An Implementation Guide to the MIDI Wire Protocol Packetization (MWPP)

Status of this Memo

This document is an Internet-Draft and is subject to all provisions of Section 10 of RFC2026.

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.html

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

Abstract

This memo offers non-normative implementation guidance for the MIDI Wire Protocol Packetization (MWPP), an RTP packetization for the MIDI command language. The memo provides a detailed description of a sample MWPP application: an interactive MIDI session between two parties that send and receive RTP and RTCP flows over unicast UDP transport. The Appendices focus on special issues that arise in other types of applications: content-streaming applications, multi-party applications, applications that use reliable transport such as TCP, applications that do not use RTCP, and applications that send several MWPP RTP streams in a single session.
# Table of Contents

1. Introduction ................................................. 3
2. Session Management: Starting MWPP Sessions ................. 4
3. Session Management: Session Housekeeping .................. 10
4. Sending MWPP Streams: General Considerations .............. 11
   4.1 Queuing and Coding Incoming MIDI Data .................. 12
   4.2 Sending MWPP Packets with Empty MIDI Lists .............. 14
   4.3 Bandwidth Management and Congestion Control ............ 15
5. Sending MWPP Streams: The Recovery Journal ................ 17
   5.1 Initializing the Sending Structure ...................... 19
   5.2 Traversing the Sending Structure ....................... 19
   5.3 Updating the Sending Structure ......................... 19
   5.4 Trimming the Sending Structure ......................... 20
   5.5 Implementation Notes ................................... 20
6. Receiving MWPP Streams: General Considerations ............. 22
7. Receiving MWPP Streams: The Recovery Journal ............... 22
8. Congestion Control ........................................... 22
9. Security Considerations ...................................... 22
Appendix A. Content Streaming with MWPP ....................... 22
Appendix B. Multi-party MWPP Sessions ......................... 22
Appendix C. MWPP and Reliable Transport ....................... 22
Appendix D. Using MWPP without RTCP .......................... 23
Appendix E. Multi-stream MWPP Sessions ....................... 23
Appendix F. Author Addresses .................................. 23
Appendix G. References ......................................... 24
1. Introduction

The MIDI Wire Protocol Packetization (MWPP, [1]) is a general-purpose RTP/AVP [2,3] packetization for the MIDI [4] command language. [1] normatively defines the MWPP RTP bitfield syntax, and defines the Session Description Protocol (SDP, [5]) parameters that may be used to customize MWPP session behavior. In addition, [1] motivates the development of MWPP, by describing the application space for the packetization.

However, [1] does not normatively define the algorithms for sending and receiving MWPP streams. Implementors are free to use any sending or receiving algorithm that conforms to the MWPP bitfield definitions. A consequence of this protocol definition method is that [1] offers little guidance on how to actually implement MWPP in applications.

In this memo, we address this deficiency, and offer advice on how to implement MWPP. Note that this memo is informative: it does not normatively specify any new MWPP behaviors, but rather offers practical guidance for creating applications that use MWPP.

The application space for MWPP is diverse, and each class of application has unique implementation issues. The taxonomy of MWPP applications includes these dimensions:

- Interactive or streaming. Interactive applications (such as the remote operation of musical instruments) require low end-to-end latency, preferably near the underlying network latency. Streaming applications (such as the incremental delivery of MIDI files) may trade off higher latency for robustness.

- Two-party or multi-party. Two-party MWPP applications have two session participants; multi-party MWPP applications have an arbitrary number of participants. Multi-party applications map efficiently to multicast transport, but may also be mapped onto multiple unicast flows.

- Transport. MWPP streams may use unreliable transport (such as unicast or multicast UDP) or reliable transport (such as TCP).

- Single-stream or multi-stream. Simple MWPP sessions consist of a single RTP stream to convey a single MIDI name space (16 voice channels + systems). Multi-stream sessions use several RTP streams to code a larger number of MIDI channels, or to split a MIDI name space across different physical streams.
The RTP standard [2] defines a backchannel protocol, the RTP Control Protocol (RTCP). MWPP RTP streams work best if paired with an RTCP stream, but MWPP may be used without RTCP.

