bits long, supporting a maximum `MIDI list` size of 4095 octets. This mode is intended for content-streaming applications, that encode many MIDI commands per packet to amortize header and journal overhead.

Figure 3 shows the `MIDI list` structure: a paired list of `Delta Time` fields and `MIDI Command` fields. If the Z header bit is set to 1, the `Delta Time 0` field is absent from the list, and `MIDI Command 0` has an implicit delta time value of 0.

Each `MIDI Command` field codes one of the MIDI command types that may legally appear on a MIDI 1.0 DIN cable [2]. The first channel command in the list must begin with a status octet, but subsequent channel commands may use running status [2].

As a rule, each `MIDI Command` field codes a complete command, in the binary command format defined in [2]. However, this rule has several exceptions, to handle MIDI constructs such as System Exclusive commands and System Real-time messages. See [4] for details.

4.1. Timestamps

The base timestamp of an RTP MIDI packet is set by the 32-bit RTP header timestamp field (Figure 1). The `Delta Time` fields in the `MIDI list` encode the timing of individual MIDI commands relative to the base timestamp of the packet, using an algorithm we now describe.

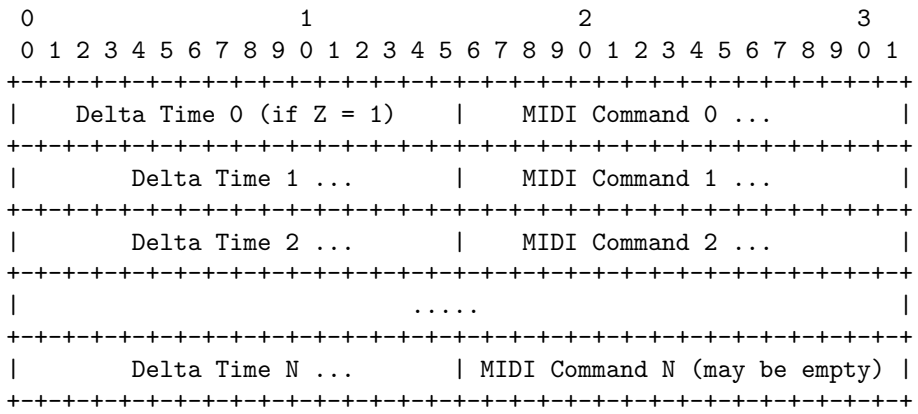

Fig. 3: `MIDI list` structure ("Z" as shown in Figure 2)

The `Delta Time` fields use a modified form of the MIDI File delta time syntax [2]. RTP MIDI delta times use 1-4 octet fields to encode 32-bit unsigned integers. Figure 4 shows the encoded and decoded forms of delta times. The units of decoded delta times match the units of the RTP header timestamp.

The command timestamp for `MIDI Command K` is the summation (modulo 2^{32}) of the RTP timestamp for the packet and decoded delta times `0` through `K`.

By default, a command timestamp indicates the execution time for the command. The difference between two timestamps indicates the time delay between the execution of the commands. This difference may be zero, coding simultaneous execution.

This interpretation of timestamps works well for transcoding a Standard MIDI File (SMF, [2]) into an RTP MIDI stream, as SMFs code a timestamp for each MIDI command stored in the file. Other interpretations of timestamps may work better for transcoding a MIDI source that uses implicit command timing (such as MIDI 1.0 DIN cables) into an RTP MIDI stream. The timestamp interpretation for a stream may be configured during session setup (Section 10).

Note that RTP timestamps have units of Hz, not musical score beats. To stream a MIDI file over RTP, senders use the tempo map of the MIDI file (encoded as Meta-Events) to convert metric units into Hz.

In a MIDI stream, the RTP header timestamps for two sequential packets may be identical, or the second packet may have a timestamp arbitrarily larger than the first packet (modulo 2^{32}). All command timestamps in the `MIDI list` of a packet must be less than or equal to the RTP header timestamp of the next packet in the stream (modulo 2^{32}).

Returning to Figure 3, we note that the last `MIDI Command` field in the `MIDI list` may be empty. Senders may use this feature to precisely set the media time of a packet. The media time is the total amount of time coded by the packet, and is computed by subtracting the last command timestamp in the `MIDI list` from the RTP timestamp (modulo 2^{32}).

The ability to precisely set packet media times may be important for compatibility with audio streaming architectures that expect all packets in a stream to code the same period of media time.

Interactive applications often use a 0 ms packet media time to minimize encoding latency. In this mode of operation, the `MIDI list` of each packet codes a single MIDI command. The payload header's `Z` bit is set to 0, so that the RTP timestamp acts as the timestamp for the MIDI command. Thus, the packet codes a moment in time, and the media time coded by the packet is 0 ms.

