INTERNET-DRAFT
August 21, 2001
Expires: February 21, 2002

John Lazzaro
John Wawrzynek
UC Berkeley

MWPP: A resilient MIDI RTP packetization for MPEG 4 Structured Audio

<draft-lazzaro-avt-mwpp-midi-mpeg4-00.txt>

Status of this Memo

Abstract

This memo describes the MIDI Wire Protocol Packetization (MWPP). MWPP is a resilient RTP packetization for the MIDI wire protocol, for use with MPEG 4 Structured Audio. MWPP is specialized for low-latency applications such as network musical performance. MWPP supports the subset of the MIDI wire protocol that may be coded in Structured Audio real-time streaming units (midi_event chunks of SA_access_units).

MWPP defines a multicast-compatible recovery system for gracefully handling lost and late packets during a network musical performance. MWPP also defines sender and receiver proxies that implement the recovery system, to support thin clients such as MIDI electronic pianos.

1. Introduction

The MIDI standard [1] began its life as a data networking protocol. The standard defines a wire protocol to interconnect electronic musical instruments into a distributed real-time system, using short coaxial "MIDI cables" for the physical layer.

The MPEG 4 Structured Audio [2] defines a language and run-time environment for software-based electronic musical instruments. The standard supports real-time MIDI control of instruments, using a subset of the MIDI wire protocol command set encoded into real-time streaming units (midi_event chunks in SA_access_units).

If a Structured Audio (SA) decoder (such as [3]) is running on a computer with a MIDI input jack, a musician can use a MIDI cable to connect an electronic piano keyboard to the Structured Audio decoder. In this way, the musician plays a software-based instrument in real-time with normative semantics, by using the SA_access_units mechanism.

If this computer is connected to the Internet, and if a second similarly-configured computer is also connected to the Internet at a different location, the two SA decoders could exchange SA_access_units, turning both local and remote MIDI data into sound. If the nominal latency is sufficiently low, musicians using these systems can engage in a network musical performance [5].

This memo describes the MIDI Wire Protocol Packetization (MWPP), a resilient RTP [4] packetization for the low-latency transmission of SA_access_units that encode the MIDI wire protocol. In this framework, each SA decoder sends MWPP RTP packets coding the MIDI events of its local player to remote players, and receives the MWPP RTP packets of the remote players. Each sender-receiver transport pair acts as a virtual MIDI extension cable.

Network musical performance systems work well when the nominal total latency between the participating musicians is reasonably short [5]. This memo does not address the nominal latency issue; we assume a system using MWPP has a sufficiently low nominal total latency to support the application.

MWPP is designed for use over UDP and other unreliable datagram transport: the design goal is graceful recovery from lost and late UDP packets, without using packet retransmission. To meet this goal, MWPP defines its own bitstream format; it does not use the bitstream format of the midi_event chunks of SA_access_units defined in [2]. MWPP also supports reliable transport such as TCP: it includes features to minimize bandwidth overhead when used with TCP.

Sending the MIDI wire protocol over unreliable transport is not trivial. The MIDI standard defines a set of commands, that reflect the gestures musicians make in playing their instruments ("NoteOn" command to start a new note, "NoteOff" command to end the note, etc). Gestural commands make MIDI data streams very compact, but also very fragile: a single lost "NoteOff" command could result in a sound that sustains indefinitely long.

MWPP defines a recovery system that ensures these sort of catastrophic command losses do not indefinitely impact a performance. The MWPP recovery system uses domain knowledge, which takes the form of the MIDI command semantics defined in [1] and the SA_access_unit midi_event semantics defined in [2].

The remainder of this memo describes MWPP in detail, and assumes a working knowledge of [1] and the MIDI-related sections of [2]. Readers unacquainted with these documents may read Section 6 of [5], which provide sufficient detail to understand this memo. MWPP as described in this memo is implemented in [3].

2. Sending MWPP RTP packets

This section describes sending MWPP RTP packets. We describe MWPP for use over unreliable datagram transport without sender proxies. Reliable transport and sender proxies are described in Section 9 of this memo.

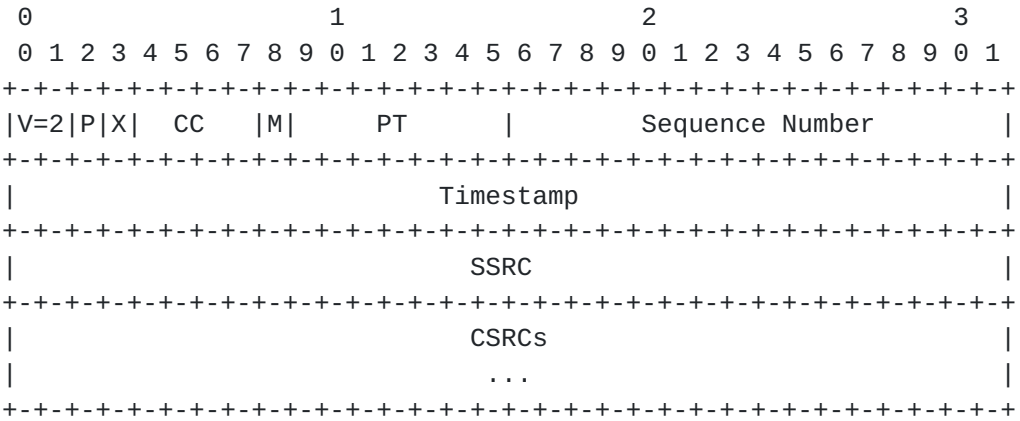


Figure 1 -- RTP header

An MWPP packet begins with a standard RTP header (Figure 1). MWPP does not use header extensions or (usually) the marker bit (see Section 9 for an exception). As is standard, each MWPP RTP packet sent has its sequence number incremented by one modulo 65536.