In the main body of this memo, we discuss implementation issues for MWPP systems that lie at one point in this taxonomy: an interactive, two-party, single-stream session over unicast UDP transport that uses RTCP. Sections 2 and 3 cover session management; Sections 4 and 5 cover sending MWPP streams; Sections 6 and 7 cover receiving MWPP streams. This example is based on the network musical performance system described in [6].

In the Appendices of this memo, we discuss implementation issues for other operating points in this taxonomy. For example, Appendix A describes implementation issues in content streaming. Each Appendix covers all facets of a session: session management, sender design, and receiver design.

This memo is limited in scope, in that it assumes that all session participants have access to the SDP session description(s) that describe the session. The distribution of session descriptions, perhaps via protocols such as the Session Initiation Protocol (SIP, [7]) or the Real Time Streaming Protocol (RTSP, [8]), is not discussed. We anticipate that other memos will define frameworks for session description distribution for MWPP in different application domains, and that these memos will include implementation guidance.

2. Session Management: Starting MWPP Sessions

In this section, we discuss how implementations start MWPP sessions. As an example, we consider an interactive, two-party, single-stream session over unicast UDP transport that uses RTCP. In the Appendices, we discuss session startup issues that are unique to other session variants (such as multi-party sessions or sessions that do not use RTCP).

We assume that the two session participants have agreed on the session configuration, embodied by a pair of Session Description Protocol (SDP, [5]) session descriptions. The first session description defines the IP number and UDP port numbers to which the first party intends to send its RTP and RTCP streams. The second session description defines the IP number and UDP port numbers to which the second party intends to send its RTP and RTCP streams.
Note that even if one participant does not intend to send an MWPP RTP stream (an intention coded by the SDP attribute recvonly [5]) two session descriptions are necessary in the two-party unicast UDP case, because both parties send RTCP backchannel streams. Two session descriptions are required to specify the bidirectional transport configuration for RTCP.

In Figures 1 and 2, we show the complete session descriptions that define our example session.

```
v=0
o=first 2520644554 2838152170 IN IP4 first.berkeley.edu
s=Example
t=3238012065 0
m=audio 5004 RTP/AVP 101
c=IN IP4 169.229.60.105
a=rtpmap: 101 mwpp/44100

Figure 1 -- Session description for first participant.
```

```
v=0
o=second 2520644554 2838152170 IN IP4 second.berkeley.edu
s=Example
t=3238012065 0
m=audio 16112 RTP/AVP 96
c=IN IP4 169.229.60.94
a=rtpmap: 96 mwpp/44100

Figure 2 -- Session description for second participant.
```
The session description in Figure 1 codes that the first party intends to send an MWPP RTP stream to IP4 number 169.229.60.105 (coded in the c= line) at UDP port 5004 (coded in the m= line). Implicit in the SDP m= line syntax [5] is that the first party also intends to send an RTCP backchannel stream to 69.229.60.105 at UDP port 5005 (5004 + 1). The PTYPE field of each RTP header will be set to 101 (coded in both the m= and a= lines).

Likewise, the session description in Figure 2 codes that the second party intends to send an MWPP RTP stream to IP4 number 169.229.60.94 at UDP port 16112, and also intends to send an RTCP backchannel stream to 169.229.60.94 at UDP port 16113 (16112 + 1). The PTYPE field of each RTP header will be set to 96.

We now show the actions the first participant takes to start the session, using the UNIX sockets API. The first party listens to ports 16112 and 16113 on the IP4 network connection for 169.229.60.94. Assuming a single-homed machine, Figure 3 shows the C code fragment to set up this behavior. The (undefined) ERROR_RETURN macro in this code is used to flag fatal setup errors.

After the setup code in Figure 3 runs, the first party can check for the arrival of new RTP or RTCP packets, by using the UNIX system call recv() on the rtp_fd or rtcp_fd socket descriptors.

By default, a recv() on these socket descriptors will block until a packet arrives. Figure 4 shows a code fragment to configure these sockets to be non-blocking, so that recv() calls may be done in time-critical code, without fear of I/O blocking. Figure 5 shows how to use recv() to check a non-blocking socket for new packets.