5. THE RECOVERY JOURNAL SYSTEM

MIDI is a fragile code. As we discussed in Section 3, a single lost command in a MIDI stream may produce an artifact of indefinite duration in the rendered audio performance.

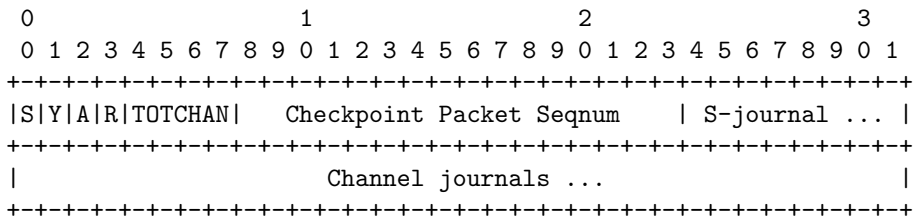

Fig. 5: Top-level recovery journal format.

```
One-Octet Delta Time:

   Encoded form: 0ddddddd
   Decoded form: 00000000 00000000 00000000 0ddddddd

Two-Octet Delta Time:

   Encoded form: 1ccccccc 0ddddddd
   Decoded form: 00000000 00000000 00cccccc cddddddd

Three-Octet Delta Time:

   Encoded form: 1bbbbbbb 1ccccccc 0ddddddd
   Decoded form: 00000000 000bbbbb bbcccccc cddddddd

Four-Octet Delta Time:

   Encoded form: 1aaaaaaa 1bbbbbbb 1ccccccc 0ddddddd
   Decoded form: 0000aaaa aaabbbbb bbcccccc cddddddd
```

Fig. 4: Decoding delta time formats.

In this section, we introduce the recovery journal system. The system ensures that the audio rendered from an RTP MIDI stream does not contain indefinite artifacts, even if the stream is sent over a network that loses packets.

The recovery journal system does not use packet retransmission. Instead, if the system is in use, each packet includes a special section, called the *recovery journal*.

The recovery journal codes the history of the stream, back to an earlier packet called the *checkpoint packet*. The range of coverage for the journal is called the *checkpoint history*. The recovery journal codes the information necessary to recover from the loss of an arbitrary number of packets in the checkpoint history.

When a receiver detects a packet loss, it compares its own knowledge about the history of the stream with the history information coded in the recovery journal of the packet that ends the loss event. By noting the differences in these two versions of the past, a receiver is able to transform all indefinite artifacts in the rendered performance into transient artifacts, by executing MIDI commands to repair the stream.

In most respects, senders are not required to use specific algorithms to create recovery journals, and receivers are not required to use specific algorithms to locate artifacts in a journal or to perform repairs on a stream. Senders may use any algorithm that produces correct journals for a stream (as defined by [4]), and receivers may use any algorithm that correctly repairs the stream (as defined by [4]).

5.1. Bandwidth Efficiency

To simplify our introduction to the journal system, we described the journal as coding a "history" of the stream. However, if the journal coded a simple log of all MIDI commands in the stream, the payload size would grow in an unbounded way in time.

In practice, two factors control the journal size. First, the journal does not code a command log. Instead, the journal codes the minimal amount of data needed to ensure that the rendered performance does not contain indefinite artifacts. In the general case, the journal does not code sufficient information to reconstruct the `MIDI list` fields of lost packets.

In addition, in preparing a journal, senders examine the the most recent RTCP packet from each receiver (in particular, the EHSNR field described in Section 2). The sender selects a checkpoint packet identity that generates the smallest recovery journal that provides protection from indefinite artifacts.

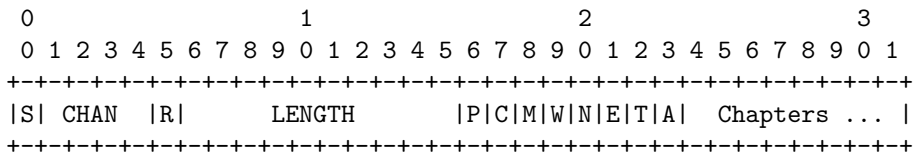

Fig. 6: Channel journal format.

As a general rule, a sender trims the recovery journal in response to a received RTCP packet, and a sender expands the recovery journal after sending an RTP packet. After an initial transient, the average amount of session time coded by the journal reaches an equilibrium value. For typical sessions, the average value is in the 1-10 second range.