MWPP packets encode MIDI commands that are scheduled for execution at a particular moment in time on the sender's Structured Audio decoder. The RTP timestamp field encodes the moment of execution.

The timestamp has the units of the SDP rtpmap attribute srate; for simple uses of MWPP, this srate value is identical to the Structured Audio global srate parameter (see Section 10 for details). For example, if srate has a value of 44100Hz, two MWPP packets coding MIDI commands that are executed 2 seconds apart on the sender's SA decoder have RTP header timestamps that differ by 88200.

The timestamp is a monotonically increasing function of the sequence number, as expressed in modulo arithmetic. As is standard in RTP, the timestamp field is initialized to a randomly chosen value.

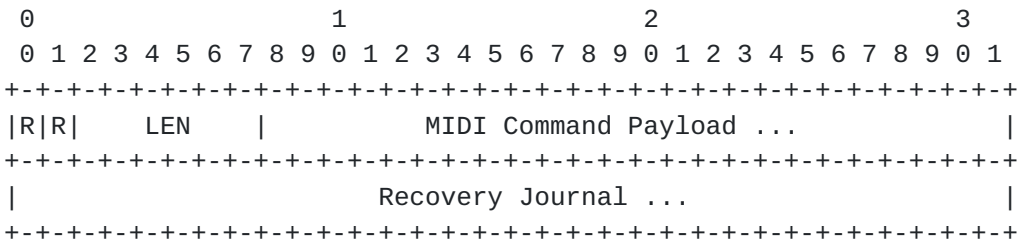


Figure 2 -- MWPP payload

The RTP header is followed by the MWPP payload. The MWPP payload has two variable-length sections: a MIDI command section, and a recovery journal.

The MIDI command section encodes the MIDI commands executed on the sender's SA decoder at the moment encoded by the RTP timestamp. The MIDI command section has a one-octet header, followed by the MIDI command payload.

The header codes the length (in units of octets) of the data field, via the 6-bit value LEN. A LEN value of 0 is legal, and codes an empty payload. The header also contains two reserved R bits. All reserved bits in this memo are named R: reserved bits MUST be set to zero by senders and ignored by receivers.

The MIDI command payload must contain zero or more complete MIDI commands. The first MIDI command in the MIDI data field MUST have a status octet, but subsequent commands MAY use the running status data compression scheme [1].

The MIDI command payload MUST only contain the MIDI commands that are legally coded in the midi_event chunk of SA_access_unit real-time streaming units of Structured Audio (i.e. all MIDI commands except the

MIDI System command (0xF)). Note that all commands in the MIDI command payload are scheduled for the same moment in time; the 320 us/octet serial delay of a MIDI cable is not emulated.

The MIDI command section of the MWPP payload is followed by the recovery journal. Information encoded in the recovery journal enables the receiver to gracefully recover from the loss of all RTP packets sent since an earlier RTP packet, called the checkpoint packet.

The growth in size of the recovery journal is limited in two ways.

- The recovery journal encodes the minimal session history that is needed for recovery, not a trace log of all MIDI commands sent since the checkpoint packet.

- A sender monitors the "last extended sequence number received" field of RTCP RR reports [4], and advance the checkpoint packet to reflect the known state of all receivers. This mechanism is multicast compatible, and does not require a change in the RTCP mechanism as described in [4].

Detailed information about the format of the recovery journal appears in Section 5.

3. Receiving MWPP RTP packets

In this section, we describe how receivers process RTP MWPP packets. We assume that the RTP sender and receiver use a transmission channel with sufficiently low nominal latency to support the application, but that transient network disturbances may result in lost packets, and in packets received with significantly longer latencies that the nominal latency.

When a new RTP packet arrives, the receiver first examines the timestamp field, and classifies the packet as either "ontime" or "late." To perform this classification, the receiver typically maintains a model of the latency of the channel (see Appendix B of [5] for an example latency model, that is implemented in [3]).

The receiver then examines the RTP sequence number, and classifies the situation as:

- Normal. The extended packet sequence number of new RTP packet is one greater than the extended sequence number of the last RTP packet number received.

- Sequence Break. The extended packet sequence number of new RTP packet

is greater than in the normal case. This classification also applies if the new RTP packet is the first RTP packet received in the session.

- Out of Order. The extended packet sequence number of new RTP packet is less than in the normal case.

The most common occurrence is for packets to be normal/ontime. In this case, the receiver schedules all commands in the MIDI command section of the MWPP payload for execution on the local SA decoder. The details of scheduling are implementation-specific: simple decoders (such as [3]) may execute MIDI commands as soon as they arrive, but other approaches are possible.

If a packet is normal/late, the MIDI command section of the MWPP payload is executed as in the normal/ontime case, except for MIDI NoteOn commands with non-zero velocity, which are discarded. These semantics prevent "straggler notes" from disturbing a performance, quiets "soft stuck notes" immediately, and updates all other MIDI state in an acceptable way.

If a packet is a sequence break packet, the receiver first processes the recovery journal section of the payload, as described in Section 5. This processing may result in the execution of one or more MIDI commands to gracefully recover from the packet loss. After processing the recovery journal, the receiver processes the MIDI command section of a sequence break packet as if it were a "normal" packet.

If a packet is an out of order packet, its MIDI command and recovery journal sections are ignored.

4. Sender Addendum: Guard Packets

One shortcoming of MWPP as presented in Sections 2 and 3 is that senders are not required to maintain a minimum sending rate for RTP packets. This can cause problems if a packet that encodes a MIDI NoteOff event is lost. Until another packet is sent, the recovery journal mechanism cannot function to quiet the stuck note.

MWPP senders may address this problem by sending RTP packets with empty MIDI command sections at regular intervals: the recovery journal section of these "guard packets" serves to quiet stuck notes and update the MIDI state of the receiver's SA decoder. Guard packets also serve to prevent intermediaries (such as Network Address Translators) from timing out their services.