The first party may also use the rtp_fd and rtcp_fd socket descriptors to send RTP and RTCP packets to the second party. In Figure 6, we show how to prepare socket addresses that correspond to the transport information coded in the session description shown in Figure 1. In Figure 7, we show how to use sendto() call to send an RTP packet to the prepared RTP socket address.

The first party should prepare to render the incoming MIDI stream. In some applications, the session description of the second party may use the SDP parameter render (Appendix A.5 of [1]) to code the rendering method the first party should use to process the incoming stream.

Note that the setup code shown in Figures 3-7 assume a clear network path between the participants. If firewalls or Network Address Translation (NAT) devices are present in the network path, the code shown may not work. Standardized methods for using RTP in NAT and firewall environments are under development [10].
#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>

int one = 1;

/******************************************************
/* set non-blocking status, shield spurious ICMP errno */
/*******************************************************/

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");

if (setsockopt(rtp_fd, SOL_SOCKET, SO_BSDCOMPAT, &one, sizeof(one)))
    ERROR_RETURN("Couldn’t shield RTP socket");

if (setsockopt(rtcp_fd, SOL_SOCKET, SO_BSDCOMPAT, &one, sizeof(one)))
    ERROR_RETURN("Couldn’t shield 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;

while ((len = recv(rtp_fd, packet, UDPMAXSIZE + 1, 0)) > 0) {
    /* process packet[], be cautious if (len == UDPMAXSIZE + 1) */
}

if ((len == 0) || (errno != EAGAIN)) {
    /* while() may have exited in an unexpected way */
}

Figure 5 -- Code to check rtp_fd for new RTP packets.
/* set RTP address, as coded in Figure 1’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("169.229.60.105");

/* set RTCP address, as coded in Figure 1’s SDP */

rtcp_addr = calloc(1, sizeof(struct sockaddr_in));
rtcp_addr->sin_family = AF_INET;
rtp_addr->sin_port = htons(5005); /* 5004 + 1 */
rtp_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.
3. Session Management: Session Housekeeping

In Section 2, we showed how to extract transport information from the session descriptions that define our example session. We used this information to set up socket descriptors that listen for RTP and RTCP packets, and to set up addresses which may be used with sendto() to send RTP and RTCP packets.

Once this initialization occurs, the two participants begin to send and receive MWPP RTP packets. In Sections 4-7, we discuss MWPP sending and receiving algorithms in detail. In this section, we briefly review secondary "housekeeping" tasks that MWPP parties also perform as a part of session management.

One housekeeping function is the choice (and maintenance) of the 32-bit SSRC value that uniquely identifies each party. Section 8 of [2] describes SSRC issues in detail.

Another housekeeping function is the sending and receiving of RTCP backchannel streams. MWPP uses the standard techniques for sending and receiving RTCP streams, which are described in Section 6 of [2]. However, MWPP uses a somewhat unusual definition of the sampling instant of an RTP packet (see Section 2.1 of [1]), which must be taken into account in the calculation of RTCP reception statistics.

Another housekeeping function concerns security. As detailed in the Security Considerations section of [1], per-packet authentication is strongly recommended for use with MWPP, because the acceptance of rogue MWPP packets may lead to the execution of arbitrary MIDI commands. [11] describes a standardized approach to authenticating RTP and RTCP packets.

A final housekeeping function concerns the termination of an MWPP RTP session. For 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.

Note that the exit of a session participant may be signalled in several ways. Session management tools may offer a reliable signalling method for session termination (such as the SIP BYE method [7]). RTCP also offers a message to code the exit of a participant (the RTCP BYE packet [2]). Receivers may also sense the lack of RTCP activity to effect a timeout mechanism, or may use transport methods to detect an exit.
4. Sending MWPP Streams: General Considerations

In this section, we discuss MWPP sender implementation issues, in the context of the example session shown in Section 2 (an interactive, two-party, single-stream session over unicast UDP transport that uses RTCP). In specific, we describe how the first participant in this example (defined by the session description in Figure 1) sends an MWPP RTP stream to the second participant (defined by the session description in Figure 2).

