INTERNET-DRAFT
January 23, 2006
Expires: July 23, 2006

J. Lazzaro
J. Wawrzynek
UC Berkeley

An Implementation Guide for RTP MIDI

<draft-ietf-avt-rtp-midi-guidelines-15.txt>

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with Section 6 of BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on July 23, 2006.

Abstract

This memo offers non-normative implementation guidance for the RTP MIDI payload format. The memo presents its advice in the context of a network musical performance application. In this 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 sessions over unicast UDP. Algorithms for sending and receiving recovery journals (the resiliency structure for the payload format) are described in detail. Although the memo focuses on network musical performance, the presented implementation advice is relevant to other RTP MIDI applications.

Table of Contents

1. Introduction

[RTPMIDI] normatively defines a Real Time Protocol (RTP, [RFC3550]) payload format for the MIDI command language [MIDI], for use under any applicable RTP profile (such as the Audio/Visual Profile (AVP, [RFC3551])).

However, [RTPMIDI] does not define algorithms for sending and receiving MIDI streams. Implementors are free to use any sending or receiving algorithm that conforms to the normative text in [RTPMIDI] [RFC3550] [RFC3551] [MIDI].

In this memo, we offer implementation guidance on sending and receiving MIDI RTP streams. Unlike [RTPMIDI], this memo is not normative.

RTP is a mature protocol, and excellent RTP reference materials are available [RTPBOOK]. This memo aims to complement the existing literature, by focusing on issues that are specific to the MIDI payload format.

The memo focuses on one application: two-party network musical performance over wide-area networks, following the interoperability guidelines in Appendix C.7.2 of [RTPMIDI]. Underlying the performances are RTP MIDI sessions over unicast UDP transport. Resiliency is provided by the recovery journal system [RTPMIDI]. The application also uses the RTP control protocol (RTCP, [RFC3550]).

The application targets a network with a particular set of characteristics: low nominal jitter, low packet loss, and occasional outlier packets that arrive very late. However, in Section 6.2 of this memo we discuss adapting the application to other network environments.

As defined in [NMP], a network musical performance occurs when a group of musicians, located at different physical locations, interact over a network to perform as they would if located in the same room.

Sections 2-3 of this memo describe session startup and maintenance. Sections 4-5 cover sending MIDI streams, and Sections 6-7 cover receiving MIDI streams.

2. Starting the Session

In this section, we describe how the application starts a two-player session. We assume that the two parties have agreed on a session configuration, embodied by a pair of Session Description Protocol (SDP, [SDP]) session descriptions.

One session description (Figure 1) defines how the first party wishes to receive its stream. The other session description (Figure 2) defines how the second party wishes to receive its stream.

The session description in Figure 1 codes that the first party intends to receive a MIDI stream on IP4 number 192.0.2.94 (coded in the c= line) at UDP port 16112 (coded in the m= line). Implicit in the SDP m= line syntax [SDP] is that the first party also intends to receive an RTCP stream on 192.0.2.94 at UDP port 16113 (16112 + 1). The receiver expects that the PT field of each RTP header in the received stream will be set to 96 (coded in the m= line).

Likewise, the session description in Figure 2 codes that the second party intends to receive a MIDI stream on IP4 number 192.0.2.105 at UDP port 5004, and intends to receive an RTCP stream on 192.0.2.105 at UDP port 5005 (5004 + 1). The second party expects that the PT RTP header field of received stream will be set to 101.

```
v=0
o=first 2520644554 2838152170 IN IP4 first.example.net
s=Example
t=0 0
c=IN IP4 192.0.2.94
m=audio 16112 RTP/AVP 96
b=AS:20
b=RS:0
b=RR:400
a=rtpmap:96 mpeg4-generic/44100
a=fmtp:96 streamtype=5; mode=rtp-midi; config=""; profile-level-id=12;
cm_unused=ABFGHJKMQTVXYZ; cm_unused=C120-127; ch_never=ADEFMQTVX;
tsmode=buffer; linerate=320000; octpos=last; mperiod=44; rtp_ptime=0;
rtp_maxptime=0; guardtime=44100; render=synthetic; rinit="audio/asc";
url="http://example.net/sa.asc";
cid="xjflsoeiurvpa09itnvlduihgnvet98pa3w9utnuighbuk"
```

(The a=fmtp line has been wrapped to fit the page to accommodate memo formatting restrictions; it comprises a single line in SDP)

Figure 1 -- Session description for first participant.

```
v=0
o=second 2520644554 2838152170 IN IP4 second.example.net
s=Example
t=0 0
c=IN IP4 192.0.2.105
m=audio 5004 RTP/AVP 101
b=AS:20
b=RS:0
b=RR:400
a=rtpmap:101 mpeg4-generic/44100
a=fmtp:101 streamtype=5; mode=rtp-midi; config=""; profile-level-id=12;
cm_unused=ABFGHJKMQTVXYZ; cm_unused=C120-127; ch_never=ADEFMQTVX;
tsmode=buffer; linerate=320000;octpos=last;mperiod=44; guardtime=44100;
rtp_ptime=0; rtp_maxptime=0; render=synthetic; rinit="audio/asc";
url="http://example.net/sa.asc";
cid="xjflsoeiurvpa09itnvlduihgnvet98pa3w9utnuighbuk"

(The a=fmtp line has been wrapped to fit the page to accommodate
 memo formatting restrictions; it comprises a single line in SDP)
```

Figure 2 -- Session description for second participant.

The session descriptions use the mpeg4-generic media type (coded in the a=rtpmap line) to specify the use of the MPEG 4 Structured Audio renderer [MPEGSA]. The session descriptions also use parameters to customize the stream (Appendix C of [RTPMIDI]). The parameter values are identical for both parties, yielding identical rendering environments for the two client hosts.

The bandwidth (b=) AS parameter [SDP] [RFC3550] indicates that the total RTP session bandwidth is 20 kbs. This value assumes that the two players send 10 kbs streams concurrently. To derive the 10 kbs value, we begin with the analysis of RTP MIDI payload bandwidth in Appendix A.4 of [NMP], and add in RTP and IP4 packet overhead and a small safety factor.

The bandwidth RR parameter [RFC3556] indicates that the shared RTCP session bandwidth for the two parties is 400 bps. We set the bandwidth SR parameter to 0 bps, to signal that sending parties and non-sending parties equally share the 400 bps of RTCP bandwidth (note that in this particular example, the guardtime parameter value of 44100 ensures that both parties are sending for the duration of the session). The 400 bps RTCP bandwidth value supports one RTCP packet per 5 seconds from each party, containing a Sender Report and CNAME information [RFC3550].

We now show example code that implements the actions the parties take during the session. The code is written in C, and uses standard network programming techniques described in [STEVENS]. We show code for the first party (the second party takes a symmetric set of actions).

Figure 3 shows how the first party initializes a pair of socket descriptors (rtp_fd and rtcp_fd) to send and receive UDP packets. After the code in Figure 3 runs, the first party may check for new RTP or RTCP packets by calling recv() on rtp_fd or rtcp_fd.

Applications may use recv() to receive UDP packets on a socket using one of two general methods: "blocking" or "non-blocking".

A call to recv() on a blocking UDP socket puts the calling thread to sleep until a new packet arrives.

A call to recv() on a non-blocking socket acts to poll the device: the recv() call returns immediately, with a return value that indicates the polling result. In this case, a positive return value signals the size of a new received packet, and a negative return value (coupled with an errno value of EAGAIN) indicates no new packet was available.

