A Case for Network Musical Performance

John Lazzaro
CS Division
UC Berkeley
Berkeley, CA, 94720
lazzaro@cs.berkeley.edu

John Wawrzynek
CS Division
UC Berkeley
Berkeley, CA, 94720
johnw@cs.berkeley.edu

ABSTRACT

A Network Musical Performance (NMP) 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. In this paper, we present a case for NMP as a practical Internet application, and describe a method to ameliorate the effect of late and lost packets on NMP. We describe an NMP system that embodies this concept, that combines several existing standards (MIDI, MPEG 4 Structured Audio, RTP/AVP, and SIP) with a new RTP packetization for MIDI performance. We analyze NMP experiments performed on CalREN2 hosts on the UC Berkeley, Stanford, and Caltech campuses.

1. INTRODUCTION

Imagine two pianists playing a 4-hands composition. But rather than sitting side by side on a piano bench, one person is located on the UC Berkeley campus, and the other person is located on the Stanford campus. Each musician plays an electronic piano that responds to both the local and the remote player. Control information from the remote player is sent across the Bay via the inter-campus CalREN2 link.

This scenario is an example of a Network Musical Performance (NMP), which we define as a group of musicians, located at different places, who interact over a network to perform as they would if located in the same room.

In this paper, we present a case for NMP as a realizable, practical multimedia application. We begin the paper by discussing the network performance requirements of NMP, focusing on latency requirements and the effects of late and lost packets. We argue that by sending descriptions of the physical gestures musicians use to manipulate their instruments across the network, and generating audio from this gestural stream at each host, we can reduce the impact of network congestion on the performance.

We describe an NMP system that embodies this concept. The system combines existing open standards (MPEG 4 Structured Audio [6] for sound synthesis, IETF RTP [10][9] and SIP [5] for networking, and MIDI [8] for musical control) with a new RTP packetization for MIDI performance that supports the graceful recovery from lost and late packets.

We analyze data from NMP experiments conducted on CalREN2 hosts located on the UC Berkeley, Stanford, and Caltech campuses, and conclude with a discussion of future work. Appendices describe the RTP packetization for MIDI performance in a detailed way.

NOSSDAV'01, June 25-26, 2001, Port Jefferson, New York, USA.

2. PERFORMANCE AND LATENCY

An NMP system unavoidably introduces time delays between the musicians, due to the network latency of the links connecting the players and the local latency at each host. The total latency must be kept reasonably short for the NMP system to be usable.

Low latency is needed because a group of musicians playing a composition can be viewed as a distributed sensory-motor feedback loop: musicians hear the sounds created by the other players, and adjust their playing to produce a coordinated performance. Introducing delays into this system makes the act of performance more difficult.

However, some latency is always present in conventional musical performance – the acoustic latency due to the speed of sound. An intuitive way to think about the musical significance of network latency in NMP is to consider the physical separation that would yield the equivalent acoustic latency, using the formula:

$$\text{separation} = 344\,\text{m/s} \times \text{latency}$$

For example, our measurements show that CalREN2 hosts on the UC Berkeley and Stanford campuses in Northern California, separated by 40 miles, show a median symmetrical latency (RTT/2) of 2.1 ms. This value would correspond to the acoustic latency perceived by two musicians sitting 0.72 meters (2.4 feet) apart, considerably shorter than the separation distance between players in most ensembles.

Similar measurements show a 14.2 ms symmetrical median latency between UC Berkeley and a Southern California CalREN2 host (Caltech campus, 375 miles from Berkeley), corresponding to a musician separation of 4.88 meters (16.0 feet). This distance is comparable to the separation between musicians on a concert hall soundstage.

These intra-California comparisons show that for moderate geographical distances over a quality network, the network latency falls within the range of delays musicians normally accommodate. The remaining hurdle for practical NMP is the local delays introduced at each host: computational delay, audio and control I/O delay, and perhaps local acoustic delay. If these local delays are sufficiently low, a usable NMP system should result.

Finally, we note that an NMP system whose total latency fails this acoustic latency comparison test may still in fact be a usable system. Certain musical instruments have a long production latency (the delay from physical gesture to sound generation), and yet musicians can compensate for these delays during a performance. An extreme example of this phenomena is the pipe organ, whose production latency may be on the order of seconds.

We also note that musical ensembles that routinely play in large performance spaces, such as marching bands and symphony orchestras, include a conductor who keeps the ensemble synchronized via visual cues. An NMP system specialized for long latency situations might also be able to incorporate synchronization cues.

3. NMP AUDIO STREAMING

One approach to NMP is to set up high-quality audio streams between the remote sites. The basic idea is to customize an Internet telephony architecture for low latency operation, by making careful technology choices in the design of the local hosts.

As in standard Internet telephony, the hosts could use the IETF Real Time Protocol (RTP) [10] to exchange audio streams. An uncompressed sample-based packetization would be chosen to eliminate algorithmic codec delay, and the number of samples in each RTP packet (i.e. the packetization interval) would be chosen to minimize buffer delay.

The host audio chain would also be customized for low latency operation. The acoustic latency of the system would be minimized by using headphones and close microphone placement. For a PC-based system, modern soundcards would be chosen to avoid bus performance problems. The application software would directly read and write the DMA buffers accessed by the soundcard to avoid buffering delays, unless the system audio API supported small buffer sizes.

If these design principles are followed, and if the underlying network has low latency, over-provisioned bandwidth, and (ideally) Quality of Service (QoS) mechanisms, an audio streaming approach to NMP can be quite effective. For example, a recent paper [2] describes successful Internet 2 performances using uncompressed multichannel audio streams.

In our paper, we consider the case for NMP over networks where audio streaming is not practical, because of bandwidth limitations, network congestion, or the lack of QoS. In these networks, occasional packet delays and losses are inevitable, as other users transiently consume resources.

Under these conditions, sending raw or compressed audio signals at the network latency (i.e. small buffers on the hosts) leaves few options for hiding transient network problems. Packet delay and loss would result in garbled sections of sound, which would disrupt the performance.

4. GESTURAL NMP

Concealing packet loss is easier if the musical performance is sent across the Internet at a higher level of abstraction, that describes the physical gestures musicians use to manipulate their instruments. If the proper representation is chosen, artifacts produced by imperfect networks should sound closer to mistakes produced by imperfect musicians (notes that are skipped, played late, or which are held too long) than to the dropouts, clicks and gurgles produced by data-starved audio codecs.