The interactive MWPP sender in our example is a real-time data-driven entity. On an on-going basis, it examines an incoming source of MIDI data, and decides whether to transmit a new MWPP RTP packet to the second participant. The sender issues new packets for two distinct reasons:

1. One or more MIDI commands have been received from the incoming source of MIDI data, and the sender decides it is appropriate to send a new MWPP RTP packet with the data.

2. No new MIDI commands are queued for transmission, but the sender decides that too much time has elapsed since the last RTP packet transmission, and that the receiver would benefit from the updated data coded in the RTP header and recovery journal sections of a new MWPP packet.

In both cases, the sender generates an RTP packet that consists of an RTP header, a MIDI Command section, and a recovery journal. In the first case, the MIDI list field of the MIDI Command section codes the new MIDI data; in the second case, the MIDI list field is empty.
In Figure 8, we list the 5 steps a sender takes to generate and transmit an MWPP RTP packet. These steps correspond 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 an MWPP packet:


2. Generate the MIDI Command section for the new packet. See Section 3 of [1] for implementation details.

3. Generate the recovery journal for the new packet. We discuss this process in Section 5.2. We note here that the generation algorithm examines the "recovery journal sending structure," a stateful encoding of a history of the stream.

4. Send the new packet to the receiver.

5. Update the recovery journal sending structure, 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 an MWPP RTP packet.

To complete this section, we discuss implementation issues related to the sending algorithm defined by Figure 8, in a series of sub-sections.

4.1 Queuing and Coding Incoming MIDI Data

An interactive MWPP sender examines an incoming source of MIDI data, and decides whether to transmit a new MWPP RTP packet to the second participant. In this section, we review different strategies for deciding when to transmit RTP packets, and discuss coding issues in generating the RTP header and MIDI Command sections for these packets.

The simplest interactive sender algorithm is to transmit a new MWPP RTP packet as soon as the incoming MIDI source presents a new complete MIDI command. The system described in [6] uses this algorithm. An advantage of this algorithm is zero sender queuing latency, as the sender never delays the transmission of a new MIDI command.
In a relative sense, this algorithm is inefficient, as it dedicates an RTP packet, with its recovery journal section and header stack, to the transmission of a single MIDI command. For sparse data streams, this inefficiency may be acceptable (see Appendix A.4 of [6] for analysis). More sophisticated interactive sending algorithms [12] improve stream efficiency by coding small groups of MIDI commands into a single RTP packet, at the expense of non-zero sender queuing latency.

In addition to deciding when to send MIDI command data, a sender must also decide how to code the MIDI command data in the RTP packet. One decision concerns the assignment of a command timestamp to each command. Appendix C.2 of [1] describes three algorithms for command timestamp selection -- comex, async, and buffer -- and defines SDP parameters for choosing and configuring a timestamp algorithm as a part of session configuration. Note that in the example SDP in Figures 1 and 2, the tsmode parameter does not appear, and so the default comex timestamp semantics are in effect.

In addition to assigning timestamp values, a sender must also decide how to code these timestamp values in the MIDI Command section of the MWPP packet (Section 3 of [1]). The most efficient method is to set the RTP timestamp of the packet to the timestamp of the first MIDI command in the MIDI command list, and to set the Z bit of the MIDI Command section header to 0 (Figure 2 of [1]). The delta time of the remaining MIDI commands in the list are calculated relative to the RTP timestamp. However, this timestamp coding scheme produces a stream whose RTP timestamps increment at a non-uniform rate. This behavior, while permitted in [2] [3], may be sub-optimal for certain RTP tools, such as header compression systems [13]. MWPP supports the generation of streams with RTP timestamps that increment at a uniform rate. Senders should set the Z bit of the MIDI command section header to 1, and use the delta time field of the first MIDI command coded in the MIDI list to code a command execution time offset relative to the RTP timestamp.

Finally, as we discuss in Section 6, an interactive MWPP receiver may model statistical properties of the network latency and use this model to optimize its rendering performance. By necessity, receiver models use the timestamp of the last member of the MIDI list as a proxy for the wallclock time that the sender put the packet onto the network (if the MIDI list is empty, receivers models use the RTP timestamp of the packet as the proxy). To the extent possible, interactive senders should maintain a constant relationship between this proxy and the actual wallclock sending time; variance in this relationship is seen by the receiver as network jitter.
4.2 Sending MWPP Packets with Empty MIDI Lists

As we described in the preamble of Section 4, interactive senders may decide to transmit MWPP RTP packets with empty MIDI lists. Senders generate "empty packets" in two contexts: as "keep-alive" packets during periods of no MIDI activity, and as "guard" packets to improve the performance of the recovery journal system. In this section, we discuss implementation issues for empty packets.