The choice of blocking or non-blocking sockets is a critical application choice. Blocking sockets offer the lowest potential latency (as the OS wakes the caller as soon as a packet has arrived). However, audio applications that use blocking sockets must adopt a multi-threaded program architecture, so that audio samples may be generated on a "rendering thread" while the "network thread" sleeps while awaiting the next packet. The architecture must also support a thread communication mechanism, so that the network thread has a mechanism to send MIDI commands the rendering thread.

In contrast, audio applications that use non-blocking sockets may be coded using a single thread, that alternatives between audio sample generation and network polling. This architecture trades off increased network latency (as a packet may arrive between polls) for a simpler program architecture. For simplicity, our example uses non-blocking sockets and presumes a single run loop. Figure 4 shows how the example configures its sockets to be non-blocking.

Figure 5 shows how to use recv() to check a non-blocking socket for new packets.

The first party also uses rtp_fd and rtcp_fd to send RTP and RTCP packets to the second party. In Figure 6, we show how to initialize socket structures that address the second party. In Figure 7, we show how to use one of these structures in a sendto() call to send an RTP packet to the second party.

Note that the code shown in Figures 3-7 assumes a clear network path between the participants. The code may not work if firewalls or Network Address Translation (NAT) devices are present in the network path.

3. Session Management: Session Housekeeping

After the two-party interactive session is set up, the parties begin to send and receive RTP packets. In Sections 4-7, we discuss RTP MIDI sending and receiving algorithms. In this section, we describe session "housekeeping" tasks that the participants also perform.

One housekeeping task is the maintenance of the 32-bit SSRC value that uniquely identifies each party. Section 8 of [RFC3550] describes SSRC issues in detail, as does Section 2.1 in [RTPMIDI]. Another housekeeping task is the sending and receiving of RTCP. Section 6 of [RFC3550] describes RTCP in detail.

Another housekeeping task concerns security. As detailed in Appendix G of [RTPMIDI], per-packet authentication is strongly recommended for use with MIDI streams, because the acceptance of rogue packets may lead to the execution of arbitrary MIDI commands.

A final housekeeping task concerns the termination of the session. In our two-party example, the session terminates upon the exit of one of the participants. A clean termination may require active effort by a receiver, as a MIDI stream stopped at an arbitrary point may cause stuck notes and other indefinite artifacts in the MIDI renderer.

The exit of a party may be signalled in several ways. Session management tools may offer a reliable signal for termination (such as the SIP BYE method [RFC3261]). The (unreliable) RTCP BYE packet [RFC3550] may also signal the exit of a party. Receivers may also sense the lack of RTCP activity and timeout a party, or may use transport methods to detect an exit.

4. Sending Streams: General Considerations

In this section, we discuss sender implementation issues.

The sender 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 header and recovery journal data.

In both cases, the sender generates a packet that consists of an RTP header, a MIDI command section, and a recovery journal. In the first case, the MIDI list of the MIDI command section codes the new MIDI data. In the second case, the MIDI list is empty.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

  int rtp_fd, rtcp_fd;       /* socket descriptors */
  struct sockaddr_in addr;   /* for bind address   */

  /*********************************/
  /* create the socket descriptors */
  /*********************************/

  if ((rtp_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    ERROR_RETURN("Couldn't create Internet RTP socket");

  if ((rtcp_fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    ERROR_RETURN("Couldn't create Internet RTCP socket");


  /*********************************/
  /* bind the RTP socket descriptor */
  /*********************************/

  memset(&(addr.sin_zero), 0, 8);
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = htonl(INADDR_ANY);
  addr.sin_port = htons(16112); /* port 16112, from SDP */

  if (bind(rtp_fd, (struct sockaddr *)&addr,
        sizeof(struct sockaddr)) < 0)
     ERROR_RETURN("Couldn't bind Internet RTP socket");


  /*********************************/
  /* bind the RTCP socket descriptor */
  /*********************************/

  memset(&(addr.sin_zero), 0, 8);
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = htonl(INADDR_ANY);
  addr.sin_port = htons(16113); /* port 16113, from SDP */

  if (bind(rtcp_fd, (struct sockaddr *)&addr,
        sizeof(struct sockaddr)) < 0)
      ERROR_RETURN("Couldn't bind Internet RTCP socket");
```

Figure 3 -- Setup code for listening for RTP/RTCP packets.

```
#include <unistd.h>
#include <fcntl.h>

  /***************************/
  /* set non-blocking status */
  /***************************/

  if (fcntl(rtp_fd, F_SETFL, O_NONBLOCK))
    ERROR_RETURN("Couldn't unblock Internet RTP socket");

  if (fcntl(rtcp_fd, F_SETFL, O_NONBLOCK))
    ERROR_RETURN("Couldn't unblock Internet RTCP socket");
```

Figure 4 -- Code to set socket descriptors to be non-blocking.

```
#include <errno.h>
#define UDPMAXSIZE 1472     /* based on Ethernet MTU of 1500 */

unsigned char packet[UDPMAXSIZE+1];
int len, normal;

 while ((len = recv(rtp_fd, packet, UDPMAXSIZE + 1, 0)) > 0)
  {
    /*  process packet[].  If (len == UDPMAXSIZE + 1), recv()
     *  may be returning a truncated packet -- process with care
     */
  }

  /* line below sets "normal" to 1 if the recv() return */
  /*   status indicates no packets are left to process  */

 normal = (len < 0) && (errno == EAGAIN);

 if (!normal)
  {
    /*
     *  recv() return status indicates an empty UDP payload
     *  (len == 0) or an error condition (coded by (len < 0)
     *  and (errno != EAGAIN)).  Examine len and errno, and
     *  take appropriate recovery action.
     */
  }
```

Figure 5 -- Code to check rtp_fd for new RTP packets.

```
#include <arpa/inet.h>
#include <netinet/in.h>

struct sockaddr_in * rtp_addr;      /* RTP destination IP/port  */
struct sockaddr_in * rtcp_addr;     /* RTCP destination IP/port */


  /* set RTP address, as coded in Figure 2's SDP */

  rtp_addr = calloc(1, sizeof(struct sockaddr_in));
  rtp_addr->sin_family = AF_INET;
  rtp_addr->sin_port = htons(5004);
  rtp_addr->sin_addr.s_addr = inet_addr("192.0.2.105");

  /* set RTCP address, as coded in Figure 2's SDP */

  rtcp_addr = calloc(1, sizeof(struct sockaddr_in));
  rtcp_addr->sin_family = AF_INET;
  rtcp_addr->sin_port = htons(5005);   /* 5004 + 1 */
  rtcp_addr->sin_addr.s_addr = rtp_addr->sin_addr.s_addr;
```

Figure 6 -- Initializing destination addresses for RTP and RTCP.