In this model, each host should execute identical audio signal processing algorithms to generate the sounds of the instruments played in the session, under the control of local and network gestural data. Gestural data sent across the network should be tagged with timestamps and sequence numbers, and should include contextual information about recently sent gestures, so that late and lost packets can be detected and concealed.

For example, imagine two pianists, in different locations, playing a 4-hands composition on electronic keyboards that generate control information (piano key depression and piano key release events). At both sites, audio is generated in response to local and remote key events, using computer music synthesis techniques to generate the piano sounds.

In this scenario, delayed or lost control information can often be gracefully handled in an event-specific way. For example, lost and late key depressions are best skipped, so that a note does not sound at an inopportune time. Conversely, lost and late key releases are always executed, so that a depressed key does not sound indefinitely. Other gestures such as sustain pedal movements and volume changes have different recovery semantics that reflect their sonic effect. With careful design, the qualitative effect of an occasional lost or late packet can be made quite subtle.

We have implemented an experimental system for gestural NMP, that is based on open standards from MPEG and the IETF. The system includes an *NMP client,* which is used by each musician in a performance, a *conference server,* which coordinates communications between clients, and a *mirror server,* used for session debugging.

5. CLIENT AND SERVER OVERVIEW

The NMP client is a real-time sound synthesis engine, that is driven both by local control information (from the musician playing at the client site) and networked control information (from musicians at other sites). The sound synthesis engine is programmable, so that musicians can exchange instrument models at the start of a session. The NMP client also transmits the local musician's gestural information to the other clients, and interacts with the conference server.

The NMP client uses Structured Audio (SA) for music synthesis, which is part of the MPEG-4 audio standard [6]. This standard defines a programming language, SAOL, that is specialized for sound. SAOL semantics are normative: a compliant decoder generates audio that sounds identical to the audio produced by other compliant decoders. SAOL supports two types of control input: the popular MIDI format, and the more powerful SASL language.

The NMP client uses the **sfront** SA engine [7]. Sfront executes a SA program by converting it into a C program that generates audio when executed. The C programs sfront generates support low-latency real-time interaction, via audio input from a microphone and control input from music instrument controllers that generate MIDI.

We have extended sfront to act as an NMP client by adding the ability to send and receive UDP Internet Protocol packets. The NMP client uses the IETF Real Time Protocol (RTP) [10] under the AVP profile [9] to exchange MIDI control information with the other endpoints. The NMP client uses a new RTP packetization for MIDI that is specialized for performance applications. The client fully

Table 1: MPEG 4 Structured Audio semantics for MIDI

Name	Bit Pattern	Action taken on channel `cccc`
NoteOff	`1000cccc 0nnnnnnn 00000000`	End note number `nnnnnnn`, velocity ignored.
NoteOn	`1001cccc 0nnnnnnn 0vvvvvvv`	Start note number `nnnnnnn` with velocity `vvvvvvv`, `0000000` $\rightarrow$ *NoteOff*.
PTouch	`1010cccc 0nnnnnnn 0aaaaaaa`	Set aftertouch value for note number `nnnnnnn` to `aaaaaaa`.
CControl	`1011cccc 0xxxxxxx 0yyyyyyy`	Set controller number `xxxxxxx` to value `yyyyyyy`.
PChange	`1100cccc 0ppppppp`	Set channel `cccc` to play program timbre `ppppppp`.
CTouch	`1101cccc 0aaaaaaa`	Set aftertouch value for channel `cccc` to `aaaaaaa`.
PWheel	`1110cccc 0xxxxxxx 0yyyyyyy`	Set pitch bend 14-bit value to `yyyyyyyxxxxxxx`.
System	`1111xxxx ...`	All *System* commands ignored.

implements the Real Time Control Protocol (RTCP) back-channel of RTP.

Musicians using NMP clients do not need to know the Internet addresses of the other musicians in a session. Instead, the musicians choose a session name for the performance, and the NMP clients interact with a conference server to handle connection details. Our conference server uses the Session Initiation Protocol (SIP) [5].

Our system also includes an auxiliary server for system testing, called a mirror server. Participants may temporarily launch a mirror server at the start of the session, to help debug the link and measure latencies.

Mirror servers implement the RTP and SIP functionality of an NMP client, but do not include an SA engine and are not meant to be locally controlled by a musician. When a mirror server receives an RTP packet, it alters the information in the packet (for example, by transposing notes by a fixed interval) and returns the modified packet to the sender. In a typical use, a player may launch a mirror on a remote machine, play a few notes to "hear the echo," and then disable it.

The next two sections of the paper describe the RTP packetization for MIDI performance in more detail. The packetization is best understood as an RTP encapsulation of the MIDI wire protocol for transport over unreliable networks, for applications where the latency cost of retransmitting lost packets is unacceptable. Section 6 summarizes the MIDI wire protocol as used by Structured Audio, while Section 7 describes how the RTP packetization encodes the MIDI wire protocol.

We note that the basic idea of sending the MIDI wire protocol over packet networks is not new. For example, the Open Sound Control standard [12] is a popular protocol for music synthesizer control that has been used for MIDI-over-IP. Our work differs from previous efforts in its compatibility with RTP, and its focus on feed-forward approaches to lost packet recovery in lieu of packet retransmission.

6. THE MIDI WIRE PROTOCOL

The MIDI standard [8] has its roots in data communication: the protocol was designed to send real-time musical control data over coaxial cables, so that a musician could trigger several music synthesizers from one piano keyboard.

MIDI is sent on a wire as a series of variable-sized commands, using a bit encoding that makes command segmentation implicit. The most popular MIDI commands are 1, 2 or 3 bytes long, and the fixed link bandwidth of 320 microsecond/byte results in a command latency of a fraction of a millisecond. MIDI commands do not have timestamps, but are meant to be executed upon receipt.

Table 1 summaries the MIDI commands that Structured Audio supports. These commands specify a specific gestural action to occur on one of 16 channels (specified with the `cccc` bits in the first byte of each command).

For example, the *NoteOn* command specifies the start or end of a note with a particular pitch (coded by a 7-bit note number `nnnnnnn`) and strike velocity (coded by a 7-bit value `vvvvvvv`).