The use of guard packets by senders is implementation-dependent (but see Section 12).

5. The Recovery Journal

The recovery journal section of MWPP RTP packets has a three-level structure:

- Top-level header. Encodes recovery journal structure.
- Channel journal header. Encodes recovery information for a single MIDI channel (a MIDI command executes on one of 16 MIDI channels).
- Chapters. Describes recovery information for a single MIDI command type. Chapters are specialized to the semantics of the command, as defined in [1] and [2].

In this section, we specify the format of the top-level and chapter journal headers. Subsequent sections describe how senders and receivers use these headers as part of the recovery system. Appendices describe the semantics of each journal chapter.

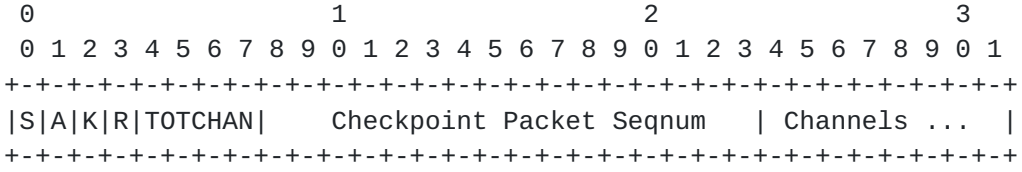


Figure 3 -- Top-level recovery journal format

Figure 3 shows the top-level structure of the recovery journal. A recovery journals consists of a 3-octet header, followed by a list of channel journals. Channel journals encode recovery information for a single MIDI channel.

If the A bit is set in the recovery journal header, the recovery journal is "empty", and contains no channel journals. If the A bit is clear, the channel journal list contains (TOTCHAN + 1) channel journals.

The recovery journal header includes an S bit. S bits appear on structures throughout the recovery journal format, with uniform semantics: if the S bit is set, the structure may be ignored if a sequence break of exactly one RTP packet triggered the recovery journal processing.

A set S bit on the recovery journal header indicates the previous sent packet is a guard packet (lost guard packets can be ignored because their MIDI command payloads are empty).

The 16-bit Checkpoint Packet Seqnum field codes the sequence number of

the checkpoint packet used by sender to create this journal. If this sequence number has changed since the last MWPP packet sent, the K bit is set, else it is clear.

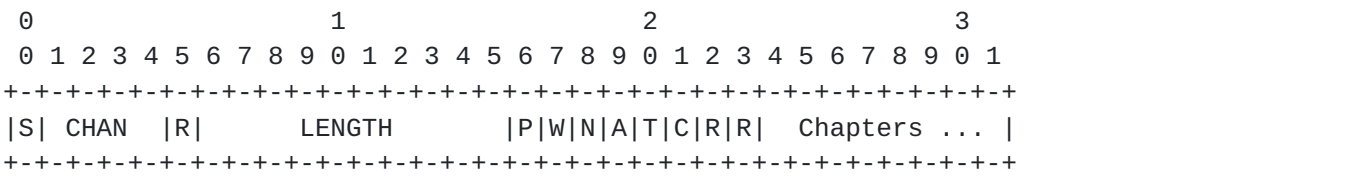


Figure 4 -- Channel journal format

Figure 4 shows the structure of a channel journal: a 3-octet header, followed by a list of leaf elements called chapters. A channel journal encodes recovery information for commands sent on the MIDI channel coded by the 4-bit CHAN header field. The 10-bit LENGTH field codes the number of octets in the channel journal, including the header.

The third octet of the channel journal header is the Table of Contents (TOC) of the channel journal. The TOC is a set of bits to encode the presence of a chapter in the journal. Each chapter contains information to recover from the loss of a certain class of MIDI commands:

- Chapter P: MIDI Program Change (0xC)
- Chapter W: MIDI Pitch Wheel (0xE)
- Chapter N: MIDI NoteOff (0x8), NoteOn (0x9)
- Chapter A: MIDI Poly Aftertouch (0xA)
- Chapter T: MIDI Channel Aftertouch (0xD)
- Chapter C: MIDI Control Change (0xB)

Chapters appear in a list following the header, in order of their appearance in the TOC. The Appendices of this memo describe the format of each chapter, and explain how senders and receivers use each chapter.

6. Sending Recovery Journals

In this section of the memo, we briefly describe how senders create recovery journals.

Senders maintain state about MIDI commands sent since the last checkpoint packet, using a data structure that records the RTP sequence number associated with each MIDI command (see [3] for a sample implementation). We refer to this data structure as the "recovery journal data structure" below.

To send a new MWPP packet, the sender first creates the MIDI command section of the packet. Then, the sender traverses the recovery journal

data structure to build the new recovery journal. Typically, the data structure elements match the structure of recovery journal chapters and headers, so that simple memory copies act to build a new journal. Timing-sensitive chapter data (such as the Y bits of Chapter N, as explained in Appendix A.3) are updated during the build process.

After the recovery journal is created, the sender inserts the MIDI commands encoded in the MIDI command section of the new packet into the recovery journal data structure. Data structure elements corresponding to the S bits of the recovery journal are updated during this insertion.

The reception of an RTCP RR packet may also result in an update of the recovery journal data structure. The sender first examines the "last extended sequence number received" field of the received RTCP RR packet, and combines it with the RTCP RR data from other receivers in the session. If the sender determines checkpoint packet may be safely updated from its current value, it traverses the recovery journal data structure, pruning MIDI data wherever possible, in order to reduce the size of future recovery journals.

7. Receiving Recovery Journals

In this section, we briefly describe how receivers parse recovery journals.

In the case of a loss of a single RTP packet, the receiver uses the S bits of the recovery journal to skip over channels and chapters which do not encode information about the lost packet. For each chapter whose S bit is clear, the receiver executes the chapter-specific recovery algorithm described in the Appendices.