```
unsigned char packet[UDPMAXSIZE];  /* RTP packet to send   */
int size;                          /* length of RTP packet */


  /* first fill packet[] and set size ... then: */

  if (sendto(rtp_fd, packet, size, 0, rtp_addr,
          sizeof(struct sockaddr))  == -1)
    {
      /*
       * try again later if errno == EAGAIN or EINTR
       *
       * other errno values --> an operational error
       */
    }
```

Figure 7 -- Using sendto() to send an RTP packet.

Figure 8 shows the 5 steps a sender takes to issue a packet. This algorithm corresponds to the code fragment for sending RTP packets shown in Figure 7 of Section 2. Steps 1, 2, and 3 occur before the sendto() call in the code fragment. Step 4 corresponds to the sendto() call itself. Step 5 may occur once Step 3 completes.

Algorithm for Sending a Packet:

1. Generate the RTP header for the new packet. See Section 2.1 of [RTPMIDI] for details.

2. Generate the MIDI command section for the new packet. See Section 3 of [RTPMIDI] for details.

3. Generate the recovery journal for the new packet. We discuss this process in Section 5.2. The generation algorithm examines the Recovery Journal Sending Structure (RJSS), a stateful coding of a history of the stream.

4. Send the new packet to the receiver.

5. Update the RJSS to include the data coded in the MIDI command section of the packet sent in step 4. We discuss the update procedure in Section 5.3.

Figure 8 -- A 5 step algorithm for sending a packet.

In the sections that follow, we discuss specific sender implementation issues in detail.

4.1 Queuing and Coding Incoming MIDI Data

Simple senders transmit a new packet as soon as the local player generates a complete MIDI command. The system described in [NMP] uses this algorithm. This algorithm minimizes the sender queuing latency, as the sender never delays the transmission of a new MIDI command.

In a relative sense, this algorithm uses bandwidth inefficiently, as it does not amortize the overhead of a packet over several commands. This inefficiency may be acceptable for sparse MIDI streams (see Appendix A.4 of [NMP]). More sophisticated sending algorithms [GRAME] improve efficiency by coding small groups of commands into a single packet, at the expense of increasing the sender queuing latency.

Senders assign a timestamp value to each command issued by the local player (Appendix C.3 of [RTPMIDI]). Senders may code the timestamp value of the first MIDI list command in two ways. The most efficient method is to set the RTP timestamp of the packet to the timestamp value of the first command. In this method, the Z bit of the MIDI command section header (Figure 2 of [RTPMIDI]) is set to 0, and the RTP timestamps increment at a non-uniform rate.

However, in some applications, senders may wish to generate a stream whose RTP timestamps increment at a uniform rate. To do so, senders may use the Delta Time MIDI list field to code a timestamp for the first command in the list. In this case, the Z bit of the MIDI command section header is set to 1.

Senders should strive to maintain a constant relationship between the RTP packet timestamp and the packet sending time: if two packets have RTP timestamps that differ by 1 second, the second packet should be sent **1 second after the first packet.** To the receiver, variance in this relationship is indistinguishable from network jitter. Latency issues are discussed in detail in Section 6.

Senders may alter the running status coding of the first command in the MIDI list, in order to comply with the coding rules defined in Section **3.2 of [RTPMIDI].** The P header bit (Figure 2 of [RTPMIDI]) codes this alteration of the source command stream.

4.2 Sending Packets with Empty MIDI Lists

During a session, musicians might refrain from generating MIDI data for extended periods of time (seconds or even minutes). If an RTP stream followed the dynamics of a silent MIDI source, and stopped sending RTP packets, system behavior might be degraded in the following ways:

- The receiver's model of network performance may fall out of date.
- Network middleboxes (such as Network Address Translators) may "time-out" the silent stream and drop the port and IP association state.
- If the session does not use RTCP, receivers may misinterpret the silent stream as a dropped network connection.

Senders avoid these problems by sending "keep-alive" RTP packets during periods of network inactivity. Keep-alive packets have empty MIDI lists.

Session participants may specify the frequency of keep-alive packets during session configuration with the MIME parameter "guardtime" (Appendix C.4.2 of [RTPMIDI]). The session descriptions shown in Figures 1-2 use guardtime to specify a keep-alive sending interval of 1 second.

Senders may also send empty packets to improve the performance of the recovery journal system. As we describe in Section 6, the recovery process begins when a receiver detects a break in the RTP sequence number pattern of the stream. The receiver uses the recovery journal of the break packet to guide corrective rendering actions, such as ending stuck notes and updating out-of-date controller values.

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). 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.

To handle this situation, senders may transmit empty packets to "guard" the stream during silent sections. The guard packet algorithm defined in Section 7.3 of [NMP], as applied to the situation described above, sends a guard packet after 100 ms of player inactivity, and sends a second guard packet 100 ms later. Subsequent guard packets are sent with an exponential backoff, with a limiting period of 1 second (set by the "guardtime" parameter in Figures 1-2). The algorithm terminates once MIDI activity resumes, or once RTCP receiver reports indicate that the receiver is up to date.

The perceptual quality of guard packet sending algorithms is a quality of implementation issue for RTP MIDI applications. Sophisticated implementations may tailor the guard packet sending rate to the nature of the MIDI commands recently sent in the stream, to minimize the perceptual impact of moderate packet loss.

As an example of this sort of specialization, the guard packet algorithm described in [NMP] protects against the transient artifacts that occur when NoteOn commands are lost. The algorithm sends a guard packet 1 ms after every packet whose MIDI list contains a NoteOn command. The Y bit in Chapter N note logs (Appendix A.6 of [RTPMIDI]) supports this use of guard packets.

Congestion control and bandwidth management are key issues in guard packet algorithms. We discuss these issues in the next section.

4.3 Congestion Control and Bandwidth Management

The congestion control section of [RTPMIDI] discusses the importance of congestion control for RTP MIDI streams, and references the normative text in [RFC3550] and [RFC3551] that concerns congestion control. To comply the requirements described in those normative documents, RTP MIDI senders may use several methods to control the sending rate:

- As described in Section 4.1, senders may pack several MIDI commands into a single packet, thereby reducing stream bandwidth (at the expense of increasing sender queuing latency).

- Guard packet algorithms (Section 4.2) may be designed in a parametric way, so that the tradeoff between artifact reduction and stream bandwidth may be tuned dynamically.

- The recovery journal size may be reduced, by adapting the techniques described in Section 5 of this memo. Note that in all cases, the recovery journal sender must conform to the normative text in Section 4 of [RTPMIDI].

- The incoming MIDI stream may be modified, to reduce the number of MIDI commands without significantly altering the performance. Lossy "MIDI filtering" algorithms are well developed in the MIDI community, and may be directly applied to RTP MIDI rate management.

RTP MIDI senders incorporate these rate control methods into feedback systems to implement congestion control and bandwidth management. Sections 10 and 6.4.4 of [RFC3550] and Section 2 in [RFC3551] describe feedback systems for congestion control in RTP, and Section 6 of [SDP] describes bandwidth management in media sessions.

5. Sending Streams: The Recovery Journal

In this section, we describe how senders implement the recovery journal system. The implementation we describe uses the default "closed-loop" recovery journal semantics (Appendix C.2.2.2 of [RTPMIDI]).

We begin by describing the Recovery Journal Sending Structure (RJSS). Senders use the RJSS to generate the recovery journal section for RTP MIDI packets.

The RJSS is a hierarchical representation of the checkpoint history of the stream. The checkpoint history holds the MIDI commands that are at risk to packet loss (Appendix A.1 of [RTPMIDI] precisely defines the checkpoint history). The layout of the RJSS mirrors the hierarchical

structure of the recovery journal bitfields.

Figure 9 shows a RJSS implementation for a simple sender. The leaf level of the RJSS hierarchy (the jsend_chapter structures) corresponds to channel chapters (Appendices A.2-9 in [RTPMIDI]). The second level of the hierarchy (jsend_channel) corresponds to the channel journal header (Figure 9 in [RTPMIDI]). The top level of the hierarchy (jsend_journal) corresponds to the recovery journal header (Figure 8 in [RTPMIDI]).

Each RJSS data structure may code several items:

1. The current contents of the recovery journal bitfield associated with the RJSS structure (jheader[], cheader[], or a chapter bitfield).

2. A seqnum variable. Seqnum codes the extended RTP sequence number of the most recent packet that added information to the RJSS structure. If the seqnum of a structure is updated, the seqnums of all structures above it in the recovery journal hierarchy are also updated. Thus, a packet that caused an update to a specific jsend_chapter structure would update the seqnum values of this structure and of the jsend_channel and jsend_journal structures that contain it.

3. Ancillary variables used by the sending algorithm.

A seqnum variable is set to zero if the checkpoint history contains no information at the level or at any level below it. This coding scheme assumes that the first sequence number of a stream is normalized to 1, and limits the total number of stream packets to 2^32 - 1.

The cm_unused and ch_never parameters in Figures 1-2 define the subset of MIDI commands supported by the sender (see Appendix C.2.3 of [RTPMIDI] for details). The sender transmits most voice commands, but does not transmit system commands. The sender assumes the MIDI source uses note commands in the typical way, and does not use the Chapter E note resiliency tools (Appendix A.7 of [RTPMIDI]). The sender does not support Control Change commands for controller numbers with All Notes Off (123-127), All Sound Off (120), and Reset All Controllers (121) semantics, and does not support enhanced Chapter C encoding (Appendix A.3.3 of [RTPMIDI]).

We chose this subset of MIDI commands to simplify the example. In particular, the command restrictions ensure that all commands are active, all note commands are N-active, and all Control Change commands are C-active (see Appendix A.1 of [RTPMIDI] for definitions of active, N-active, and C-active).

In the sections that follow, we describe the tasks a sender performs to manage the recovery journal system.

5.1 Initializing the RJSS

At the start of a stream, the sender initializes the RJSS. All seqnum variables are set to zero, including all elements of note_seqnum[] and control_seqnum[].

The sender initializes jheader[] to form a recovery journal header that codes an empty journal. The S bit of the header is set to 1, and the A, Y, R, and TOTCHAN header fields are set to zero. The checkpoint packet sequence number field is set to the sequence number of the upcoming first RTP packet (per Appendix A.1 of [RTPMIDI]).