Other commands code the end of a note (*NoteOff*), a change in value of a pressure sensor under each key (*PTouch*) or under the entire keyboard (*CTouch*), the change in value of continuous pitch bend (*PWheel*), or state changes in ancillary devices such as pedals and sliders (*CChange*).

MIDI uses the channel concept to let musicians play several different timbres at once. For example, a piano keyboard controller may be configured to send notes below Middle C on one channel, and note above Middle C on a second channel. Channels are bound to timbres using the *PChange* command; Structured Audio uses this command to bind a SAOL instrument model to a MIDI channel.

Note that the first byte of MIDI commands (the command byte) begins with a binary 1, but all subsequent bytes (the data bytes) begin with a binary 0. This coding scheme supports the *running status* data compression technique: a sequence of MIDI commands sharing the same channel and command type may be sent as a single MIDI command byte followed by the data bytes for each command in the sequence. Running status is particularly efficient when coding long strings of *NoteOn* and *NoteOff* commands, because *NoteOff* commands may alternatively be coded as *NoteOn* commands with zero velocity.

The exact semantics of a protocol whose commands describe artistic expression are inevitably slippery. However, MIDI is over 20 years old, and many product generations of musical instruments have collectively defined interoperability in practice. Our RTP packetization uses these semantics to guide the recovery procedures for lost and late packets.

Experienced MIDI users may notice that Table 1 is a restricted version of MIDI in several minor respects: the *NoteOff* velocity and all MIDI *System* commands are ignored. These restrictions are normative elements of the Structured Audio standard, and our RTP packetization uses this fact to optimize the recovery procedure for lost packets.

Finally, we note one unfortunate aspect of MIDI that constrained the design space for our RTP packetization. The coding technique used for sending notes – matched pairs of *NoteOn* and *NoteOff* commands – is very vulnerable to lost commands. A single lost *NoteOff* command results in a note sounding arbitrarily long.

7. RTP PACKETIZATION

Figure 1 shows the structure of an RTP packet in the MIDI packetization. A packet begins with the standard RTP header, which is followed by a *MIDI command* section and a *recovery journal* section. In this section, we describe how to send and receive these packets, and discuss aspects of the packetization design.