[3] measured a musical performance system running over wide area network that used an early version of RTP MIDI. The MIDI musician in these experiments was a fast-playing pianist, who also made heavy use of continuous controllers. The median payload bandwidth of the stream was 4.7 kb/s. This bandwidth is in the range of modern Internet telephony voice codecs.

6. RECOVERY JOURNAL STRUCTURE

In this section, we describe the design of the recovery journal bitfields. The recovery journal has a hierarchical structure. Figure 5 shows the top level of the recovery journal. This structure is placed in the `Recovery journal` field in Figure 2.

The top-level bitfield may include a system journal (`S-Journal`), which codes recovery information for MIDI Systems commands, and `Channel Journal`s, which code recovery information for a specific MIDI channel (0-15).

In this discussion, we focus on the channel journal. Figure 6 shows the channel journal format. Header fields specify the MIDI channel number `CHAN` and the channel journal size `LENGTH`.

A list of *chapters* follows the header: different chapters code recovery information for different MIDI command types.

We list the chapter types below:

- Chapter P: Program Change (`0xC`)
- Chapter C: Control Change (`0xB`)
- Chapter M: Parameter System (part of `0xB`)
- Chapter W: Pitch Wheel (`0xE`)
- Chapter N: NoteOff (`0x8`), NoteOn (`0x9`)
- Chapter E: Note Command Extras (`0x8`, `0x9`)
- Chapter T: Channel Aftertouch (`0xD`)
- Chapter A: Poly Aftertouch (`0xA`)

The bit flags at the end of the header (`P|C|M|W|N|E|T|A`) act as a *table of contents* for the chapter list. If the bit flag associated with a chapter is set, the chapter appears in the `Chapters` field.

For each chapter, the payload format specification [4] precisely defines when the chapter must appear in the journal, and what recovery information the chapter must code. These definitions form the heart of the recovery journal system. In an implicit way, they tell senders how to create recovery journals that protect against indefinite artifacts, and they tell receivers how to repair the stream.

In this discussion, we confine our attention to a simple chapter. Chapter W protects the MIDI Pitch Wheel command (`0xE`). Typically, MIDI piano keyboards send Pitch Wheel commands when the player moves a "pitch wheel" controller located next to the piano keys. Sound generators react to Pitch Wheel commands by "bending" the pitch of current and future notes.

Figure 7 shows the Chapter W bitfield. Chapter W has a fixed size of 2 octets. Roughly speaking, Chapter W must appear in a channel journal if a Pitch Wheel command on the channel appears in the checkpoint history. The `FIRST` and `SECOND` fields

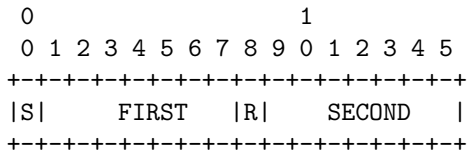

Fig. 7: Chapter W format.

code the 14-bit data value of the most recent Pitch Wheel command on the channel.

Note that Chapter W differs from a simple command log in two respects: only the most recent Pitch Wheel command is coded in the journal, and Chapter W need not appear in the journal once this command no longer appears in the checkpoint history. Senders use these features to minimize the size of the recovery journal.

A receiver uses Chapter W to repair a stream that has lost Pitch Wheel commands, as we describe in detail later in this paper (Section 8).

Finally, we note that the structures shown in Figures 5, 6, and 7 begin with an `S` flag bit. `S` bits appear throughout the recovery journal, and mark the parts of the journal that have recently changed. Receivers use the `S` bits to speed the search for indefinite artifacts, in the common case of a loss event consisting of a single lost packet.

7. SENDING RTP MIDI STREAMS

So far in this paper, we have focused on the static qualities of RTP MIDI: the layout of the payload bitfields and the meaning of the bitfield values.

In the next two sections, we focus on the dynamic qualities of RTP MIDI: the algorithms senders use to transmit packets (this section), and the algorithms receivers use to render packets (Section 8). These algorithms are not normative, and may vary within and across application classes. The protocol design ensures that applications that use different algorithms are able to interoperate.

We describe these algorithms in the context of a music performance application. Two musicians, located in different physical locations, interact over a network to perform as they would if located in the same room. Underlying the performances are RTP MIDI streams sent over unicast UDP. The description below is an abbreviated version of [3] and [5].

In this example, the two musicians use MIDI controllers that are connected to networked personal computers. Each computer sends and receives an RTP MIDI stream over the network. Each computer generates audio in response to the MIDI produced by the local player and in response the received RTP MIDI stream. The two computers execute identical rendering algorithms for the MIDI commands associated with a particular musician.