In the case of a multi-packet loss, the S bits are ignored, and each chapter of each channel journal undergoes the recovery procedure described in the Appendices.

The recovery algorithms for most chapters require information about MIDI commands received in previous RTP packets. See [3] for a sample implementation of data structures to efficiently maintain this state.

Note that receivers MUST use the LENGTH field of the chapter journal header to traverse from chapter to chapter, and not rely on the sizes of each chapter journal. This restriction is needed for backward compatibility, as the R bits in the TOC may be used for new chapters in future versions of MWPP.

8. MWPP Startup Issues

The recovery journal mechanism depends on senders tracking the status of receivers, by:

- Knowing the first MWPP RTP packet sent to a new receiver.
- Examining the "last extended sequence number received" field of RTCP RR reports.

In simple unicast applications, both mechanisms work well. In more complex situations, such as true or simulated multicast transport, senders may not know of the presence of a receiver until the first RTCP packet arrives. In this case, lost packets early in a session may not be protected by the recovery journal mechanism, because the sender "incorrectly" moved the checkpoint packet.

Receivers can detect this situation by using the Checkpoint Packet Seqnum field coded in the recovery journal header, as shown in Figure 3. Note that this technique requires that receivers examine the recovery journal of every MWPP packet received, although the K bit that marks checkpoint updates minimizes the work per packet required.

To detect this situation, the receiver examines the Checkpoint Packet Seqnum, and checks to see if it is consistent with the reception history of the receiver. If an inconsistency is detected, receivers should assume that the sender is not aware of its existence, and take precautions to ensure that catastrophic MIDI errors do not occur (for example, NoteOn commands could be "timed out" with a matching NoteOff command after a suitably long period of time).

9. Reliable Transport and Proxies

The recovery journal adds significant overhead to MWPP. When sending MWPP over reliable transport (TCP, or a point-to-point reliable IP link) the recovery journal section of MWPP packets may be safely deleted without affecting the proper operation of the system.

MWPP recovery journals may also be safely deleted if the SAOL program running on the Structured Audio decoders uses application-layer recovery techniques that make the MWPP recovery journal scheme redundant.

Senders and receivers SHOULD use the Session Description Protocol (SDP) [6] mechanism described in Section 10 to indicate that an MWPP session does not use the recovery journal mechanism. In this case, the RTP header of each MWPP packet is constructed in accordance with Section 2 of this memo, specifically with respect to marker bit usage.

However, if SDP exchange or other out-of-band communication is not possible, senders MAY indicate the lack of a recovery journal section by setting the marker bit on each RTP packet, and receivers SHOULD be prepared to handle this case. This use of the marker bit is discouraged, because it may interfere with RTP header compression.

MWPP packets without recovery journals are also used in association with sender and receiver proxies. Sender and receiver proxies are used when Structured Audio clients are running on thin clients, such as electronic piano keyboards. These keyboards may have special-purpose audio processing hardware for Structured Audio decoding, but may have simple general-purpose processors that cannot handle the overhead of recovery journal send and receive operations.

If the thin client has a reliable channel to a suitable host, sender and receiver proxies may be used to offload the recovery journal processing task. In this scheme, the thin client would send and receive MWPP packets without recovery journals to the proxies. The sending proxy would add recovery journals to outgoing packets, and the receiving proxy would handle the lost and late packet processing described in Section 3. The sender and receiver proxies MUST also handle the RTCP duties for the thin client, because the thin client is not able to compute transport statistics correctly.

10. Session Description Protocol

This section describes Session Description Protocol (SDP) [6] definitions for MWPP.

The MIME name for this packetization is mpeg4-samidi. The SDP rtpmap attribute is declared as

a=rtpmap <payload> mpeg4-samidi/<srate>/<krate>/<rj>

The <srate> parameter codes the audio sampling rate used for the RTP timestamp field. Typically, this value corresponds to the srate global parameter value of the SAOL program (see [2] subpart 5.8.5.2.1). We specify <srate> in the rtpmap so that musicians can choose a different local srate value without disturbing the MWPP system.

The <krate> parameter codes the control sampling rate, which typically corresponds to the krate global parameter value of the SAOL program (see [2] subpart 5.8.5.2.2). This memo does not refer to the krate value; we include it in the rtpmap for possible future use, since like srate, musicians may wish to choose a different local krate value.

The <rj> parameter codes the presence or absence of recovery journals in

MWPP packets (see Section 9 for details). The two valid values for <rj> are "rj" and "no-rj". If the <rj> parameter does not exist, its value is assumed to be "rj".

For example, the following lines bind the packetization to dynamic payload number 96, and specifies an srate of 44100 Hz, a krate of 1260 Hz, and the presence of a recovery journal in each RTP packet:

```
m=audio 5004 RTP/AVP 96
c=IN IP4 171.64.92.160
a=rtpmap 96 mpeg4-samidi/44100/1260/rj
```

Note that the packetization does not directly support multiple 16-channel MIDI Input sources. Different UDP ports should be used in this case, each devoted to a single source:

```
m=audio 5004 RTP/AVP 96
c=IN IP4 171.64.92.160
a=rtpmap 96 mpeg4-samidi/44100/1260/rj
m=audio 5006 RTP/AVP 97
c=IN IP4 171.64.92.160
a=rtpmap 97 mpeg4-samidi/44100/1260/rj
```

Note that the SDP does not include a binary encoding of the SAOL program to run on the decoder (StructuredAudioSpecificConfig). This memo assumes that StructuredAudioSpecificConfig is sent out-of-band.

Finally, note that MWPP is self-framing, and so TCP transport is possible without explicit framing.

11. Security Considerations

Cryptographic authentication of incoming RTP and RTCP packets is highly recommended when using MWPP. Without such protections, attackers could forge MIDI commands into an ongoing session, potentially damaging speakers and eardrums. An attacker could also craft RTP and RTCP packets to exploit known bugs in the client, and take effective control of a client machine.