Figure 1: Packet format

```
RTP header:

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       Sequence Number         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Timestamp                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             SSRC                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Payload:

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|R|    LEN    |          MIDI Command Payload ...             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Recovery Journal ...                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

7.1 Sending packets

Structured Audio programs define a control rate, which typically ranges from 500Hz to 4000Hz for real-time work. Our NMP client checks the MIDI In port of the soundcard at the control rate, to see if new MIDI bytes have arrived from an attached musical instrument controller.

If the NMP client determines that a complete MIDI command has arrived, it schedules the command for immediate execution on the local Structured Audio engine, so that the local player hears a low-latency response to local actions. The NMP client also prepares a new RTP packet, and sends it to the other players at remote locations. All players receive the same packet; in this sense, the system emulates a multicast transport by using several unicast flows.

The sequence number of the RTP packet is one greater than the last packet sent, modulo 32. The timestamp of the RTP packet reflects the local Structured Audio time, in the units of the audio sampling rate: if the sampling rate is 44,100 Hz, two packets sent exactly 2 seconds apart have timestamps that differ by 88,200.

The MIDI command section of the packet codes the new data generated by the local player. The MIDI command bytes are placed in the payload field of this section, and the `LEN` field is set to code the number of payload bytes. The 6-bit `LEN` field supports sending many MIDI commands in a single packet; note that the first MIDI command in a payload may not use running status compression, but subsequent commands may use this technique. The `R` bits are reserved for future use, as are all bits named `R` in the format.

The recovery journal section of the packet holds information a receiver uses to recover from earlier lost packets. The journal enables the recovery from the loss of all RTP packets sent since an earlier RTP packet, called the *checkpoint packet*. Typically, a sender creates the recovery journal for packet P_i by slightly modifying the recovery journal sent in packet P_{i-1} to reflect the MIDI command section of packet P_{i-1}. Appendix A.1 describes the recovery journal format in detail.

The sender also tracks the last RTP packet received at each remote location, by monitoring the data returned in RTCP receiver reports. The sender uses this knowledge when preparing the recovery journal of a new RTP packet: it trims journal data for recovering from the loss of packet P_i once it knows that all remote locations have received a packet $P_{j>=i}$. By trimming the data, the sender updates the identity of the checkpoint packet. Note that this feedback scheme is compatible with multicast transport.

7.2 Receiving packets

When a new RTP packet arrives from a remote client, the NMP client first determines if the packet has arrived significantly late, and if so sets the binary flag `late` to 1. As described in Appendix B, the NMP client judges the lateness of an incoming packet by comparing its RTP timestamp against a model of the current time at the remote client.

The NMP client then examines the sequence number of the RTP packet, and determines if previous packets have been lost, or if this packet has been delivered out of sequence.

If neither condition is detected, the client processes the the MIDI command section of the packet, scheduling each MIDI command in the payload for immediate execution on the local Structured Audio engine. However, if the `late` flag is set, some MIDI commands may be skipped, because their execution would disturb the musical performance.

The current NMP client uses a simple decision rule for skipping late commands: all *NoteOn* commands with non-zero velocity are skipped, all other commands are executed. This rule acts to mute notes that would otherwise play many beats too late, but promptly ends notes which are sounding beyond their intended length.

If packet loss has been detected, the NMP client parses the recovery journal section of the packet. If necessary, the NMP client schedules one or more MIDI commands for immediate execution on the local Structured Audio engine, to implement recovery actions. The recovery procedure takes into account the state of the `late` flag; Appendix A.3 describes the recovery procedure in more detail. Once recovery processing is complete, the NMP client executes the MIDI command section of the RTP packet, as described above.

If out of sequence delivery is detected, the RTP packet is discarded, without processing its MIDI command payload; an earlier RTP packet delivery resulted in the parsing of a recovery journal and the execution of recovery measures.

7.3 Discussion

The recovery journal mechanism shares common traits with forward error correction (FEC) and reliable multicast transport (RMT) approaches to resilience [1] [4]. Like FEC, redundant information is included in the forward transmission path, in lieu of packet retransmission.

Like most RMT schemes, an aspect of behavior is guaranteed regardless of the number of packets lost: the first packet received after the lost packet(s) results in the recovery from all artifacts attributable to the loss, and restores the system to a normal state.

For example, after several consecutive lost packets, several notes might be "stuck on" (due to lost *NoteOff* commands), the timbre for a MIDI channel might be incorrect (due to a lost *PChange* command), and the volume of a channel might be incorrect (due to a lost *CTouch*) command. The first packet received after the loss is guaranteed to fix these issues. This guarantee can be made because senders do not remove information from the recovery journal until RTCP receiver reports from all remote locations indicate that it is safe to do so.

As worded above, this guarantee has a loophole: if a long period of time occurs without a musician generating a MIDI event, no new RTP packets are sent to the remote sites. If the last packet sent was lost, the artifacts due to the loss could persist arbitrarily long.

To handle this situation, our NMP client detects periods of musician inactivity, and sends *guard packets* to the remote clients. The MIDI command section of these guard packets has an empty payload (the LEN field is set to zero).

In the current implementation, the NMP client sends the first guard packet after 100 ms. of inactivity, and a second guard packet 100 ms. later. Subsequent guard packets are sent with an exponential backoff, with a limiting period of 1 second. The client ceases sending guard packets once MIDI activity resumes, or once RTCP receiver reports indicate that all remote clients are up to date.

In addition, the NMP client optionally sends a single guard packet shortly after sending a packet that contains a *NoteOn* command, if no packet has been sent in the interim. This technique provides good protection against the transient artifact of skipped notes due to lost *NoteOn* commands, at a moderate bandwidth cost.

Another aspect of the recovery journal guarantee concerns what it does *not* guarantee: no claim is made that a receiver can reconstruct all lost MIDI command packet payloads by examining the recovery journal.

As Appendix A details, the recovery journal records the *most recent* MIDI event of a particular type since the checkpoint packet, not *all* MIDI events of this type. For example, 20 *PWheel* commands may have been sent on a channel since the last checkpoint packet (60 bytes of total MIDI command payload), but the recovery journal only codes the data in the 20th *PWheel* command (using 2 bytes). The relative weakness of this guarantee makes the packetization adequate for MIDI performance transport, but inappropriate for use in archival applications.

However, the weakness of the guarantee is also what makes the packetization practical: as analyzed in Appendix A.4 and experimentally shown in Section 8.3, the bandwidth overhead of the recovery journal mechanism is reasonable. The weak guarantee, coupled with the use of RTCP feedback to trim the journal of irrelevant events, prevent the recovery journal from consuming an arbitrary amount of bandwidth as a performance proceeds.

Finally, we compare the dynamics of the packetization to standard RTP audio packetizations, to highlight the differences in the two approaches. Apart from periods of silence, most RTP audio packetizations send packets at regular intervals, and the RTP timestamp reflects this pace. If no FEC is used, the data in a single packet completely codes the audio for a fixed time interval of known duration.

In contrast, the MIDI packetization sends out packets irregularly, reflecting physical motions of a human. The packets code MIDI commands which receivers execute immediately, but whose sonic effects may last an arbitrary amount of time. Many packets may contribute to sound generation during a period of time, often in an additive way (for example, several *NoteOn* commands, sent in different packets, that create a sustained chord). RTP timestamps directly participate in the fine-time structure of audio generation: the algorithm for discarding late *NoteOn* commands relies on the RTP and RTCP timestamp values (Appendix B).

These differences can be reduced to a single observation: the MIDI packetization maps an asynchronous information source (a musician) to an asynchronous channel (packet networks) without an intermediate synchronous representation. The preservation of distinct events during transport makes the graceful recovery from lost and late packets possible.

8. NMP EXPERIMENTS

8.1 Experimental Setup