The packet sending system is a real-time data-driven entity. On an on-going basis, the sender checks to see if the local player has generated new MIDI data. At any time, the sender may transmit a new RTP packet to the remote player, for the reasons described below:

1. New MIDI data has been generated by the local player, and the sender decides it is time to issue a packet coding the data.
2. The local player has not generated new MIDI data, but the sender decides too much time has elapsed since the last RTP packet transmission. The sender transmits a packet in order to relay updated RTP header and recovery journal data.

In the first case, the `MIDI list` field (Figure 3) codes the new MIDI data. In the second case, the `MIDI list` is empty.

We emphasize the *decisions* made by the sender, because these decisions determine the quality/bandwidth tradeoff of the stream.

For example, to minimize queuing latency, the sender may wish to send an RTP packet as soon as a complete MIDI command arrives from the local musician. Alternatively, by waiting a few milliseconds for the potential arrival of a follow-on command, a sender may significantly reduce stream bandwidth. In this case, the best sending algorithm depends on desired quality/bandwidth tradeoff.

As a second example, consider the situation where the local player produces a MIDI NoteOff command (which the sender promptly transmits in a packet), but then 5 seconds pass before the player produces

another MIDI command (which the sender transmits in a second packet).

What happens if the packet coding the NoteOff is lost? The receiver is not be aware of the packet loss incident for 5 seconds, and the rendered MIDI performance contains a note that sounds for 5 seconds too long. The note ends upon reception of the second packet, because the recovery journal instructs the receiver to end the note.

To reduce the severity of this transient artifact, a sender may transmit packets with empty `MIDI lists` to "guard" the stream during silent sections. In this example, if a guard packet was sent 10 ms after the NoteOff packet, and if the guard packet arrived safely at the receiver with normal network latency, the perceptual impact of the transient artifact would be small. The best design for a guard algorithm depends on the desired quality/bandwidth tradeoff.

Finally, we note that senders reduce the packet sending rate in response to network congestion, as required by [1].

7.1. Sender Journal Maintenance

A complete description of how a sender generates the recovery journal for a packet is beyond the scope of this paper. See [5] for a thorough description, complete with C language data structures.

Here, we show a brief sketch of the packetization process. A sender prepares a new RTP MIDI packet by following these steps:

1. Generate the RTP header for the new packet (Figure 1).
2. Generate the payload header (Figure 2) and `MIDI list` field (Figure 3) for the new packet.
3. Generate the recovery journal for the new packet. To do so, the sender encodes the contents of the Recovery Journal Sending Structure (RJSS) into the bitfields shown in Figure 5-7 (and into many other bitfields not shown in this paper). The RJSS is a special data structure that codes a history of the stream.
4. Send the packet over the network.
5. Update the RJSS to code the commands in the `MIDI list` field of the packet sent in Step 4. This step extends the checkpoint history to cover the sent commands.

In addition to sending RTP packets, the sender also monitors incoming RTCP packets (Section 2). RTCP packets include the EHSNR field, which indicates the most recent RTP packet a receiver has seen. The sender uses the EHSNR value to remove RJSS entries that are no longer relevant to any receiver. In this way, the sender trims the recovery journal sent in the next packet.

8. RECEIVING RTP MIDI STREAMS

In this section, we discuss receiver algorithms for RTP MIDI streams. The description below is an abbreviated version of [3] and [5].

To begin, we imagine that the application runs over an ideal network. On this network, packets are never lost or reordered, and the end-to-end latency is constant.

In addition, the senders in this application ensure that all commands coded in a packet's `MIDI list` share the same timestamp, which is identical to the packet's RTP timestamp. The senders also maintain a constant relationship between the RTP timestamp value and the packet sending time: if two packets have timestamps that differ by 1 second, the second packet is sent exactly 1 second after the first packet.

Under these conditions, a simple receiver algorithm may be used to render a high-quality performance. Upon the receipt of an RTP packet, the receiver immediately executes the commands coded in the `MIDI list`. The command timestamps are ignored.

Unfortunately, this simple algorithm breaks down once we relax our assumptions about the network and the sender:

- If we permit lost packets to occur in the network, the algorithm may produce indefinite and transient audio artifacts.
- If we permit the network to exhibit variable latency, the algorithm modulates the network jitter onto the rendered audio.

- If we permit a `MIDI list` to code commands with different timestamps, the algorithm adds temporal jitter to the rendered performance, as it ignores `MIDI list` timestamps.