```
typedef unsigned char  uint8;      /* must be 1 octet  */
typedef unsigned short uint16;     /* must be 2 octet  */
typedef unsigned long  uint32;     /* must be 4 octets */

/*************************************************************/
/* leaf level hierarchy: Chapter W, Appendix A.5 of [RTPMIDI] */
/*************************************************************/

typedef struct jsend_chapterw {  /* Pitch Wheel (0xE) */
 uint8  chapterw[2]; /* bitfield Figure A.5.1 [RTPMIDI] */
 uint32 seqnum;      /* extended sequence number, or 0 */
} jsend_chapterw;

/*************************************************************/
/* leaf level hierarchy: Chapter N, Appendix A.6 of [RTPMIDI] */
/*************************************************************/

typedef struct jsend_chaptern { /* Note commands (0x8, 0x9) */

 /* chapter N maximum size is 274 octets: a 2 octet header, */
 /* and a maximum of 128 2-octet logs and 16 OFFBIT octets  */

 uint8  chaptern[274];     /* bitfield Figure A.6.1 [RTPMIDI] */
 uint16 size;              /* actual size of chaptern[]     */
 uint32 seqnum;            /* extended seq number, or 0     */
 uint32 note_seqnum[128];  /* most recent note seqnum, or 0 */
 uint32 note_tstamp[128];  /* NoteOn execution timestamp    */
 uint32 bitfield_ptr[128]; /* points to a chapter log, or 0 */
} jsend_chaptern;

/*************************************************************/
/* leaf level hierarchy: Chapter C, Appendix A.3 of [RTPMIDI] */
/*************************************************************/

typedef struct jsend_chapterc {     /* Control Change (0xB) */

 /* chapter C maximum size is 257 octets: a 1 octet header */
 /* and a maximum of 128 2-octet logs                      */

 uint8  chapterc[257];    /* bitfield Figure A.3.1 [RTPMIDI] */
 uint16 size;             /* actual size of chapterc[]      */
 uint32 seqnum;           /* extended sequence number, or 0 */
 uint32 control_seqnum[128]; /* most recent seqnum, or 0    */
 uint32 bitfield_ptr[128]; /* points to a chapter log, or 0 */
} jsend_chapterc;
```

Figure 9 -- Recovery Journal Sending Structure (part 1)

```
/************************************************************/
/* leaf level hierarchy: Chapter P, Appendix A.2 of [RTPMIDI] */
/************************************************************/

typedef struct jsend_chapterp { /* MIDI Program Change (0xC) */

 uint8  chapterp[3]; /* bitfield Figure A.2.1 [RTPMIDI] */
 uint32 seqnum;      /* extended sequence number, or 0 */

} jsend_chapterp;

/**************************************************/
/* second-level of hierarchy, for channel journals */
/**************************************************/

typedef struct jsend_channel {

 uint8  cheader[3]; /* header Figure 9 [RTPMIDI]) */
 uint32 seqnum;     /* extended sequence number, or 0  */

 jsend_chapterp chapterp;          /* chapter P info  */
 jsend_chapterc chapterc;          /* chapter C info  */
 jsend_chapterw chapterw;          /* chapter W info  */
 jsend_chaptern chaptern;          /* chapter N info  */

} jsend_channel;

/******************************************************/
/* top level of hierarchy, for recovery journal header */
/******************************************************/

 typedef struct jsend_journal {

 uint8 jheader[3]; /* header Figure 8, [RTPMIDI] */
                   /* Note: Empty journal has a header */

 uint32 seqnum;    /* extended sequence number, or 0   */
                   /* seqnum = 0 codes empty journal   */

 jsend_channel channels[16];  /* channel journal state */
                              /* index is MIDI channel */