This section describes a series of network experiments we conducted using the NMP system described in Section 5, in May 2001. All experiments use the recovery journal and late packet detection technologies described in Section 7. The system setup consists of an NMP client at UC Berkeley, two mirror servers, located at Stanford and Caltech, and a SIP conference server for session management, located at UC Berkeley. In most of the experiments, a musician controls the NMP client, using a MIDI piano keyboard controller; one of the authors (JL) acted as the musician in the experiments described below.

This experimental approach lets us evaluate the effect of latency and packet loss on NMP in a controlled situation, which would be more difficult to do if the mirrors were replaced with additional musicians running NMP clients.

In response to keyboard actions, the musician hears an audio response from three sources: an unfiltered command stream direct from the keyboard, and the same command stream processed by the mirrors at Caltech and Stanford. Any stream may be optionally muted. The mirrored streams differ from the direct stream in two respects: added delay due to the round-trip network latency and mirror processing time, and added network congestion artifacts (late and lost packets).

Table 2: Traceroute to Stanford mirror

1	e0-5.inr-180-soda.Berkeley.EDU	0.351 ms
2	gig10-cnr1.EECS.Berkeley.EDU	0.460 ms
3	gigE5-0-0.inr-210-cory.Berkeley.EDU	0.733 ms
4	vlan229.inr-201-eva.Berkeley.EDU	1.202 ms
5	gigE2-0.inr-000-eva.Berkeley.EDU	0.727 ms
6	pos3-0.c2-berk-gsr.Berkeley.EDU	0.908 ms
7	SUNV--BERK.POS.calren2.net	2.073 ms
8	STAN--SUNV.POS.calren2.net	2.446 ms
9	i2-gateway.Stanford.EDU	2.512 ms
10	Core4-gateway.Stanford.EDU	3.112 ms
11	rtf-gateway.Stanford.EDU	3.295 ms
12	stanford mirror	3.061 ms

Tables 2 and 3 shows a traced route from the NMP client to the Stanford and Caltech mirror machines during the experiments. Traced routes back to the NMP client traversed the same paths in the inverse direction; route stability was not formally monitored during the experiments, but in general these routes tend to be stable. The long-distance portion of the routes were point-to-point for the Berkeley-Stanford link (7-8 in Table 2) but more circuitous for Berkeley-Caltech (7-11 in Table 3); the intra-campus por-

Table 3: Traceroute to Caltech mirror

1	e0-5.inr-180-soda.Berkeley.EDU	0.348 ms
2	gig10-cnr1.EECS.Berkeley.EDU	0.470 ms
3	gigE5-0-0.inr-210-cory.Berkeley.EDU	0.737 ms
4	vlan229.inr-202-doecev.Berkeley.EDU	1.121 ms
5	gigE3-0.inr-000-eva.Berkeley.EDU	0.776 ms
6	pos3-0.c2-berk-gsr.Berkeley.EDU	0.789 ms
7	SUNV--BERK.POS.calren2.net	2.157 ms
8	C2-QANH-GSR-QSV-GSR.ATM.calren2.net	22.58 ms
9	UCI--QANH.POS.calren2.net	23.45 ms
10	UCR--UCI.POS.calren2.net	24.39 ms
11	CIT--UCR.POS.calren2.net	26.00 ms
12	BoothBorder-Calren.caltech.edu	26.39 ms
13	Moore-RSM2.ilan.caltech.edu	26.69 ms
14	caltech mirror	26.74 ms

tion of the routes were simplest at Caltech (12-14 in Table 3) and most complex at Berkeley (1-6 in Table 2).

In these experiments, the NMP client runs on a 450MHz Intel PIII PC with 128MB of RAM and SCSI disk, running Linux 2.2.17. A Creative Labs PCI 128 card, driven by the `es1370` OSS-free device driver, handles audio and MIDI I/O processing. The 49-note MIDI piano keyboard controller is capable of generating the *NoteOff*, *NoteOn*, *PChange*, *PWheel*, and *CChange* commands. The mirror and conference servers ran on a variety of platforms (SunOS 5.7, HPUX 9.0, Linux 2.2.17), under a range of system loads.

In most of these experiments, the NMP client runs the `linbuzz` SAOL instrument, an example file in the sfront distribution [11]. The amplitude envelope of the instrument does not decay with time: once launched with a *NoteOn* command, the voice sounds at full volume until a *NoteOff* command executes. The voice uses *PWheel* commands for continuous pitch bending, and uses the volume and modulation wheel *CChange* controllers to dynamically modulate the amplitude and spectral shape of the sound.

8.2 Qualitative Experiences

To qualitatively test the system, we configure it to use a single mirror, and we mute the local feedback to the musician. The musician relies on the network stream for audible feedback: network congestion and delay directly disturb the sensory-motor feedback loop. This setup creates a musical instrument whose latency and reliability depends on network behavior.

In our experience, this configuration results in a playable musical instrument, for both the Stanford and the Caltech mirrors, as long as the network links are not experiencing prolonged episodes of severe congestion. For the Stanford mirror, the nominal system latency is difficult to distinguish from the latency of local feedback. For the Caltech mirror, significant nominal latency is readily apparent; however, a musician can quickly adjust to the delay and play fluidly. Network congestion affects operation in several ways:

1. Depressed keys sometimes do not sound (a consequence of lost and late *NoteOn* commands).

2. The onsets time of NoteOn commands that do sound have perceptible jitter. Note that the `maxlate` parameter in Equation B.1 in Appendix B controls the trade off between onset jitter and note skipping.

3. Released keys sometimes keep sounding for a short time period before falling to silence (a consequence of lost and late *NoteOff* commands).

The severity of these artifacts varies widely from session to session; the perceived quality of a particular performance session hinges on the level and distribution of these impairments. We quantitatively explore this issue in Section 8.4.

Note that without the recovery systems described in Section 7, the artifacts from lost and late *NoteOn* and *NoteOff* command would be severe: mild congestion could result in clusters of notes sounding at inopportune times, and lost packets could result in stuck notes.

8.3 Payload Bandwidth

The payload bandwidth of the MIDI RTP packetization depends on two factors: the number of packets sent per second and the payload size of each packet. These factors vary dynamically; the actions of the sending musician influence the payload bandwidth, as does the behavior of the network and the receiving clients (because RTCP receiver reports influence recovery journal trimming).

We designed an experiment to measure the upper range of the payload bandwidth of the MIDI packetization, when driven by a piano keyboard controller. We used the setup described in Section 8.1, activating both the Stanford and Caltech mirrors: multiple receivers reduce the effectiveness of receiver report trimming, since data must remain in the recovery journal until all receivers acquiesce to its removal. We also enabled the NMP client transmission of optional guard packets to protect *NoteOn* commands (Section 7.3).

Using this setup, we recorded the sending time, payload size, and MIDI payload command type of all RTP packets, for a 5.6 minute keyboard improvisation. The musical playing style alternated between fast rhythmic bursts of two-handed chords and single-handed melody lines enhanced with pitch and modulation wheel motion.

We measured the distribution of MIDI commands sent during the performance. The majority of packets (67%) code *NoteOn* and *NoteOff* commands. Controller wheels usage results in 12% of the packets carrying *CChange* and *PWheel* commands. About 21% of the packets have empty payloads; the bulk of these guard packets protect *NoteOn* packets.

We analyzed the data trace for the performance, and calculated three measures – the number of packets per second, the number of bits per packet, and the number of bits per second – averaged over 1 second bins during the 5.6 minute performance. Histograms of these traces show distributions that are roughly unimodal in shape. Table 4 summarizes the traces using several common statistics.

Table 4: Payload bandwidth statistics

Measure	packets/s	bits/packet	bits/s
Median	29	162.8	4712
Mean	30.1	161.8	4979
Std D	11.7	16.7	2188
Min	0	32	0
Max	70	199.2	12840

The median bandwidth of 4.7 kb/s is consistent with the analysis presented in Appendix A.4, and is in the range of modern Internet telephony voice codecs. The median payload size (20 bytes) and median packetization interval (34 ms.) are also in the range of voice codecs during a talkspurt.

The bandwidth variance is significant, and is a consequence of the variance in the packet sending rate and the payload size. The packet sending rate is a direct function of the number and type of MIDI commands generated by the musician.

The payload size has a more subtle connection to musical activity, as mediated by the data representation of the recovery journal (Appendix A.1). For example, a single *PWheel* command expands the size of every packet sent thereafter by 3 bytes, until RTCP receiver reports from all receivers indicate the safety of removing Chapter W from the journal.

8.4 Late and Lost Packets

Section 8.2 describes the qualitative effect of lost and late *NoteOn* and *NoteOff* commands on a performance. To quantify this issue, we designed an experiment to estimate performance quality at regular intervals during a weekday (Wednesday May 9, 2001).

At hourly intervals between 10:30 and 19:30 PST, the musician performed for 5 minutes; we recorded the $t_m - t_p$ value for each received packet (see Appendix B), and information about each lost packet detected. We activated both the Stanford and Caltech mirrors; the musician heard only local feedback, so that artifacts would not alter the playing style. The musician played continuously during the 5 minute interval, alternating between melodies and chordal patterns with no pauses. Total packets sent were roughly constant from hour to hour (mean 7498, standard deviation 261).

The mirrors were configured to preserve the RTP timestamps of incoming packets, so that delays introduced in the link from the client to the mirror would be seen by the late packet detection algorithm (Appendix B). We chose this method so that lost and late packets would have the same network link dependencies.