To compensate for sender and network jitter, and to support the use of timestamps in `MIDI lists`, receivers may use the following methods:

- Playout buffering. RTP audio receivers use a playout buffer to smooth network jitter. Audio buffer architectures are easy to adapt to RTP MIDI. As in the audio case, a playout buffer increases the end-to-end latency of an RTP MIDI stream.
- Semantic filtering. For latency-sensitive applications running over networks with low jitter, semantic filtering offers an alternative to buffering. The receiver maintains a model of network latency, and uses it to mark each MIDI command as "on-time" or "late". On-time commands are executed immediately. Late commands are handled in a command-specific way. For example, the receiver in [3] discards late NoteOn commands, but executes all other late commands.

To handle lost packets, receivers use the recovery journal system, as we discuss next.

8.1. Receivers and Recovery Journals

To prepare for recovery from lost packets, receivers maintain a special data structure, the Recovery Journal Receiver Structure (RJSS).

The RJRS codes information about the MIDI commands the receiver executes (both incoming stream commands and self-generated recovery commands). At the start of the stream, the RJRS is initialized to code that no commands have been executed. Immediately after executing a MIDI command, the receiver updates the RJRS with information about the command.

The receiver keeps track of the highest sequence number received in the stream, and predicts that an incoming packet will have a sequence number one greater than this value. If the sequence number of an incoming packet is greater than the prediction, a packet loss has occurred.

Upon the detection of a packet loss, the receiver examines the recovery journal of the packet that ends the loss. For each channel journal (Figure 6) in the recovery journal, the receiver compares the data in each chapter journal to the RJRS data for the chapter. If the data are inconsistent, the algorithm infers that MIDI command(s) related to the chapter journal have been lost. The recovery algorithm executes MIDI commands to repair the loss, and updates the RJRS to reflect the repair. Payload features, such as the `S` bits described in Section 6, streamline the comparison process.

As an example of a repair operation, we consider a repair algorithm for Chapter W (the Pitch Wheel command chapter, shown in Figure 7). For each MIDI channel, the RJRS codes the 14-bit data value of the most recent Pitch Wheel command that has arrived on a channel.

At the end of a loss event, a receiver may find a Chapter W bitfield in a channel journal. If the value in the Chapter W bitfield does not match the RJRS pitch wheel value, one or more commands have been lost.

To recover from this loss, the receiver immediately executes a MIDI Pitch Wheel command on the channel, using the data value coded in the recovery journal. The receiver then updates the RJRS to reflect the executed command.

9. SESSION MANAGEMENT OVERVIEW

So far in this paper, we have focused on the operation of an ongoing RTP MIDI stream. To conclude the paper, we look at the protocol features for customizing the characteristics of a stream at the start of a session.

RTP MIDI's customization tools do not work within RTP itself. Instead, the tools work within the IETF protocols that are used for media session management. The Session Initiation Protocol (SIP, [19]) performs session management for interactive applications such as telephony, and the Real Time Streaming Protocol (RTSP, [20]) performs session management for content-streaming applications like Internet radio. In this paper we focus on SIP, whose operation we describe below.

9.1. Session Initiation Protocol (SIP)

In conventional switched-circuit telephony, a person who subscribes to a telephone service is given a telephone number by the phone company. The person distributes this number to others, who may contact the person by dialing the number on the keypad of their own phone.

We use the term *call management* to describe the actions a phone system takes to support the user experience of dialing the number and listening for a ringing or busy signal, and for other services such as call termination. Call management does not include audio transport.

SIP is able to perform call management for interactive Internet multimedia sessions (Internet telephony, video conferences, etc). Following the telephony analogy, we use the term *SIP phone* (or just *phone*) to describe user devices, even though the form and media capabilities of a SIP phone may share little in common with conventional telephones (particularly true for MIDI "phones"!).

SIP is a very general protocol, and can support many models for user identity. In this paper, we assume that a user's SIP identity takes the form of a SIP URL, such as `sip:user@machine`. The `sip:machine` is controlled by an entity that has some sort of relationship to the user (employer, service provider, etc). The `user` is the account name the entity provides to the user.

For example, `sip:lazzaro@cs.berkeley.edu` could be the SIP identity for one of the authors of this paper. Just as the author distributes his email address so that others may email him, the author distributes his SIP URL so that others may phone him.

`sip:cs.berkeley.edu` provides a service to accept phone calls for `lazzaro`, even if he is not able to take calls at the moment the call arrives. If `lazzaro` is unavailable, the call might be forwarded to a network service that acts as an answering machine.

To signal that he is available to accept calls, `lazzaro` sends a SIP `REGISTER` request to `sip:cs.berkeley.edu`. Information sent in the `REGISTER` includes contact information for the phone (network address, ports) and the types of media the phone supports (audio, video, text, MIDI). After registration, `sip:cs.berkeley.edu` routes SIP requests for `sip:lazzaro@cs.berkeley.edu` to `lazzaro`'s phone.