 } jsend_journal;
```

Figure 9 (continued) -- Recovery Journal Sending Structure

In jsend_chaptern, elements of note_tstamp[] are set to zero. In jsend_chaptern and jsend_chapterc, elements of bitfield_ptr[] are set to the null pointer index value (bitfield_ptr[] is an array whose elements point to the first octet of the note or control log associated with the array index).

5.2 Traversing the RJSS

Whenever an RTP packet is created (Step 3 in the algorithm defined in Figure 8), the sender traverses the RJSS to create the recovery journal for the packet. The traversal begins at the top level of the RJSS. The sender copies jheader[] into the packet, and then sets the S bit of jheader[] to 1.

The traversal continues depth-first, visiting every jsend_channel whose seqnum variable is non-zero. The sender copies the cheader[] array into the packet, and then sets the S bit of cheader[] to 1. After each cheader[] copy, the sender visits each leaf-level chapter, in order of its appearance in the chapter journal Table of Contents (first P, then C, then W, then N, as shown in Figure 9 of [RTPMIDI]).

If a chapter has a non-zero seqnum, the sender copies the chapter bitfield array into the packet, and then sets the S bit of the RJSS array to 1. For chaptern[], the B bit is also set to 1. For the variable-length chapters (chaptern[] and chapterc[]), the sender checks the size variable to determine the bitfield length.

Before copying chaptern[], the sender updates the Y bit of each note log to code the onset of the associated NoteOn command (Figure A.6.3 in [RTPMIDI]). To determine the Y bit value, the sender checks the note_tstamp[] array for note timing information.

5.3 Updating the RJSS

After an RTP packet is sent, the sender updates the RJSS to refresh the checkpoint history (Step 5 in the sending algorithm defined in Figure 8). For each command in the MIDI list of the sent packet, the sender performs the update procedure we now describe.

The update procedure begins at the leaf level. The sender generates a new bitfield array for the chapter associated with the MIDI command, using the chapter-specific semantics defined in Appendix A of [RTPMIDI].

For Chapter N and Chapter C, the sender uses the bitfield_ptr[] array to locate and update an existing log for a note or controller. If a log does not exist, the sender adds a log to the end of the chaptern[] or chapterc[] bitfield, and changes the bitfield_ptr[] value to point to the log. For Chapter N, the sender also updates note_tstamp[].

The sender also clears the S bit of the chapterp[], chapterw[], or chapterc[] bitfield. For chaptern[], the sender clears the S bit or the B bit of the bitfield, as described in Appendix A.6 of [RTPMIDI].

Next, the sender refreshes the upper levels of the RJSS hierarchy. At the second-level, the sender updates the cheader[] bitfield of the channel associated with the command. The sender sets the S bit of cheader[] to 0. If the new command forced the addition of a new chapter or channel journal, the sender may also update other cheader[] fields. At the top-level, the sender updates the top-level jheader[] bitfield in a similar manner.

Finally, the sender updates the seqnum variables associated with the changed bitfield arrays. The sender sets the seqnum variables to the extended sequence number of the packet.

5.4 Trimming the RJSS

At regular intervals, receivers send RTCP receiver reports to the sender (as described in Section 6.4.2 of [RFC3550]). These reports include the extended highest sequence number received (EHSNR) field. This field codes the highest sequence number that the receiver has observed from the sender, extended to disambiguate sequence number rollover.

When the sender receives an RTCP receiver report, it runs the RJSS trimming algorithm. The trimming algorithm uses the EHSNR to trim away parts of the RJSS, and thus reduce the size of recovery journals sent in subsequent RTP packets. The algorithm conforms to the closed-loop sending policy defined in Appendix C.2.2.2 of [RTPMIDI].

The trimming algorithm relies on the following observation: if the EHSNR indicates that a packet with sequence number K has been received, MIDI commands sent in packets with sequence numbers J <= K may be removed from the RJSS without violating the closed-loop policy.

To begin the trimming algorithm, the sender extracts the EHSNR field from the receiver report, and adjusts the EHSNR to reflect the sequence number extension prefix of the sender. Then, the sender compares the adjusted EHSNR value with seqnum fields at each level of the RJSS, starting at the top level.

Levels whose seqnum is less than or equal to the adjusted EHSNR are trimmed, by setting the seqnum to zero. If necessary, the jheader[] and cheader[] arrays above the trimmed level are adjusted to match the new journal layout. The checkpoint packet sequence number field of jheader[] is updated to match the EHSNR.

At the leaf level, the sender trims the size of the variable-length

chaptern[] and chapterc[] bitfields. The sender loops through the note_seqnum[] or control_seqnum[] array, and removes chaptern[] or chapterc[] logs whose seqnum value is less than or equal to the adjusted EHSNR. The sender sets the associated bitfield_ptr[] to null, and updates the LENGTH field of the associated cheader[] bitfield.

Note that the trimming algorithm does not add information to the checkpoint history. As a consequence, the trimming algorithm does not clear the S bit (and for chaptern[], the B bit) of any recovery journal bitfield. As a second consequence, the trimming algorithm does not set RJSS seqnum variables to the EHSNR value.

5.5 Implementation Notes

For pedagogical purposes, the recovery journal sender we describe has been simplified in several ways. In practice, an implementation would use enhanced versions of the traversing, updating, and trimming algorithms presented in Sections 5.2-4.

6. Receiving Streams: General Considerations

In this section, we discuss receiver implementation issues.

To begin, we imagine that an ideal network carries the RTP stream. Packets are never lost or reordered, and the end-to-end latency is constant. In addition, we assume that all commands coded in the MIDI list of a packet share the same timestamp (an assumption coded by the "rtp_ptime" and "rtp_maxptime" values in Figures 1-2; see Appendix C.4.1 of [RTPMIDI] for details).

Under these conditions, a simple 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 command section of the payload. Commands are executed in order of their appearance 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 MIDI list:

1. If we permit lost and reordered packets to occur in the network, the algorithm may produce unrecoverable rendering artifacts, violating the mandate defined in Section 4 of [RTPMIDI].

2. If we permit the network to exhibit variable latency, the algorithm modulates the network jitter onto rendered MIDI command stream.

3. 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.

In this section, we discuss interactive receiver design techniques under these relaxed assumptions. Section 6.1 describes a receiver design for high-performance Wide Area Networks (WANs), and Section 6.2 discusses design issues for other types of networks.

6.1 The NMP Receiver Design

The Network Musical Performance (NMP) system [NMP] is an interactive performance application that uses an early version of the RTP MIDI payload format. NMP is designed for use between universities within the State of California, using the high-performance CalREN2 network.

In the NMP system, network artifacts may affect how a musician hears the performances of remote players. However, the network does not affect how a musician hears his own performance.

Several aspects of CalREN2 network behavior (as measured in 2001 timeframe, as documented in [NMP]) guided the NMP system design:

- The median symmetric latency (1/2 the round-trip time) of packets sent between network sites is comparable to the acoustic latency between two musicians located in the same room. For example, the latency between Berkeley and Stanford is 2.1 ms, corresponding to an acoustic distance of 2.4 feet (0.72 meters). These campuses are 40 miles (64 km) apart. Preserving the benefits of the underlying network latency at the application level was a key NMP design goal.

- For most times of day, the nominal temporal jitter is quite short. For Berkeley-Stanford, the standard deviation of the round-trip time was under 200 microseconds.

- For most times of day, a few percent (0-4%) of the packets

sent arrive significantly late (> 40 ms), probably due to a queuing transient somewhere in the network path. More rarely (< 0.1%), a packet is lost during the transient.

- At predictable times during the day (before lunchtime, at the end of the workday, etc), network performance deteriorates (10-20% late packets) in a manner that makes the network unsuitable for low-latency interactive use.

- CalREN2 has deeply over-provisioned bandwidth, relative to MIDI bandwidth usage.

The NMP sender freely uses network bandwidth to improve the performance experience. As soon as a musician generates a MIDI command, an RTP packet coding the command is sent to the other players. This sending algorithm reduces latency at the cost of bandwidth. In addition, guard packets (described in Section 4.2) are sent at frequent intervals, to minimize the impact of packet loss.

The NMP receiver maintains a model of the stream, and uses this model as the basis of its resiliency system. Upon the receipt of a packet, the receiver predicts the RTP sequence number and the RTP timestamp (with error bars) of the packet. Under normal network conditions, about 95% of received packets fit the predictions [NMP]. In this common case, the receiver immediately executes the MIDI command coded in the packet.

Note that the NMP receiver does not use a playout buffer; the design is optimized for lowest latency at the expense of command jitter. Thus, the NMP receiver design does not completely satisfy the interoperability text in Appendix C.7.2 of [RTPMIDI], which requires that receivers in network musical performance applications be capable of using a playout buffer.

Occasionally, an incoming packet fits the sequence number prediction, but falls outside the timestamp prediction error bars (see Appendix B of [NMP] for timestamp model details). In most cases, the receiver still executes the command coded in the packet. However, the receiver discards NoteOn commands with non-zero velocity. By discarding late commands that sound notes, the receiver prevents "straggler notes" from disturbing a performance. By executing all other late commands, the receiver quiets "soft stuck notes" immediately, and updates the state of the MIDI system.

More rarely, an incoming packet does not fit the sequence number prediction. 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. If the sequence number of the received packet is less than the prediction, the packet has been received out of order. All sequence number calculations are modulo 2^16, and use standard methods (described in [RFC3550]) to avoid tracking errors during rollover.

If a packet loss has occurred, the receiver examines the journal section of the received packet, and uses it to gracefully recover from the loss episode. We describe this recovery procedure in Section 7 of this memo. The recovery process may result in the execution of one or more MIDI commands. After executing the recovery commands, the receiver processes the MIDI command encoded in the packet, using the timestamp model test described above.

If a packet is received out of order, the receiver ignores the packet. The receiver takes this action because a packet received out of order is always preceded by a packet that signalled a loss event. This loss event triggered the recovery process, which may have executed recovery commands. The MIDI command coded in the out-of-order packet might, if executed, duplicate these recovery commands, and this duplication might endanger the integrity of the stream. Thus, ignoring the out-of-order packet is the safe approach.

6.2 High-Jitter Networks, Local Area Networks

The NMP receiver targets a network with a particular set of characteristics: low nominal jitter, low packet loss, and occasional outlier packets that arrive very late. In this section, we consider how networks with different characteristics impact receiver design.

Networks with significant nominal jitter cannot use the buffer-free receiver design described in Section 6.1. For example, the NMP system performs poorly for musicians that use dial-up modem connections, because the buffer-free receiver design modulates modem jitter onto the performances. Receivers designed for high-jitter networks should use a substantial playout buffer. References [GRAME] and [CCRMA] describe how to use playout buffers in latency-critical applications.

Receivers intended for use on Local Area Networks (LANs) face a different set of issues. A dedicated LAN fabric built with modern hardware is in many ways a predictable environment. The network problems addressed by the NMP receiver design (packet loss and outlier late packets) might only occur under extreme network overload conditions.

Systems designed for this environment may choose to configure streams without the recovery journal system (Appendix C.2.1 of [RTPMIDI]). Receivers may also wish to forego, or simplify, the detection of outlier late packets. Receivers should monitor the RTP sequence numbers of

incoming packets, to detect network unreliability.

However, in some respects, LAN applications may be more demanding than WAN applications. In LAN applications, musicians may be receiving performance feedback from audio that is rendered from the stream. The tolerance a musician has for latency and jitter in this context may be quite low.

To reduce the perceived jitter, receivers may use a small playout buffer (in the range of 100us to 2ms). The buffer does add a small amount of latency to the system, that may be annoying to some players. Receiver designs should include buffer tuning parameters, to let musicians adjust the tradeoff between latency and jitter.

7. Receiving Streams: The Recovery Journal

In this section, we describe the recovery algorithm used by the NMP receiver [NMP]. In most ways, the recovery techniques we describe are generally applicable to interactive receiver design. However, a few aspects of the design are specialized for the NMP system:

- The recovery algorithm covers a subset of the MIDI command set. MIDI Systems (0xF), Poly Aftertouch (0xA), and Channel Aftertouch (0xD) commands are not protected, and Control Change (0xB) commands protection is simplified. Note commands for a particular note number are assumed to follow the typical NoteOn->NoteOff->NoteOn->NoteOff pattern. The cm_unused and ch_never parameters in Figures 1-2 specify this coverage.

- The NMP system does not use a playout buffer. Therefore, the recovery algorithm does not address interactions with a playout buffer.

At a high level, the receiver algorithm works as follows. Upon the detection of a packet loss, the receiver examines the recovery journal of the packet that ends the loss event. If necessary, the receiver executes one or more MIDI commands to recover from the loss.

To prepare for recovery, a receiver maintains a data structure, the Recovery Journal Receiver Structure (RJRS). 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.

We now describe the recovery algorithm in detail. We begin with two definitions that classify loss events. These definitions assume that the packet that ends the loss event has RTP sequence number I.

- Single-packet loss. A single-packet loss occurs if the last packet received before the loss event (excluding out-of-order packets) has the sequence number I-2 (modulo 2^16).

- Multi-packet loss. A multi-packet loss occurs if the last packet received before the loss event (excluding out-of-order packets) has a sequence number less than I-2 (modulo 2^16).

Upon the detection of a packet loss, the recovery algorithm examines the recovery journal header (Figure 8 of [RTPMIDI]) to check for special cases:

- If the header field A is 0, the recovery journal has no channel journals, and so no action is taken.

- If a single-packet loss has occurred, and the header S bit is 1, the lost packet has a MIDI command section with an empty MIDI list. No action is taken.

If these checks fail, the algorithm parses the recovery journal body. For each channel journal (Figure 9 in [RTPMIDI]) in the recovery journal, the receiver compares the data in each chapter journal (Appendix A of [RTPMIDI]) 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 this loss, and updates the RJRS to reflect the repair.

For single-packet losses, the receiver skips channel and chapter journals whose S bits are set to 1. For multi-packet losses, the receiver parses each channel and chapter journal and checks for inconsistency.

In the sections that follow, we describe the recovery steps that are specific to each chapter journal. We cover 4 chapter journal types: P (Program Change, 0xC), C (Control Change, 0xB), W (Pitch Wheel, 0xE), and N (Note, 0x8 and 0x9). Chapters are parsed in the order of their appearance in the channel journal (P, then W, then N, then C).

The sections below reference the C implementation of the RJRS shown in Figure 10. This structure is hierarchical, reflecting the recovery journal architecture. At the leaf level, specialized data structures (jrec_chapterw, jrec_chaptern, jrec_chapterc, and jrec_chapterp) code state variables for a single chapter journal type. A mid-level

structure (jrec_channel) represents a single MIDI channel, and a top-level structure (jrec_stream) represents the entire MIDI stream.