Late packet statistics for each performance are shown in the first column of Tables 5 (Stanford) and Table 6 (Caltech). We use Equation B.1 in Appendix B to determine if a packet is late. Late packet rates vary from $< 1\%$ to $> 10\%$ for both links, and vary systematically during the day. Lost packets were less common. All sessions had a lost packet rate under 1.0%; 50% of Caltech sessions and 80% of Stanford sessions had a lost packet rate under 0.1%. Even under low loss rates, the recovery journal system is essential: in its absence, a single lost *NoteOff* destroys the performance.

To estimate the impact of network congestion on performance quality, we break each performance into 60 non-overlapping 5-second intervals, and rank the quality of each interval as *Perfect*, *Impaired*, or *Damaged*. Tables 5 and 6 show the distribution of rankings for each session on each mirror; bold-face rows show the best and worst session of the day, based on the number of damaged intervals.

To rank an interval, we determine the number of late *NoteOn* and *NoteOff* commands it contains; the interval associated with a command is determined by the estimated time the command was sent. If an interval contains no late *NoteOn* or *NoteOff* commands, it is ranked perfect. Impaired intervals have less than 15% late *NoteOn* and 15% late *NoteOff* commands; all remaining intervals are damaged. This simple estimate does not consider lost packets.

The rankings in Table 5 and 6 show bimodal behavior: for most of the day, the performance quality is good or very good, but for a few time periods (11:30 and 19:30) damaged intervals are in the 25-50% range, and the system as designed is unusable. An examination of trace data for the outlier sessions shows congestion delays of several hundred milliseconds throughout the damaged intervals, a hostile network environment for any interactive application.

Table 5: Performance fidelity: Stanford

Time	% Late	% Perfect	% Impaired	% Damaged
10:30	0.5	91.7	6.67	1.67
11:30	10.1	58.3	13.3	28.3
12:30	**0.0**	**100**	**0.0**	**0.0**
13:30	0.4	88.3	11.7	0.0
14:30	0.0	100	0.0	0.0
15:30	0.3	93.3	5	1.67
16:30	2.5	85	6.67	8.33
17:30	7.1	83.3	5	11.7
18:30	1.5	90	1.67	8.33
19:30	**15.5**	**51.7**	**1.67**	**46.7**

Table 6: Performance fidelity: Caltech

Time	% Late	% Perfect	% Impaired	% Damaged
10:30	3.9	73.3	21.7	5
11:30	16.1	36.7	23.3	40
12:30	**0.5**	**85**	**15**	**0.0**
13:30	2.4	65	30	5
14:30	1.3	78.3	16.7	1.67
15:30	1.0	81.7	16.7	1.67
16:30	2.5	80	11.7	8.33
17:30	8.3	71.7	15	13.3
18:30	1.7	80	15	5
19:30	**17.0**	**43.3**	**11.7**	**45**

8.5 Latency Measurements

As we described in Section 2, the total latency of an NMP system should be kept reasonably short. We now describe an experiment that estimates the total latency of our system.

In this experiment, a musician does not send MIDI commands to the NMP client. Instead, we modify the NMP client to send MIDI commands to the MIDI Out jack of the soundcard, and use a loopback cable to connect the MIDI Out and MIDI In soundcard jacks.

In this way, a client can generate a MIDI *NoteOn* command on the loopback cable at a known time, and then detect the resulting sound as it plays through the speaker. The client senses speaker output by monitoring the audio input jack of the soundcard, which connects to a microphone placed in front of the speaker. The elapsed time from the *NoteOn* issuance to the sound detection is a conservative estimate of total latency. We describe the estimate as conservative because it includes the time for audio input, which is not part of the total latency of the NMP client.

To make a latency measurement, we run two SAOL instruments on the NMP client: one instrument generates a 5 kHz 100 ms. tone in response to a *NoteOn* command, while the second instrument monitors the microphone input from the soundcard, and detects the onset of the 5 kHz tone.

Table 7 shows the measured total latency for local feedback, Stanford mirror, and Caltech mirror cases. For each test, we measured the latency 60 times over a minute; the table shows the mean and standard deviation of the 60 latency values. The mirror rows also include the round trip times, as measured from RTCP reports received during the test. Care was taken to avoid periods of network congestion: our goal was to measure quiescent network conditions.

Table 7: Total latency, RTCP round-trip times

	Total Latency		RTT		
	Mean	**S. D.**	**Mean**	**S. D.**	
Local	6.09	0.22	–	–	ms
Stanford	9.79	0.32	3.62	0.16	ms
Caltech	33.5	0.37	27.4	0.43	ms

The data in Table 7 are self-consistent: for each mirror, subtracting the round-trip time from the total latency yields a value close to the local feedback total latency. Table 7 also explains why it was difficult for the musician in Section 8.2 to distinguish the latency of the Stanford mirror from the local feedback latency: note these values are quite close.

To understand the limitations of this measurement, we briefly discuss NMP client implementation details. We use the POSIX real-time scheduling support in Linux 2.2.17 to place the NMP client at the highest priority; this ensures the client promptly runs whenever it is not blocking.

The client blocks only on audio I/O. The audio driver uses 4 output buffers, each holding 725 μs of audio: the driver blocks on the fifth consecutive buffer `write()`, and remains blocked until the first buffer has been fully turned into sound. Audio input blocks on `read()` in a similar way. Four `write()`s of silent buffers start a session.

Most of the time, the NMP client is blocked awaiting audio output completion. When a buffer is ready, the client quickly fills up a new buffer (and perhaps handles non-blocking MIDI, network, or onset detect chores) and then blocks upon the audio output `write()`. Given this scenario, we can know the time the client requests a MIDI Out write to $\pm$ 0.36 ms ($\pm$ one-half the buffer length), assuming jitter-free OS scheduling. Note that all timing is cued from audio I/O buffer counts; `gettimeofday()` is not used.

We note that much of the local feedback latency shown in Table 7 can be attributed to known factors: 2.9 ms. of audio output buffering, 0.64 ms. of running-status MIDI wire time, and at least 0.725 ms of audio input buffering leaves only 1.825 ms. of delay attributable to other factors.

9. FUTURE WORK

The experiments described in Section 8 have many limitations: the artificial nature of the mirror setup, the small number of network links tested, and the use of a single musician for qualitative testing. We are currently preparing to conduct multi-player performance experiments on a wider range of networks. We also plan to work on several related topics that impact NMP, as described below.

9.1 Reducing Last-Mile Latency

The experiments described in this paper use hosts that connect to enterprise routers via low-latency Ethernet network interface cards. Unfortunately, most xDSL, cable modem, and POTS modems trade shorter latency for higher bandwidth, without providing hooks for applications to alter the tradeoff. In many cases, the delay of these "last mile" technologies dominates the end-to-end latency between two hosts, and results in a total latency too high for usable NMP.

A potential solution to this problem lies in enhancing these last-mile technologies to support dynamic reconfiguration of the latency-bandwidth tradeoff. Recent work in DSL telephony standardization [3] is promising in this regard.

9.2 Acoustic Control Input

Certain musical instruments, such as the human voice and the guitar, are not easily replaced by a MIDI controller. These instruments are best incorporated into the framework described in this paper by capturing real-time input from the instrument using a microphone, and coding it in a gestural representation that is customized for the instrument.

In some cases, MIDI might be a suitable control language for these special-purpose codecs. However, the SASL control language (a part of MPEG 4 Structured Audio) has advanced features that simplify the implementation of codecs in SAOL. An RTP packetization for SASL is needed, that handles network congestion in a graceful manner.

10. CONCLUSIONS

In this paper, we presented a standards-based approach to NMP. We described a way to gracefully recover from lost and late packets during a performance, and showed experiments that confirm the effectiveness of the recovery system.

11. ACKNOWLEDGMENTS

Thanks to Carver Mead, Tobi Delbruck, Michael Godfrey for access to machines at Caltech and Stanford, and to the anonymous reviewers for many helpful suggestions. This work supported by DARPA contract DABT63-C-0048.

12. REFERENCES

[1] J. C. Bolot and A. V. Garcia. Control mechanisms for packet audio in the Internet. In *Proceedings of the Conference on Computer Communications*, 1996.

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

[3] TR-039: Requirements for voice over DSL. Technical report, DSL Forum, 2001. Annex A addresses latency.

[4] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. RFC 2887: The reliable multicast design space for bulk data transfer, 2000.

[5] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. RFC 2543: SIP: Session initiation protocol, 1999.

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

[7] J. Lazzaro and J. Wawrzynek. Compiling MPEG 4 structured audio into C. In *Proceedings of the 2nd IEEE MPEG-4 Workshop and Exhibition*, June 2001.

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

[9] H. Schulzrinne. RFC 1890: RTP profile for audio and video conferences with minimal control, 1996.

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

[11] Sfront source code release, http://www.cs.berkeley.edu/~lazzaro/sa/.

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

APPENDIX

A. THE RECOVERY JOURNAL

A.1 Recovery Journal Format

A recovery journal has a 3-level structure (Figure 2). At the top level, the journal consists of a 3-byte header followed by a list of *channel journals*, that hold recovery information for a single MIDI channel.

Figure 2: Recovery Journal Format