Imagine that `marylou`, with SIP URL `sip:marylou@example.com` wishes to place a call to `sip:lazzaro@cs.berkeley.edu`. When `marylou` places the call, her phone creates a SIP `INVITE` request and sends it to `sip:example.com`. The `INVITE` consists of two parts: a list of SIP *headers*, followed by a *message body*.

The SIP headers, like the headers on an email message, encode the SIP protocol itself. For example, SIP header fields would specify that the message is an `INVITE` request to start a call, and specify the recipient as `sip:lazzaro@cs.berkeley.edu`.

Upon reception of the `INVITE`, `sip:example.com` examines the SIP headers and forwards the `INVITE` to `sip:cs.berkeley.edu`, which in turn forwards the `INVITE` to `lazzaro`'s SIP phone. Thus, `marylou`'s phone sends an `INVITE` to `lazzaro`'s phone, even though `marylou`'s phone did not know the network address of the phone (all it knew was `lazzaro`'s SIP URL).

The message body of the `INVITE`, like the message body of the email, is intended for the recipient. In our example, the message body is a session description, coded in the Session Description Protocol (SDP, [21]), that describes the session `marylou` wishes to start with `lazzaro`.

The session description encodes the network address and port numbers where `marylou`'s phone wishes to receive the media stream, and the protocols the phone wishes to use for media (for example, RTP/AVP over UDP, using the RTP MIDI payload format for a network musical performance session).

If `lazzaro` wishes to take the call, and if the session description contents are acceptable to his SIP phone, a SIP `OK` response is sent back to `sip:marylou@example.com`. The body of the `OK` also contains a session description, that encodes information about how `lazzaro`'s phone wishes to receive its media stream. Upon receipt of the `OK`, `marylou`'s phone sends `lazzaro` a SIP `ACK` message, to complete the three-way handshake that starts a SIP call (`INVITE`, `OK`, `ACK`).

The phones use the information in the received session descriptions to start bidirectional RTP MIDI

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4
  first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP4 192.0.2.94
a=rtpmap:96 rtp-midi/44100
```

Fig. 8: Minimal native session description.

flows, and the performance begins. Note that the media flows directly between `marylou`'s phone and `lazzaro`'s phone, without passing through `sip:cs.berkeley.edu` or `sip:example.com`.

We note that upon receipt of `marylou`'s `INVITE`, `lazzaro`'s SIP phone might examine the session description and find aspects of it incompatible or undesirable. In this case, the phone may choose to reject the `INVITE` (via a SIP rejection response to the `INVITE`), and then propose an alternative session description in a new `INVITE` to `marylou`. This mechanism provides part of a simple session description negotiation system, called the Offer/Answer Protocol [28].

SIP also lets callers query a call recipient for its desired session description details before placing the call, using the `OPTIONS` request. `OPTIONS` is sent to a user's SIP URL, which returns a user's preferences coded in a session description. A caller uses the information returned by `OPTIONS` to create an acceptable `INVITE` request.

10. SESSION DESCRIPTION EXAMPLES

In this section, we show example session descriptions for RTP MIDI sessions. Session descriptions are coded with the Session Description Protocol (SDP, [21]). We also introduce the RTP MIDI MIME parameters, which may be used to customize sessions.

RTP MIDI streams come in two flavors: native streams, which are used by general-purpose applications, and `mpeg4-generic` streams, which are used by MPEG 4 applications [17].

Figure 8 shows a minimal native session description. We refer to this session description as *minimal* because it does not use RTP MIDI parameters to customize the stream. The list below describes the key information coded in session descriptions (see [21] for more details):

- The *media* line (begins with `m=`) codes that an audio RTP/AVP stream is delivered to UDP port 5004, and that the `PT` header field of each RTP packet in the stream is set to 96. Note that RTP MIDI is classified as an audio payload format.
- The *connection* line (begins with `c=`) codes that the stream is sent to the IP4 network address 192.0.2.94.
- The rtpmap *attribute* line (begins with `a=`) codes that the stream uses RTP MIDI, with a clock rate of 44,100 Hz.

For typesetting reasons, the session description examples in Figures 8-10 break long SDP lines over several print lines. Note that such line breaks are not legal SDP syntax.

10.1. Minimal Streams

In [4], we define the characteristics of a minimal stream. Minimal streams are intended to provide a reasonable set of default behaviors, that work well for many general-purpose MIDI applications.

A minimal stream sent over UDP transport (such as the minimal native stream shown in Figure 8) uses the recovery journal; a minimal stream sent over TCP does not use the journal. Minimal streams interpret MIDI command timestamps as execution times (as described in Section 3.1), and permit the amount of media time coded by a packet to range from 0-200 ms.

A minimal `mpeg4-generic` stream uses of one of the MPEG 4 renderers: General MIDI [2], DLS 2 [29], or Structured Audio [17]. A minimal native stream does not specify a rendering method.

Figure 9 shows a session description for a minimal `mpeg4-generic` RTP stream. The parameters on the `a=fmtp` line specify that General MIDI must be used to render the stream.

10.2. Customizing Streams

If a minimal native or `mpeg4-generic` stream is not a good match for an application, RTP MIDI parameters and standard SDP attribute parameters may