12. Congestion Control

MWPP has congestion control issues that are unique for an RTP audio packetization. When used for network musical performance, the packet rate is linked to the gestural rate of a human performer.

MWPP implementations SHOULD sense the MIDI stream for command patterns

that result in excessive packet rates, and filter these streams as part of MWPP to reduce the packet rate.

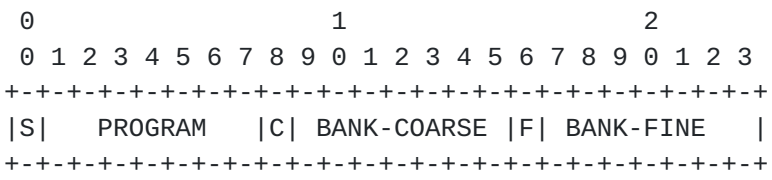
In addition, the guard packet mechanism described in Section 4 of this memo is a possible source of congestion control problems. Implementers MUST ensure that the guard packet strategies of senders are well behaved with respect to congestion control.

Appendix A.1. Chapter P: MIDI Program Change

Chapter P protects against the loss of MIDI Program Change commands, which the Structured Audio standard uses to bind SAOL instruments to MIDI channels. If a Program Change command is lost, notes played on a channel will sound with the incorrect timbre, or perhaps not sound at all.

To prepare for recovery, the receiver should store state for each channel, to indicate the program value of the last Program Change command received on this channel and the Bank Select values (Coarse and Fine) that were in effect at the time this Program Change command executed. The Bank Select values are issued via the MIDI Control Change command, and act to extend the range of program values. The stored state should also include flag bits to signify the null cases of no Program Change received, no Bank Select Coarse value received, and no Bank Select Fine value received.

The encoding for Chapter P is shown below:



The chapter has a fixed size of 24 bits. If the S bit is set to 1, the previous packet sent did not include a Program Change command on this channel, and the receiver can skip to the next Chapter if it is recovering from the loss of a single packet.

The PROGRAM field indicates the program value of the last Program Change command sent on this channel. If a Control Change command for the Bank Select Coarse controller was sent before this Program Change command, the C bit is set to 1, and the BANK-COARSE field is the Bank Select Coarse controller value that was sent. The F bit and BANK-FINE field code the Bank Select Fine value in the same manner.

The receiver should compare the values in Chapter P with the stored state for this channel, to determine if one or more Program Change commands were lost. If a loss is detected, the receiver should execute the Program Change command coded in Chapter P, and update its own recovery state.

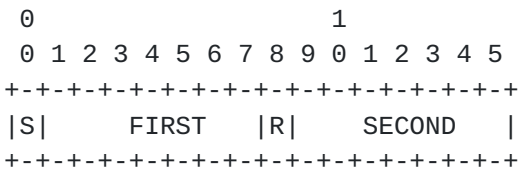
Appendix A.2. Chapter W: MIDI Pitch Wheel

Chapter W protects against the loss of MIDI Pitch Wheel commands. A common use of the Pitch Wheel command is to transmit the current position of a "pitch wheel" controller placed on the side of piano controllers, which players can use to dynamically alter the pitch of all depressed keys.

Structured Audio makes the current value of the Pitch Wheel available to SAOL programmers in the MIDIWheel standard name, which programmers typically use for continuous modification of instrument models, in a manner similar in spirit to the original pitch bend semantics of the controller. The recovery mechanisms in Chapter W are designed to protect the Pitch Wheel data stream for these types of SAOL programs.

To prepare for recovery, the receiver should store state for each channel, that codes the wheel value for the last Pitch Wheel command received, along with a flag bit to signify the null case of no Pitch Wheel command received.

The encoding for Chapter W is shown below:



The chapter has a fixed size of 16 bits. If the S bit is set to 1, the previous packet sent did not include a Pitch Wheel command on this channel, and the receiver can skip to the next Chapter if it is recovering from the loss of a single packet.

The FIRST and SECOND fields are the 7-bit values of the first and second data bytes of the last Pitch Wheel command sent on this channel. The R bit is reserved.

The receiver should compare the FIRST and SECOND fields in Chapter W with the stored pitch wheel state for this channel. If no difference is detected, Pitch Wheel commands may still have been lost, but any artifacts induced are transient in nature, and the receiver SHOULD take

no action.

If a difference is detected, the receiver should update its recovery state to reflect the values of the FIRST and SECOND fields. In addition, the receiver MAY execute a single Pitch Wheel command, or MAY plan a series of Pitch Wheel commands spaced over time.

Appendix A.3. Chapter N: MIDI NoteOff and NoteOn

Chapter N protects against the loss of MIDI NoteOn commands, which Structured Audio uses to launch new instrument instances, and MIDI NoteOff commands, which Structured Audio uses to schedule instances for termination. If a NoteOn command is lost, notes are skipped, a transient error. If a NoteOff command is lost, notes may sound indefinitely, an error that may be catastrophic for sustained timbres.

Structured Audio ignores the velocity field of the NoteOff command, and Chapter N does not protect this field. In the discussion below, our references to NoteOff commands include NoteOn commands with zero velocity, which have semantics identical to NoteOff commands in Structured Audio. Our references to NoteOn commands refer to NoteOn commands with non-zero velocity only.