```
Top-Level Structure:

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|A|K|R|TOTCHAN|    Checkpoint Packet Seqnum   | Channels ...  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Channel Journal:

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S| CHAN  |R|       LENGTH        |P|W|N|A|T|C|R|R|  Chapters ... |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A recovery journal may hold 0-16 channel journals. The `A` flag in the journal header is set to indicate an empty journal; if `A` is clear, a list of $\texttt{TOTCHAN} + 1$ channel journals follow the header. The header provides the sequence number of the checkpoint packet; the `K` flag signals checkpoint updates.

The header also includes an `S` bit, which appears in many places in the packetization with uniform semantics: the `S` bit is set if the structure may be safely ignored the event of the loss of a single packet (a sequence number break of exactly one). A set `S` bit in the header indicates the last packet sent is a guard packet; lost guard packets can be ignored because their MIDI command payloads are empty.

At mid-level, a channel journal holds recovery information for a single MIDI channel. It starts with a 3-byte header, followed by a list of leaf elements called *chapters*; a chapter holds recovery information for a single MIDI command type.

Each chapter type is associated with a letter (for example, chapter W protects the MIDI *PWheel* command). A flag bit for each chapter appears in the *table of contents* (TOC) located in the final byte of the channel journal header. Chapters whose TOC bits are set are present in the list following the header, in the order of their appearance in the TOC.

Figure 3 shows the format for each chapter. The simplest chapters (P, W, and T) have a fixed width, and hold the data bytes for the last MIDI command sent since the checkpoint packet. For example, Chapter W protects the *PWheel* command; the `FIRST` and `SECOND` fields of chapter W code the data bytes of the most recent *PWheel* command.

The most complex chapter is Chapter N, which protects *NoteOn* and *NoteOff* commands. It includes a list of 2-byte *note logs*, which describe all (non-zero velocity) *NoteOn* commands sent since the last checkpoint packet. The `LENGTH` field indicates the number of note logs, and the `Y` bit indicates that the *NoteOn* coded by this note log has recently occurred. Receivers use the `Y` bit to help determine whether to play or skip lost *NoteOn* commands.