```
v=0
o=lazzaro 2520644554 2838152170 IN IP6
  first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP6 FF1E:03AD::7F2E:172A:1E24
a=rtpmap:96 mpeg4-generic/44100
a=fmtp:96 streamtype=5; mode=rtp-midi;
  profile-level-id=12; config=7A124D54
  6864000000060000000100604D54726B0000
  000400FF2F000
```

Fig. 9: Minimal `mpeg4-generic` description.

```
v=0
o=lazzaro 2520644554 2838152170 IN IP4
  first.example.net
s=Example
t=0 0
m=audio 5004 RTP/AVP 96
c=IN IP4 192.0.2.94
a=rtpmap:96 rtp-midi/44100
a=fmtp:96 tsmode=async; linerate=320000;
  octpos=first
a=sendonly
```

Fig. 10: A session that uses RTP MIDI parameters

be used to customize the stream. [4] defines usage guidelines for 28 parameters, grouped into 5 functional roles:

- **Journalling.** Parameters offer fine control over the recovery journal system. Parties may also use the parameters to define the subset of MIDI commands that appear in a stream.
- **Timestamp semantics.** [4] defines parameters to specify three types of semantics for MIDI command timestamps: the default semantics described in Section 3.1 of this paper, and two types of semantics that are useful for sampling MIDI 1.0 DIN cable flows.
- **Packet timing.** Parameters let parties set the average and maximum amount of media time coded in a packet, and let parties set a minimum sending rate. These parameters let a simple receiver implementation signal its stream preferences to a sender.
- **Multiple streams.** Parameters support splitting a single MIDI command stream (16 voice channels + systems) into multiple RTP MIDI streams. An application may use these parameters to send voice channel commands over a UDP stream while sending system commands over a TCP stream. Parameters also support sending several MIDI sources in an ordered way (stream #1 codes voice channels 0-15, stream #2 codes voice channels 16-31, etc). Related parameters support multi-stream synchronization.
- **Rendering**. A set of parameters specify the rendering method for a stream. The specification method is extensible: renderer developers may register a MIME type [18], which may be used in session descriptions to specify the renderer. Operating system APIs may also be registered, so that an RTP stream may be presented to applications in the same manner as a local FireWire or USB MIDI stream.

Figure 10 shows an example use of RTP MIDI parameters in a session description. The `a=fmtp` line uses the `tsmode`, `linerate`, and `octpos` parameters to specify that command timestamps represent an asynchronous sampling of a MIDI time-of-arrival source (such as MIDI 1.0 DIN cables).

This specification is useful to the receiver, because it indicates that two commands cannot have identical timestamps (because only one command appears on the source at a given moment). Therefore, commands whose timestamps indicate a relative arrival time limited by the cable sending rate (coded by `linerate`) are indistinguishable from simultaneous commands.

11. ACKNOWLEDGMENTS

We thank the networking, media compression, and computer music community members who have commented or contributed to the effort, including Kurt B, Cynthia Bruyns, Steve Casner, Paul Davis, Robin Davies, Joanne Dow, Dominique Fober, Philippe Gentric, Michael Godfrey, Chris Grigg, Todd Hager,

Michel Jullian, Phil Kerr, Young-Kwon Lim, Jessica Little, Jan van der Meer, Colin Perkins, Charlie Richmond, Herbie Robinson, Larry Rowe, Eric Scheirer, Dave Singer, Martijn Sipkema, William Stewart, Kent Terry, Magnus Westerlund, Tom White, Jim Wright, Doug Wyatt, and Giorgio Zoia.

We also thank the members of the San Francisco Bay Area music and audio community for creating the context for the work, including Don Buchla, Chris Chafe, Richard Duda, Dan Ellis, Adrian Freed, Ben Gold, Jaron Lanier, Roger Linn, Richard Lyon, Dana Massie, Max Mathews, Keith McMillen, Carver Mead, Nelson Morgan, Tom Oberheim, Malcolm Slaney, Dave Smith, Julius Smith, David Wessel, and Matt Wright.

12. REFERENCES