```
typedef unsigned char  uint8;       /* must be 1 octet  */
typedef unsigned short uint16;      /* must be 2 octets */
typedef unsigned long  uint32;      /* must be 4 octets */


/*****************************************************************/
/* leaf level of hierarchy: Chapter W, Appendix A.5 of [RTPMIDI] */
/*****************************************************************/

typedef struct jrec_chapterw {   /* MIDI Pitch Wheel (0xE) */

 uint16 val;           /* most recent 14-bit wheel value   */

} jrec_chapterw;


/*****************************************************************/
/* leaf level of hierarchy: Chapter N, Appendix A.6 of [RTPMIDI] */
/*****************************************************************/

typedef struct jrec_chaptern { /* Note commands (0x8, 0x9) */

 /* arrays of length 128 --> one for each MIDI Note number */

 uint32 time[128];    /* exec time of most recent NoteOn */
 uint32 extseq[128];  /* extended seqnum for that NoteOn */
 uint8  vel[128];     /* NoteOn velocity (0 for NoteOff) */

} jrec_chaptern;


/*****************************************************************/
/* leaf level of hierarchy: Chapter C, Appendix A.3 of [RTPMIDI] */
/*****************************************************************/

typedef struct jrec_chapterc {     /* Control Change (0xB) */

 /* array of length 128 --> one for each controller number */

 uint8 value[128];   /* Chapter C value tool state */
 uint8 count[128];   /* Chapter C count tool state */
 uint8 toggle[128];  /* Chapter C toggle tool state */

} jrec_chapterc;
```

Figure 10 -- Recovery Journal Receiving Structure (part 1)

```
/*****************************************************************/
/* leaf level of hierarchy: Chapter P, Appendix A.2 of [RTPMIDI] */
/*****************************************************************/

typedef struct jrec_chapterp { /* MIDI Program Change (0xC) */

 uint8 prognum;       /* most recent 7-bit program value  */
 uint8 prognum_qual;  /* 1 once first 0xC command arrives */

 uint8 bank_msb;     /* most recent Bank Select MSB value */
 uint8 bank_msb_qual;   /* 1 once first 0xBn 0x00 arrives */

 uint8 bank_lsb;     /* most recent Bank Select LSB value */
 uint8 bank_lsb_qual;   /* 1 once first 0xBn 0x20 arrives */