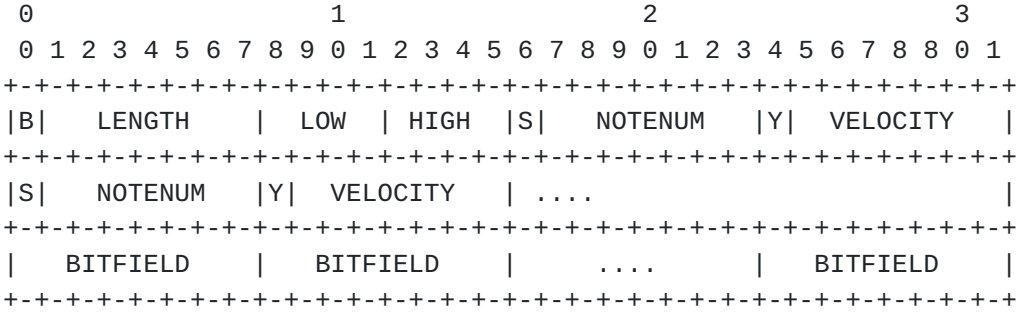
To prepare for recovery, the receiver should maintain state for each note number for an active channel. The recovery algorithms in this section references receiver recovery state variables, using the following nomenclature:

vel — This variable is initialized to zero at the start of a session. If a NoteOn command is executed for this note, vel is set to the velocity value of the command. If a NoteOff command is is executed for this note, vel is set to zero.

seq — Whenever a NoteOn or NoteOff command executes, seq is set to the extended sequence number of the RTP packet whose parsing resulted in execution of the command.

time — Time of the last NoteOn or NoteOff command, in the internal time units used by the application.

The encoding for Chapter N is shown below:



The chapter consists of a 2-byte header, followed by a list of 16-bit note logs, followed by a list of bitfields. A note number is represented in a note log or a bitfield if it has been used in a NoteOff or NoteOn command since the last checkpoint packet.

If the note number last appeared in a NoteOn command, it appears in a note log; if the note number last appeared in a NoteOff command, it appears in a bitfield.

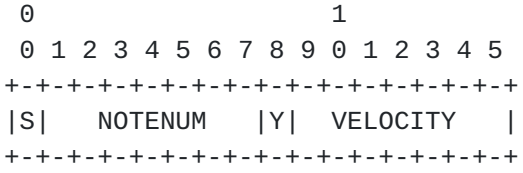
The 7-bit LENGTH field codes the number of note logs; zero is a valid value, and codes an empty note log. The maximum number of note logs is 127; in the musically unlikely case of 128 concurrent NoteOn commands, one NoteOn command is unprotected, risking a transient (not catastrophic) error on one note number.

The 4-bit fields LOW and HIGH determine the number of bitfield bytes that follow the note logs. A bitfield byte codes NoteOff information for eight consecutive MIDI note numbers, with the MSB representing the lowest note number. A 1 in a bit position indicates a NoteOff command has occurred for this note number since the last checkpoint packet, and that this NoteOff command occurred more recently than a NoteOn command.

The MSB of the first bitfield byte codes the note number 16*LOW, while the MSB of the last bitfield byte codes the note number 16*HIGH. If LOW is less that or equal to HIGH, there are (HIGH - LOW + 1) bitfield bytes in the chapter. To code a chapter with no bitfield bytes, senders MUST set LOW to 15 and HIGH to 0.

If the B bit is set to 1, the previous packet sent did not include a NoteOff command on this channel, and the receiver can skip parsing the bitfield section of the chapter if it is recovering from the loss of a single packet. To skip over a Chapter N, the receiver calculates the chapter length based on the values of LENGTH, LOW, and HIGH.

We now explain note log encoding, reproduced for reference below:



A note log will exist for a note number (coded by the 7-bit NOTENUM field) if a NoteOn command has occurred for this note number since the last checkpoint packet, and that this NoteOn command occurred more recently than a NoteOff command. The 7-bit VELOCITY field codes the velocity value for this NoteOn command; this field MUST not be zero.

If the S bit in a note log is 1, the previous packet sent did not include a NoteOn command for this note number, and the receiver can skip parsing this note log if it is recovering from the loss of a single packet.

The Y bit helps receivers to make the "play or skip" decision for recovered NoteOn events. Senders set Y to 1 for recovery packets sent shortly after the arrival of a NoteOn command from a MIDI controller; subsequent recovery packets are sent with Y = 0.

The tables below summarizes the recovery algorithm for Chapter N. For each set bitfield position, the receiver executes a strategy in the table column indicated by the recovery state variable vel for the note number:

vel	Diagnosis	Suggested recovery
zero	Either no note events have been lost, or a series of NoteOn -> NoteOff events have been lost.	Do nothing. Update seq to the current packet number.
non-zero	Either one NoteOff was lost or a series of NoteOff->On->Offs were lost.	Execute a NoteOff command to end the current note. Update velocity to 0, update seq to the current packet number.

For each note log, the receiver executes the strategy in the table column the recovery state variable vel for the note number:

vel	Diagnosis	Suggested recovery
zero	Either one NoteOn was lost, or a series of NoteOn->Off->On.	If Y = 0, never play the new note. If Y = 1, play a new note if an analysis of the current packet timestamp and the estimated delay from the sender shows the current packet is reasonably on time. Update vel to value VELOCITY, and seq to the current packet number.
non-zero	Either no note events have been lost, or a series of NoteOff-> NoteOn events have been lost.	Do one of: [1] Leave the current note (with velocity vel) playing. [2] End the current note but do not start a second note. [3] End the current note and start a second note with velocity VELOCITY. Based on the values of vel, VELOCITY, Y, seq, the current packet number, and the current checkpoint packet number. Update vel to value VELOCITY, and seq to the current packet number.

The second entry is this table is complex, due to the ambiguity of situation. A series of simple tests resolves the issue in most cases:

TEST 1: (seq < checkpoint packet sequence number)

If the value of seq is less than the current checkpoint packet sequence number, recovery is simple, since we know that the last NoteOn event executed is not a part of the note log, and so we know that a series of one or more NoteOff->NoteOn events have been lost. In this case, the current note should always be ended, and the execution of a new note should occur using the same criteria we use in the second entry of the table above. If seq indicates that the last NoteOn executed did occur during the current note log, our task is more difficult.