In an interactive application, MIDI data sources may not produce MIDI commands for extended periods of time (seconds or even minutes). If an MWPP RTP stream followed the dynamics of a silent MIDI source, and stopped sending RTP packets for an extended periods, systems behavior might be degraded in the following ways:

- Receivers may misinterpret the silent stream as a dropped network connection.
- Network middleboxes (such as Network Address Translation systems) may "time-out" the silent stream and drop the port and IP association state.
- Receiver models of network latency behavior may poorly model the condition of the network.

Senders avoid these problems by sending "keep-alive" MWPP packets (whose MIDI lists are empty) during periods of network inactivity. Session participants may specify the frequency of keep-alive packets during session configuration by using the maxptime SDP parameter, as described in Appendix C.3 of [1]. As a point of reference, the system described in [6] sends a keep-alive packet if no RTP packet has been sent for 30 seconds.

Senders may also send empty MWPP 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 incoming MIDI source produces a MIDI NoteOff command (which the sender promptly transmits in an MWPP packet), but then 5.4 seconds pass before the MIDI source produces another MIDI command (which the sender transmits in a second MWPP packet). If the MWPP packet coding the NoteOff is lost, the receiver will not be aware of the packet loss incident for 5.4 seconds, and the rendered MIDI performance will contain a note that sounds for 5.4 seconds too long.
To handle this situation, senders may transmit empty MWPP packets to "guard" the stream during silent sections. For example, the guard packet algorithm defined in Section 7.3 of [6], as applied to the situation described above, would send a guard packet after 100 ms of MIDI source inactivity, and would send a second guard packet 100 ms. later. Subsequent guard packets would be sent with an exponential backoff, with a limiting period of 1 second. Guard packet transmissions would cease once MIDI activity resumes, or once RTCP receiver reports indicate that the receiver is up to date.

We expect that guard packet sending algorithms will become a "quality of implementation" factor that differentiates MWPP implementations. Sophisticated implementations may tailor the guard packet sending rate to the nature of the MIDI commands that lie "unprotected" 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 [6] provides limited protection against the transient artifacts that occur when NoteOn MIDI commands are lost, by optionally sending a guard packet 1 ms after an MWPP packet whose MIDI list contains a NoteOn command. The Y bit in Chapter N note logs (Appendix A.4 of [1]) supports this use of guard packets.

Clearly, bandwidth management and congestion control are key issues in guard packet algorithms. We discuss these issues in the next subsection.

4.3 Bandwidth Management and Congestion Control

Senders may control the instantaneous sending rate of an MWPP stream in a variety of ways. In this section, we describe the mechanics of MWPP rate control, in the contexts of congestion control and bandwidth management.

RTP implementations have a responsibility to implement congestion control mechanisms to protect the network infrastructure (see Section 10 of [2]). In general, senders implement congestion control by monitoring packet loss via RTCP receiver reports, and reducing the stream sending rate if packet loss is excessive. Section 6.4.4 of [2] provides guidance for using the RTCP receiver report fields for congestion control.

Bandwidth management is a second use for MWPP sending rate control. An SDP session description may optionally include a bandwidth line (b=, as defined in Section 6 of [5]) to specify the maximum bandwidth an RTP stream may use. If an MWPP session description includes a bandwidth line, senders control the instantaneous sending rate of the stream so that the maximum bandwidth is not exceeded.
Interactive MWPP senders have a variety of methods to control the instantaneous sending rate:

- As described in Section 4.1, MWPP senders may pack several MIDI commands into a single MWPP packet, thereby reducing instantaneous 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 and in Section 4.1 of [1]. Note that in all cases, the recovery journal sender must conform to the mandate defined in Section 4 of [1].