[1] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 3550: RTP: A transport protocol for real-time applications, 2003.

[2] MIDI Manufacturers Association. The complete MIDI 1.0 detailed specification, 1996.

[3] J. Lazzaro and J. Wawrzynek. A case for network musical performance. In *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Port Jefferson, New York, 2001.

[4] J. Lazzaro and J. Wawrzynek. Internet-Draft: RTP Payload Format for MIDI. The current version of this document appears on the AVT webpage (http://www.ietf.org/html.charters/avt-charter.html). Page down to "Internet-Drafts". Note that Internet-Drafts are works in progress, that may be updated, obsoleted, or replaced by other documents at any time.

[5] J. Lazzaro and J. Wawrzynek. Internet-Draft: An Implementation Guide for RTP MIDI. The current version of this document appears on the AVT webpage (http://www.ietf.org/html.charters/avt-charter.html). Page down to "Internet-Drafts". Note that Internet-Drafts are works in progress, that may be updated, obsoleted, or replaced by other documents at any time.

[6] J. Lazzaro. Sfront source code release. http://www.cs.berkeley.edu/∼lazzaro/sa/.

[7] D. Fober. Real-time MIDI data flow on Ethernet and the software architecture of MidiShare. In *Proceedings of the International Computer Music Conference* pp. 447-450, San Francisco, 1994.

[8] M. Wright and A. Freed. Open Sound Control: A new protocol for communicating with sound synthesizers. In *Proceedings of the International Computer Music Conference*, 1997.

[9] P. Kerr. Standard for transmission of Musical Instrument Digital Interface (MIDI) data within local area networks: Distributed MIDI - DMIDI. Under development by the IEEE P1639 Working Group, 2003.

[10] J. Dow. The E-Show MIDI tools and network drivers. Product from Richmond Sound Design, Ltd, Vancouver, CA.

[11] E. Schooler. Distributed music: a foray into networked performance. *International Network Music Festival*, Santa Monica, CA, 1993.

[12] J. Escobar, D. Deutsch and C. Partridge. Flow synchronization protocol. *IEEE/ACM Transactions on Networking*, Vol. 2, No. 2, pp. 111-121, 1994.

[13] P. Hope. MMidi: the MBone MIDI Tool. Multimedia Networks Group, University of Virginia, http://www.cs.virginia.edu/ mn-group/projects/mmidi/, 1996.

[14] C. Chafe, S. Wilson, R. Leistikow, D. Chisholm, and G. Scavone. A simplified approach to high quality music and sound over IP. In *COST-G6 Conference on Digital Audio Effects*, pp. 159–164, 2000.

[15] J. R. Cooperstock and S. P. Spackman. The recording studio that spanned a continent. *Proceedings of the International Conference on WEB Delivering of Music* pp. 161-167, 2001.

[16] D. Fober, Y. Orlarey, and S. Letz. Real time musical events streaming over Internet. *Proceedings of the International Conference on WEB Delivering of Music*, pp. 147-154, 2001.

[17] ISO 14496 (MPEG-4), Part 3 (Audio), Subpart 5 (Structured Audio), 1999.

[18] N. Freed, J. Klensin, and J. Postel. RFC 2048: MIME part four: registration procedures, 1996.

[19] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261: SIP: Session Initiation Protocol, 2002.

[20] H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326: Real Time Streaming Protocol (RTSP), 1998.

[21] M. Handley and V. Jacobson. RFC 2327: SDP: Session Description Protocol, 1998.

[22] H. Schulzrinne and S. Casner. RFC 3551: RTP profile for audio and video conferences with minimal control, 2003.

[23] J. Postel. RFC 768: User Datagram Protocol, 1980.

[24] C. Perkins. *RTP: Audio and Video for the Internet*, Addison-Wesley, ISBN 0-672-32249-8, 2003.

[25] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, Pennsylvania, pp. 200–208, IEEE, 1990.

[26] J. Rosenberg and H. Schulzrinne. RFC 2733: An RTP payload format for generic forward error correction, 1999.

[27] J. Postel. RFC 793: Transmission Control Protocol, 1981.

[28] J. Rosenberg and H. Schulzrinne. RFC 3264: An offer/answer model with SDP, 2002.

[29] MIDI Manufacturers Association. The MIDI Downloadable Sounds specification, v98.2, 1998.