} jrec_chapterp;



/************************************************/
/* second-level of hierarchy, for MIDI channels    */
/************************************************/

typedef struct jrec_channel {

 jrec_chapterp chapterp;  /* Program Change (0xC) info  */
 jrec_chapterc chapterc;  /* Control Change (0xB) info  */
 jrec_chapterw chapterw;  /* Pitch Wheel (0xE) info  */
 jrec_chaptern chaptern;  /* Note (0x8, 0x9) info  */

} jrec_channel;



/*********************************************/
/* top level of hierarchy, for the MIDI stream */
/*********************************************/

 typedef struct jrec_stream {

 jrec_channel channels[16];  /* index is MIDI channel */

 } jrec_stream;
```

Figure 10 (continued) -- Recovery Journal Receiving Structure

7.1 Chapter W: MIDI Pitch Wheel (0xE)

Chapter W of the recovery journal protects against the loss of MIDI Pitch Wheel (0xE) commands. A common use of the Pitch Wheel command is to transmit the current position of a rotary "pitch wheel" controller placed on the side of MIDI piano controllers. Players use the pitch wheel to dynamically alter the pitch of all depressed keys.

The NMP receiver maintains the jrec_chapterw structure (Figure 10) for each voice channel in jrec_stream, to code pitch wheel state information. In jrec_chapterw, val holds the 14-bit data value of the most recent Pitch Wheel command that has arrived on a channel. At the start of the stream, val is initialized to the default pitch wheel value (0x2000).

At the end of a loss event, a receiver may find a Chapter W (Appendix A.5 in [RTPMIDI]) bitfield in a channel journal. This chapter codes the 14-bit data value of the most recent MIDI Pitch Wheel command in the checkpoint history. If the Chapter W and jrec_chapterw pitch wheel values do not match, one or more commands have been lost.

To recover from this loss, the NMP 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 jrec_chapterw variables to reflect the executed command.

7.2 Chapter N: MIDI NoteOn (0x8) and NoteOff (0x9)

Chapter N of the recovery journal protects against the loss of MIDI NoteOn (0x9) and NoteOff (0x8) commands. If a NoteOn command is lost, a note is skipped. If a NoteOff command is lost, a note may sound indefinitely. Recall that NoteOn commands with a velocity value of 0 have the semantics of NoteOff commands.

The recovery algorithms in this section only work for MIDI sources that produce NoteOn->NoteOff->NoteOn->NoteOff patterns for a note number. Piano keyboard and drum pad controllers produce these patterns. MIDI sources that use NoteOn->NoteOn->NoteOff->NoteOff patterns for legato repeated notes, such as guitar and wind controllers, require more sophisticated recovery strategies. Chapter E (not used in this example) supports recovery algorithms for atypical note command patterns (see Appendix A.7 of [RTPMIDI] for details).

The NMP receiver maintains a jrec_chaptern structure (Figure 10) for each voice channel in jrec_stream, to code note-related state information. State is kept for each of the 128 note numbers on a channel, using three arrays of length 128 (vel[], seq[], and time[]). The arrays are initialized to zero at the start of a stream.

The vel[n] array element holds information about the most recent note command for note number n. If this command is a NoteOn command, vel[n] holds the velocity data for the command. If this command is a NoteOff command, vel[n] is set to 0.

The time[n] and extseq[n] array elements code information about the most recently executed NoteOn command. The time[n] element holds the execution time of the command, referenced to the local timebase of the receiver. The extseq[n] element holds the RTP extended sequence number of the packet associated with the command. For incoming stream commands, extseq[n] codes the packet of the associated MIDI list. For commands executed to perform loss recovery, extseq[n] codes the packet of the associated recovery journal.

The Chapter N recovery journal bitfield (Figure A.6.1 in [RTPMIDI]) consists of two data structures: a bit array coding recently-sent NoteOff commands that are vulnerable to packet loss, and a note log list coding recently-sent NoteOn commands that are vulnerable to packet loss.

At the end of a loss event, Chapter N recovery processing begins with the NoteOff bit array. For each set bit in the array, the receiver checks the corresponding vel[n] element in jrec_chaptern. If vel[n] is non-zero, a NoteOff command, or a NoteOff->NoteOn->NoteOff command sequence, has been lost. To recover from this loss, the receiver immediately executes a NoteOff command for the note number on the channel, and sets vel[n] to 0.

The receiver then parses the note log list, using the S bit to skip over "safe" logs in the single-packet loss case. For each at-risk note log, the receiver checks the corresponding vel[n] element.

If vel[n] is zero, a NoteOn command, or a NoteOn->NoteOff->NoteOn command sequence, has been lost. The receiver may execute the most recent lost NoteOn (to play the note) or may take no action (to skip the note), based on criteria we describe at the end of this section. Whether the note is played or skipped, the receiver updates the vel[n], time[n], and extseq[n] elements as if the NoteOn executed.

If vel[n] is non-zero, the receiver performs several checks to test if a NoteOff->NoteOn sequence has been lost.

- If vel[n] does not match the note log velocity, the note log must code a different NoteOn command, and thus a NoteOff->NoteOn sequence has been lost.

- If extseq[n] is less than the (extended) checkpoint packet sequence numbed coded in the recovery journal header (Figure 8 of [RTPMIDI]), the vel[n] NoteOn command is not in the checkpoint

history, and thus a NoteOff->NoteOn sequence has been lost.

- If the Y bit is set to 1, the NoteOn is musically "simultaneous" with the RTP timestamp of the packet. If time[n] codes a time value that is clearly not recent, a NoteOff->NoteOn sequence has been lost.

If these tests indicate a lost NoteOff->NoteOn sequence, the receiver immediately executes a NoteOff command. The receiver decides if the most graceful action is to play or to skip the lost NoteOn, using the criteria we describe at the end of this section. Whether or not the receiver issues a NoteOn command, the vel[n], time[n], and extseq[n] arrays are updated as if it did.

Note that the tests above do not catch all lost NoteOff->NoteOn commands. If a fast NoteOn->NoteOff->NoteOn sequence occurs on a note number, with identical velocity values for both NoteOn commands, a lost NoteOff->NoteOn does not result in the recovery algorithm generating a NoteOff command. Instead, the first NoteOn continues to sound, to be terminated by the future NoteOff command. In practice, this (rare) outcome is not musically objectionable.

The number of tests in this resiliency algorithm may seem excessive. However, in some common cases, a subset of the tests are not useful. For example, MIDI streams that assigns the same velocity value to all note events are often produced by inexpensive keyboards. The vel[n] tests are not useful for these streams.

Finally, we discuss how the receiver decides whether to play or to skip a lost NoteOn command. The note log Y bit is set if the NoteOn is "simultaneous" with the RTP timestamp of the packet holding the note log. If Y is 0, the receiver does not execute a NoteOn command. If Y is 1, and if the packet has not arrived late, the receiver immediately executes a NoteOn command for the note number, using the velocity coded in the note log.

7.3 Chapter C: MIDI Control Change (0xB)

Chapter C (Appendix A.3 in [RTPMIDI]) 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.

Chapter C offers three tools for protecting a Control Change command: the value tool (for graded controllers such as sliders) the toggle tool (for on/off switches) and the count tool (for momentary-contact switches). Senders choose a tool to encode recovery information for a controller, and encode the tool type along with the data in the journal (Figures A.3.2 and A.3.3 in [RTPMIDI]).

A few uses of Control Change commands are not solely protected by Chapter C. The protection of controllers 0 and 32 (Bank Select MSB and Bank Select LSB) is shared between Chapter C and Chapter P (Section 7.4).