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

MWPP senders incorporate these rate control methods into feedback loops to implement congestion control and bandwidth management.
5. Sending MWPP Streams: The Recovery Journal

In this section, we describe how senders implement the recovery journal system. We begin by describing the recovery journal sending structure.

The sending structure is a hierarchical representation of the checkpoint history (defined in Appendix A.1 of [1]) of the stream. The hierarchy mirrors the organization of the recovery journal bitfields. Figure 9 shows the C data structure for a simplified version of sending structure.

The top level of the hierarchy (jsend_journal in Figure 9) corresponds to the top-level recovery journal format (Figure 7 in [1]). The second level of the hierarchy (jsend_channel in Figure 9) corresponds to the channel journal format (Figures 8 in [1]). The leaf level of the hierarchy (jsend_chapterw in Figure 9) corresponds to recovery journal chapters (Appendices A and B of [1]).

The simplified sending structure in Figure 9 is incomplete in several ways: the second-level Systems journal (defined in Figure 9 of [1]) is not present, and only one channel journal chapter type is present (Chapter W, defined in Appendix A.3 of [1], coding MIDI Pitch Wheel (0xE) commands).

Levels of the sending structure hierarchy store several items:

1. The current contents of the recovery journal bitfield for the level (jheader[], cheader[], and chapterw[] in Figure 9).

2. The extended sequence number (seqnum in Figure 9) of the most recent RTP packet that added information to the checkpoint history, at the level or at any level below it. Seqnum is set to zero if the checkpoint history contains no information at the level or at any level below it.

3. Ancillary variables (not present in Figure 9) to simplify operations on the sending structure.

In the sub-sections that follow, we describe how the sender uses the recovery journal sending structure in the example system defined in Section 2 of this memo (an interactive, two-party, single-stream session over unicast UDP transport that uses RTCP).
typedef unsigned char uint8; /* must be 1 octet */
typedef unsigned long uint32; /* must be 4 octets */

/**************************************************************************
/* leaf level of hierarchy: Chapter W, Appendix A.3 of [1] */
/**************************************************************************/

typedef struct jsend_chapterw {
  uint8 chapterw[2]; /* bitfield (Figure A.3.1, [1]) */
  uint32 seqnum; /* extended sequence number, or 0 */
} jsend_chapterw;

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

typedef struct jsend_channel {
  uint8 cheader[3]; /* header bitfield (Figure 8, [1]) */
  uint32 seqnum; /* extended sequence number, or 0 */
  jsend_chapterw chapterw; /* chapter W info */
} jsend_channel;

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

typedef struct jsend_journal {
  uint8 jheader[3]; /* header bitfield (Figure 7, [1]) */
  /* 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 -- Simplified recovery journal sending structure
5.1 Initializing the Sending Structure

At the start of a stream, the sender initializes the sending structure. All seqnum variables are set to zero. The jheader[] is initialized to form a recovery journal header appropriate for 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 of the header is set to the sequence number of the upcoming first RTP packet (following the guidelines in Appendix A.1 of [1]).

5.2 Traversing the Sending Structure

Whenever an MWPP RTP packet is sent (Step 3 in the algorithm defined in Figure 8), the sender traverses the sending structure to generate the recovery journal bitfield.

The traversal begins at the top level of the sending structure. The sender copies jheader[] into the array that holds the recovery journal that is under construction. After performing the copy, the sender sets the S bit of jheader[] to 1.

The traversal continues depth-first, visiting every cheader[] and chapterw[] whose seqnum variable is non-zero. The sender copies cheader[] or chapterw[] into the array that holds the recovery journal that is under construction, and then sets the S bit of cheader[] or chapterw[] to 1.

5.3 Updating the Sending Structure

After an MWPP RTP packet is sent, the sender updates the sending structure (Step 5 in the sending algorithm defined in Figure 8) to refresh the checkpoint history. The sender parses the MIDI command section of the packet sent in Step 4 of the sending algorithm, and performs an update operation for each command encoded in the section.

We now describe the update operation, assuming that the sender has parsed a MIDI Pitch Wheel (0xE) command. First, the sender updates the chapterw[] array of the channel[] that is targeted by the Pitch Wheel command. The update method for the chapterw[] bitfield is described in Appendix A.4 of [1]. Note that the S bit of the updated chapterw[] bitfield is cleared.