TEST 2: (vel != VELOCITY)

If vel doesn't equal VELOCITY, we know that the NoteOff corresponding to the last NoteOn executed was lost. In this case, the current note should always be ended, and the execution of a new note should occur using the same criteria we use in the second entry of the table above.

These two tests leave the (vel == VELOCITY) case to consider. This case is not a rare event, since many MIDI devices do not implement velocity sensing, and generate all NoteOn's with the same velocity value.

TEST 3: (Y == 1)

If Y is 1, the NoteOn in the note log occurred recently. If the receiver executed the last NoteOn recently (which was can tell by the time value for the last executed note), we know the note log represents the last executed note, and the correct action is to let the note continue to play. If the last note was not recently executed, it should be terminated, and an execution of a new note should occur using the same criteria we use in the second entry of the table above.

These three tests leave the following case unresolved: Y == 0 (the NoteOn in the note log didn't occur recently) and vel == VELOCITY (the last executed note and note log note are ambiguous). In this case, letting the last executed note continue to play, to be turned off by a forthcoming NoteOff command, is an acceptable result.

Appendix A.4. Chapter A: MIDI Poly Aftertouch

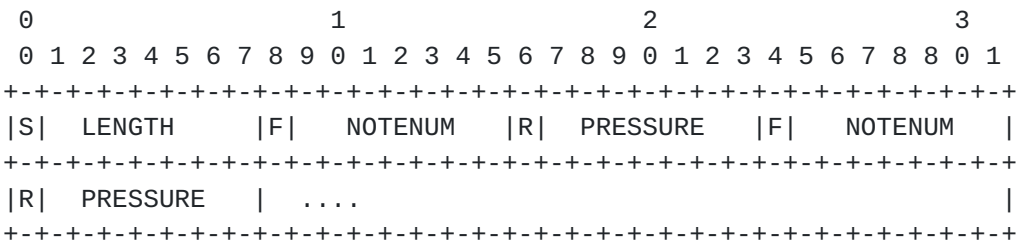
Chapter A protects against the loss of MIDI Poly Aftertouch commands.

This command supports piano keyboard controllers that have individual pressure sensors under each key, that generate a continuous signal whenever the key is depressed. Keyboard controllers that include these sensors send a stream of Poly Aftertouch commands for the duration of each key event. Because multiple keys may be down at once, the Poly Aftertouch command specifies a note number (0-127) as well as a pressure value (0-127).

SAOL programmers may access the last aftertouch value for each MIDI note in the MIDItouch[128] standard name array. Programmers typically use MIDItouch[] for continuous modification of instrument models parameters. The recovery mechanisms in Chapter A are designed to protect the aftertouch data stream for these types of SAOL programs.

To prepare for recovery, the receiver should store state for each note on each channel, that codes the pressure value of the last Poly Aftertouch command received, along with a flag bit to signify the null case of no Poly Aftertouch command received.

The encoding for Chapter A is shown below:



The chapter consists of a 1-byte header followed by a list of 16-bit note logs. Note logs exist for note numbers whose pressure value has been changed by a Poly Aftertouch command since the last checkpoint packet, and let receivers recover from the loss of those commands. Only one note log may exist in the note list for a particular note number.

The 7-bit LENGTH field codes the number of note logs minus one; the expression (1 + 2*(LENGTH + 1)) yields the number of bytes in the chapter. The maximum chapter length of 257 bytes protects the worst-case situation of Poly Aftertouch commands occurring for all 128 MIDI notes since the last checkpoint packet.

If the S bit is set to 1, the previous packet sent did not include a Poly Aftertouch command on this channel, and the receiver can skip to the next Chapter if it is recovering from the loss of a single packet.

For each note log, the 7-bit NOTENUM field identifies the MIDI note number of the log, and the 7-bit PRESSURE field indicates the pressure value of the last Poly Aftertouch command sent.

If the F bit is set to 1, the previous packet sent did not include a Poly Aftertouch for this note, and the receiver can skip to the next note log if it is recovering from the loss of a single packet.

If the F bit is 0, the receiver should compare the PRESSURE value with the stored pressure value for the note; if these values are different, the receiver should update its recovery state to reflect the value of the PRESSURE field. In addition, the receiver MAY execute a single Poly Aftertouch command, or MAY plan a series of Poly Aftertouch commands spaced over time.

Appendix A.5. Chapter T: MIDI Channel Aftertouch

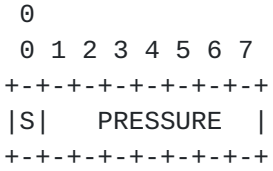
Chapter T protects against the loss of MIDI Channel Aftertouch commands. This command supports piano keyboard controllers that use a single pressure sensor for the entire keyboard. Keyboard controllers that include this sensor send a stream of Channel Aftertouch commands whenever at least one key is depressed. Unlike the Poly Aftertouch command, the Channel Aftertouch command does not specify a note number, only a pressure value (0-127).

Structured Audio makes the pressure value of the last Channel Aftertouch command available to SAOL programmers in all array positions of the MIDItouch[128] standard name array, which programmers typically use for continuous modification of instrument models parameters. The recovery mechanisms in Chapter T are designed to protect the aftertouch data stream for these types of SAOL programs.

To prepare for recovery, the receiver should store state for each channel, that codes the pressure value for the last Channel Aftertouch command received, along with a flag bit to signify the null case of no Channel Aftertouch command received.

SAOL programmers may access the last aftertouch value received via in the MIDItouch[128] standard name array; all array positions contain the same value. Programmers typically use MIDItouch[] for continuous modification of instrument models parameters. The recovery mechanisms in Chapter T are designed to protect the aftertouch data stream for these types of SAOL programs.

The encoding for Chapter T is shown below:



The chapter has a fixed size of 8 bits. If the S bit is set to 1, the previous packet sent did not include a Channel Aftertouch command on this channel, and the receiver can skip to the next Chapter if it is recovering from the loss of a single packet.