Chapter M (Appendix A.4 of [RTPMIDI]) also protects the Control Change command. However, the NMP system does not use this chapter, because MPEG 4 Structured Audio [MPEGSA] does not use the controllers protected by this chapter.

The Chapter C bitfield consists of a list of controller logs. Each log codes the controller number, the tool type, and the state value for the tool.

The NMP receiver maintains the jrec_chapterc structure (Figure 10) for each voice channel in jrec_stream, to code Control Change state information. The value[] array holds the most recent data values for each controller number. At the start of the stream, value[] is initialized to the default controller data values specified in [MPEGSA].

The count[] and toggle[] arrays hold the count tool and toggle tool state values. At the start of a stream, these arrays are initialized to zero. Whenever a Control Command executes, the receiver updates the count[] and toggle[] state values, using the algorithms defined in **Appendix A.3 of [RTPMIDI].**

At the end of a loss event, the receiver parses the Chapter C controller log list, using the S bit to skip over "safe" logs in the single-packet loss case. For each at-risk controller number n, the receiver determines the tool type in use (value, toggle, or count), and compares the data in the log to the associated jrec_chapterc array element (value[n], toggle[n], or count[n]). If the data do not match, one or more Control Change commands have been lost.

The method the receiver uses to recover from this loss depends on the tool type and the controller number. For graded controllers protected by the value tool, the receiver executes a Control Change command using the new data value.

For the toggle and count tools, the recovery action is more complex. For example, the Damper Pedal (Sustain) controller (number 64) is typically used as a sustain pedal for piano-like sounds, and is typically coded using the toggle tool. If Damper Pedal (Sustain) Control Change command(s) are lost, the receiver takes different actions depending on the starting and ending state of the lost sequence, to ensure "ringing" piano notes are "damped" to silence.

After recovering from the loss, the receiver updates the value[], toggle[], and count[] arrays to reflect the Chapter C data and the executed commands.

7.4 Chapter P: MIDI Program Change (0xC)

Chapter P of the recovery journal protects against the loss of MIDI Program Change (0xC) commands.

The 7-bit data value of the Program Change command selects one of 128 possible timbres for the channel. To increase the number of possible timbres, Control Change (0xB) commands may be issued prior to the Program Change command, to select a "program bank". The Bank Select MSB (number 0) and Bank Select LSB (number 32) controllers specify the 14-bit bank number that subsequent Program Change commands reference.

The NMP receiver maintains the jrec_chapterp structure (Figure 10) for each voice channel in jrec_stream, to code Program Change state information.

The prognum variable of jrec_chapterp holds the data value for the most recent Program Change command that has arrived on the stream. The bank_msb and bank_lsb variables of jrec_chapterp code the Bank Select MSB and Bank Select LSB controller data values that were in effect when that Program Change command arrived. The prognum_qual, bank_msb_qual and bank_lsb_qual variables are initialized to 0, and are set to 1 to qualify the associated data values.

Chapter P fields code the data value for the most recent Program Change command, and the MSB and LSB bank values in effect for that command.

At the end of a loss event, the receiver checks Chapter P to see if the recovery journal fields match the data stored in jrec_chapterp. If these checks fail, one or more Program Change commands have been lost.

To recover from this loss, the receiver takes the following steps. If the B bit in Chapter P is set (Figure A.2.1 in [RTPMIDI]), Control Change bank command(s) have preceded the Program Change command. The receiver compares the bank data coded by Chapter P with the current bank data for the channel (coded in jrec_channelc).

If the bank data do not agree, the receiver issues Control Change command(s) to align the stream with Chapter P. The receiver then updates jrec_channelp and jrec_channelc variables to reflect the executed command(s). Finally, the receiver issues a Program Change command that reflects the data in Chapter P, and updates the prognum and qual_prognum fields in jrec_channelp.

Note that this method relies on Chapter P recovery to precede Chapter C recovery during channel journal processing. This ordering ensures that lost Bank Select Control Change commands that occur after a lost Program Change command in a stream are handled correctly.

8. Security Considerations

Security considerations for the RTP MIDI payload format are discussed in the Security Considerations section of [RTPMIDI].

9. IANA Considerations

IANA considerations for the RTP MIDI payload format are discussed in the IANA Considerations section of [RTPMIDI].

A. Acknowledgments

This memo was written in conjunction with [RTPMIDI], and the Acknowledgments section of [RTPMIDI] also applies to this memo.

B. References

B.1 Normative References

[RTPMIDI] Lazzaro, J., and J. Wawrzynek. "RTP Payload Format for MIDI", work in progress, draft-ietf-avt-rtp-midi-format-15.txt.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson. "RTP: A transport protocol for real-time applications", RFC 3550, July 2003.

[RFC3551] Schulzrinne, H., and S. Casner. "RTP Profile for Audio and Video Conferences with Minimal Control", RFC 3551, July 2003.

[MIDI] MIDI Manufacturers Association. "The Complete MIDI 1.0 Detailed Specification", 1996.

[SDP] Handley, M., Jacobson, V., and C. Perkins. "SDP: Session Description Protocol", draft-ietf-mmusic-sdp-new-25.txt.

[MPEGSA] International Standards Organization. "ISO/IEC 14496 MPEG-4", Part 3 (Audio), Subpart 5 (Structured Audio), 2001.

[RFC3556] S. Casner. "Session Description Protocol (SDP) Bandwidth

Modifiers for RTP Control Protocol (RTCP) Bandwidth", RFC 3556, July 2003.

B.2 Informative References

[NMP] Lazzaro, J. and J. Wawrzynek. "A Case for Network Musical Performance", 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001) June 25-26, 2001, Port Jefferson, New York.

[RFC3261] Rosenberg, J, Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler. "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[GRAME] Fober, D., Orlarey, Y. and S. Letz. "Real Time Musical Events Streaming over Internet", Proceedings of the International Conference on WEB Delivering of Music 2001, pages 147-154.

[CCRMA] Chafe C., Wilson S., Leistikow R., Chisholm D., and G. Scavone. "A simplified approach to high quality music and sound over IP", COST-G6 Conference on Digital Audio Effects (DAFx-00), Verona, Italy, December 2000.

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

[STEVENS] Stevens, R. W, Fenner, B., and A. Rudoff. "Unix Network Programming: The Sockets Networking API", Addison-Wesley, 2003.

C. Authors' Addresses

John Lazzaro (corresponding author)
UC Berkeley
CS Division
315 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

D. Intellectual Property Rights Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

E. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

F. Change Log

[Note to RFC Editors: this Appendix, and its Table of Contents listing, should be removed from the final version of the memo]

The following changes were made to the document:

--

[1] In the Introduction, after the the paragraph that begins "The memo focuses on one application" we add the paragraph:

The application targets a network with a particular set of characteristics: low nominal jitter, low packet loss, and occasional outlier packets that arrive very late. However, in Section 6.2 of this memo we discuss adapting the application to other network environments.

--

[2] In Section 2, the phrase "uses the sockets API and other POSIX systems calls" was replaced by "uses standard network programming techniques described in [STEVENS]", where [STEVENS] is the informative reference:

[STEVENS] Stevens, R. W, Fenner, B., and A. Rudoff. "Unix Network Programming: The Sockets Networking API", Addison-Wesley, 2003.

--

[3] RTCP is now defined as "RTP control protocol", not "Real Time Control Protocol.

--

[4] The [RTPMIDI] reference was updated to -15.txt.