Next, the sender updates the cheader[] array for the channel targeted by the Pitch Wheel command and the jheader[] array at the top level. At a minimum, these updates clear the S bit for these header bitfields. In addition, the sender may update other cheader[] and jheader[] fields, to reflect the addition of a new Chapter W or a new channel journal to the recovery journal layout.
Finally, the sender updates the the seqnum variables associated with the changed chapterw[], cheader[], and jheader[] arrays, to code the extended sequence number of the packet coding the Pitch Wheel command.

5.4 Trimming the Sending Structure

At regular intervals, the sender receives RTCP receiver reports (described in 6.4.2 of [1]). 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.

The sending structure trimming algorithm uses the EHSNR to trim away parts of the sending structure, and thus reduce the size of recovery journals sent in subsequent RTP packets. The algorithm (as applied to a two-party session) 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 I <= K may be removed from the sending structure, without violating the recovery journal mandate defined in Section 4 of [1].

The sending structure trimming algorithm runs whenever an RTCP receiver report arrives. First, the EHSNR field is extracted from the receiver report, and adjusted 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 sending structure, starting at the top level. Levels whose seqnum is less than or equal to the adjusted EHSNR are cleared, by setting the seqnum to zero. If necessary, the jheader[] and cheader[] arrays above the cleared level are adjusted to match the new journal layout.

Finally, the checkpoint packet sequence number field of jheader[] is updated to the value coded in the EHSNR. Note that the trimming algorithm does not clear the S bits of the recovery journal bitfields.

5.5 Implementation Notes

The simplified recovery journal sender implementation described in this section differs from the complete, efficient implementation that would be found in a production system.

In a complete implementation, the sending structure shown in Figure 9 would be modified to cover the full recovery journal syntax. Chapter journal structures (modeled on the jsend_chapterw structure) would be added for all of the channel and system chapters defined in Appendices A and B of [1].
An efficient implementation would use enhanced versions of the traversing, updating, and trimming algorithms presented in Sections 5.2-4. These algorithms would rely on ancillary state information that would be added throughout the sending structure. See the recovery journal sender in [14] for example implementations of enhanced sending structure algorithms.

Finally, a production sender implementation would probably implement algorithms that support a variety of MWPP application domains (two-party topologies and multi-party topologies, RTCP and no-RTCP, etc). In the Appendices of this memo, we discuss recovery journal sender issues for application domains beyond the two-party example system described above.
6. Receiving MWPP Streams: General Considerations

[This section discusses MWPP receiver design. Recovery journal receiver issues are covered separately in Session 7.]

7. Receiving MWPP Streams: The Recovery Journal

[This section describes recovery journal receiver processing.]

8. Congestion Control

Congestion control issues for MWPP implementations are discussed in detail in Section 4.3 of this memo.

9. Security Considerations

General security considerations for MWPP are discussed in detail in Section 7 of [1]. Supplemental discussion on MWPP implementation security issues is presented in Section 3 of this memo.

Appendix A. Content Streaming with MWPP

[This section describes using MWPP in content streaming (non-interactive, higher-latency) applications. It includes advice for using RTSP with MWPP, and for using RTP redundancy (RFC 2198) and FEC (RFC 2733) with MWPP.]

Appendix B. Multi-party MWPP Sessions

[This section describes using MWPP in sessions with more than 2 participants. It includes an extended discussion of multicast implementation issues, and a discussion of simulating multicast with multiple unicast flows.]

Appendix C. MWPP and Reliable Transport

[This section describes using MWPP with TCP, and using MWPP in environments where datagrams are known to be reliable.]
Appendix D. Using MWPP without RTCP

This section describes how to use MWPP without RTCP. It offers implementation advice for the hybrid sending strategies defined in Section 4.1 of [1].

Appendix E. Multi-stream MWPP Sessions

This section describes synchronization issues with using multiple MWPP streams in the same session. It includes advice on how to satisfy the recovery journal mandate defined in Section 4 of [1] in a multi-stream identity relationship (as defined in Appendix C.4 of [1].)

Appendix F. Author 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
Appendix G. References