The 7-bit PRESSURE field indicates the pressure value of the last Channel Aftertouch command sent. The receiver should compare the PRESSURE value with the stored pressure value for the note; if these values are different,the receiver should update its recovery state to reflect the value of the PRESSURE field. In addition, the receiver MAY execute a single Channel Aftertouch command, or MAY plan a series of Channel Aftertouch commands spaced over time.

Appendix A.6. Chapter C: MIDI Control Change

Chapter C protects against the loss of MIDI Control Change commands. A Control Change command alters the 7-bit value of one of the 128 MIDI controllers. Most MIDI controllers are meant to be used as continuous parameters (for example, controller 7 is the Main Volume control), but some parameters have special semantics.

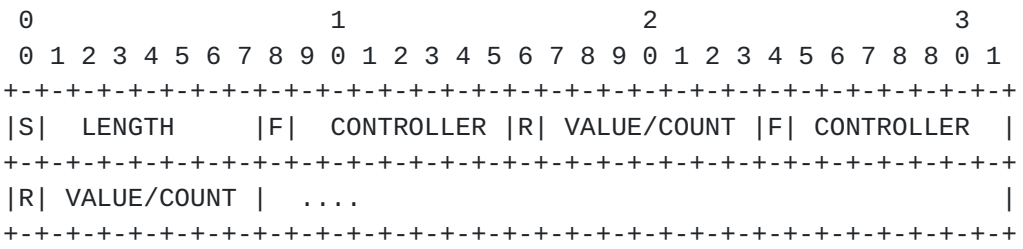
Structured Audio makes the current value of MIDI controllers available to SAOL programmers in the MIDIctrl[128] standard name array, that programmers typically use for continuous modification of instrument model parameters. The recovery mechanisms in Chapter C are designed to protect the Control Change data stream for these types of SAOL programs.

In addition, a Structured Audio decoder implements special semantics for the Bank Select Coarse and Fine, Sustain Pedal, All Notes Off, and All Sound Off controllers. The recovery mechanism in Chapter C, in conjunction with Chapter P, protects the special semantics of these five controllers.

To prepare for recovery, the receiver should store state for each channel about the Control Change data stream. For the All Notes Off and All Sound Off controllers, the receiver should keep a count, module 128, of the total number of Control Change commands received.

For all other controllers, the receiver should store the value of the last Control Command received, along with a flag bit to signify the null case of no Control Change command received. Note that this state should not reflect any changes to MIDIctrl[] made by assignment statements by SAOL code.

The encoding for Chapter C is shown below:



The chapter consists of a 1-byte header followed by a list of 16-bit controller logs. A controller log exist for All Notes Off and All Sound Off controllers if a new Control Change command has been received for the controller since the last checkpoint packet. For all other controllers, a controller log exists for controllers whose values have been changed by a Control Change command since the last checkpoint packet. Only one controller log may exist in the controller list for a particular controller number.

The 7-bit LENGTH field codes the number of controller logs minus one; the expression (1 + 2*(LENGTH + 1)) yields the number of bytes in the chapter.

If the S bit is set to 1, the previous packet sent did not include a Control Change command on this channel, and the receiver may skip to the next Chapter if it is recovering from the loss of a single packet.

For each controller log, the 7-bit CONTROLLER field identifies the controller number. For most controllers, the VALUE/COUNT field codes the value of the last Control Change command sent for this controller.

However, if the controller log codes the All Notes Off or All Sound Off controllers, the VALUE/COUNT field codes the total number of Control Change commands received for the lifetime of the session. If this value exceeds 127, modulo arithmetic is used, but the value 0 is skipped.

If the controller log codes the Sustain Pedal controller, zero is used to code pedal release. To code pedal depression, the the VALUE/COUNT field codes the total number of pedal depressions that occur during a session. If this value exceeds 127, modulo arithmetic is used, but the

value 0 is skipped.

If the F bit is set to 1, the previous packet sent did not include a Control Change command for this controller, and the receiver can skip to the next controller log if it is recovering from the loss of a single packet.

If the F bit is 0, the receiver should compare the VALUE/COUNT field with its stored state for the controller.

For the All Notes Off and All Sound Off controllers, and the Sustain Pedal controllers coding a depressed pedal, if the stored modulo count for the controller does not match the VALUE/COUNT field, the receiver should update its state for this controller, and execute the semantics of the lost command. Note these commands happen sufficiently infrequently that the ambiguity of modulo comparisons should not affect the recovery process.

For all other controllers, if the recovery state does not match the VALUE/COUNT field, the receiver should update its recovery state to reflect the value of the VALUE/COUNT. In addition, the receiver MAY execute a single Control Change command, or MAY plan a series of Control Change commands spaced over time.

Appendix B. Author Addresses

John Lazzaro (corresponding author)
UC Berkeley
CS Division
413 Soda Hall
Berkeley CA 94720-1776
Email: lazzaro@cs.berkeley.edu

John Wawrzynek
UC Berkeley
CS Division
631 Soda Hall
Berkeley CA 94720-1776
Email: johnw@cs.berkeley.edu

Appendix C. References

[1] MIDI Manufacturers Association. The complete MIDI 1.0 detailed specification, 1996. http://www.midi.org

[2] International Standards Organization. ISO 14496 MPEG-4,

Part 3 (Audio) Subpart 5 (Structured Audio) 1999.

[3] Sfront source code release, includes a Linux networking client that implements the MIDI RTP packetization. http://www.cs.berkeley.edu/~lazzaro/sa/

[4] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications, 1996.

[5] John Lazzaro and John Wawrzynek. A Case for Network Musical Performance. The 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001) June 25-26, 2001, Port Jefferson, New York. http://www.cs.berkeley.edu/~lazzaro/sa/pubs/pdf/nossdav01.pdf

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

Appendix D. Expiration Notice

This document expires February 21, 2002.