Chapter N also includes a bitfield structure to code the *NoteOff* commands sent since the last checkpoint packet. It consists of $\texttt{LOW} + \texttt{HIGH} + 1$ bytes that follow the note logs; the

Figure 3: Chapter Formats

```
Chapter P (PChange command):

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|   PROGRAM   |C| BANK-COARSE |F| BANK-FINE   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Chapter W (PWheel command):

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|    FIRST    |R|    SECOND   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Chapter T (CTouch command):

 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|S|   PRESSURE  |
+-+-+-+-+-+-+-+-+

Chapter N (NoteOn and NoteOff commands):

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 8 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|B|   LENGTH    |  LOW  | HIGH  |S|   NOTENUM   |Y|  VELOCITY   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|   NOTENUM   |Y|  VELOCITY   | ....                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   BITFIELD    |   BITFIELD    |      ....     |   BITFIELD    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Chapter A (PTouch command):

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 8 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|  LENGTH     |F|   NOTENUM   |R|  PRESSURE   |F|   NOTENUM   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R|  PRESSURE   |  ....                                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Chapter C (CChange command):

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 8 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|S|  LENGTH     |F|    NUMBER   |R| VALUE/COUNT |F|   NUMBER    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R| VALUE/COUNT |  ....                                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

condition $\texttt{LOW} > \texttt{HIGH}$ codes an empty bitfield. A set bit corresponds to a sent *NoteOff* command; the most significant bit of the first `BITFIELD` byte codes note number $8 \times \texttt{LOW}$. The `B` bit serves as the `S` bit for the bitfield structure.

Chapters A and C protect the *PTouch* and *CChange* commands, respectively; these chapters are borrow the list structure of Chapter N. In these chapters, the `F` bit serves as the `S` bit for a list element. The recovery procedure for certain *CChange* commands uses the *number* of times a command has been sent since the last checkpoint packet, thus the `VALUE/COUNT` field name in Chapter C.

A.2 Sending Recovery Journals

Section 7.1 in the main text describes the process of sending recovery journals at a high level of abstraction: the journal for packet P_i is a modified version of the journal for packet P_{i-1}, and these modifications reflect the MIDI commands sent in packet P_{i-1}.

The format description in Appendix A.1 adds detail to this description. The MIDI command in P_{i-1} may result in the addition of a new channel journal or a new chapter, or may result in the alteration of an existing chapter. Conversely, when an RTCP receiver report is processed, chapters and channels may be trimmed from the journal.

A.3 Receiving Recovery Journals

As described in Section 7.2 in the main text, a receiver parses a recovery journal when a packet loss has been detected, and schedules MIDI commands on the local Structured Audio engine to recover from the lost packet(s).

In the common case of a single lost packet, the receiver can use the `S` bits of the recovery journal format to locate the chapter elements that protect the lost packet. The receiver then compares these elements with data structures that hold the current MIDI state of the local Structured Audio engine, and decides on the appropriate recovery strategy, in conjunction with the `late` flag described in the main text.

For example, if the lost packet contained a *NoteOff* command, the receiver analyzes the `S` and `B` bits, and determines that the bitfield structure held the lost data. The receiver then looks at its local MIDI state, to see if a set bitfield matched a current *NoteOn* command active on the channel. If a match is found, the receiver deduces a "stuck note" is in progress, and executes a *NoteOff* command to recover.

When recovering from a multi-packet loss the recovery journal is parsed completely, and comparisons are made with local MIDI state to find all potential anomalies.

A.4 Recovery Journal Bandwidth

In this subsection, we perform a simple bandwidth analysis of the recovery journal system, by examining limiting cases. We model the MIDI source as a 61-key piano keyboard, that generates data on a single MIDI channel.

We assume the musician is playing with both hands at a quick tempo, producing *NoteOn* and *NoteOff* commands across the entire keyboard (note numbers 36-96), along with the occasional *PChange* command (to change timbres) and *CChange* commands using 5 controller numbers (to code events such as sustain pedalling and volume adjustments).

Given this model, we estimate the musician generates about 20 MIDI commands per second, implying an RTP packet is sent on average every 50 ms. At this packet rate, guard packets are rarely sent; to simply the analysis, we assume guard packets are never sent. The average bandwidth of the MIDI command section of RTP packets is 640 bits/s.

To begin our analysis of recovery journal bandwidth, we consider the *open-loop* case, where RTCP feedback has been disabled entirely. In this situation, the recovery journal gradually expands to a limiting size, as the musician exercises the entire keyboard and performs other actions. We calculate the limiting size via the following observations:

1. The limiting journal contains a journal header (24 bits) and a channel journal header (24 bits).
2. The channel journal contains a Chapter P coding the last timbre change (24 bits) and a Chapter C coding changes on five controllers (88 bits).
3. Eventually, all 61 keys are touched, and Chapter N contains a bitfield structure that spans note numbers 36-96 (72 bits). On average, we assume 4 keys are depressed at a time, and so Chapter N contains 4 note logs (64 bits). Chapter N also uses 16 bits for its header.

These observation yield a journal size of 312 bits, and a total payload size of 344 bits (MIDI command and recovery journal). The payload data rate is 6880 bits/s (344 bits/packet $\times$ 20 packets/second). Note that in an absolute sense, the packetization is reasonably efficient even without RTCP feedback: a 6.88 kb/s rate is comparable to the data rates of modern voice codecs.

Estimating the data rate improvement provided by RTCP feedback is complex in the general case. Here, we discuss the simple case of a single receiver and a fixed RTCP reporting interval of 5 seconds. We assume that RTP and RTCP packets do not cross in flight: the recovery journal starts in an empty state, grows in size over 5 seconds as RTP packets are sent, and empties when an RTCP receiver report is received.

In this scenario, the recovery journal grows in size as 100 RTP packets are sent over 5 seconds. We calculate the journal size for the 100th RTP packet, by changing our open-loop analysis in the following ways:

1. Timbre changes are rare $\rightarrow$ no Chapter P.
2. Over a 5 second interval, Chapter C will code one controller number (chapter size of 24 bits), since most controllers are rarely adjusted.
3. Over a 5 second interval, *NoteOff* commands are probably not executed for both the highest and lowest range of the piano keyboard. We estimate the Chapter N bitfield structure size to be 56 bits, 22% smaller than the 72 bit maximum size for a 61-note keyboard.

These estimates yield a maximum journal size of 208 bits. If we assume worst case scenario of the earliest MIDI commands expanding the journal to full size, we estimate the payload bandwidth for the packetization with RTCP feedback to be 4800 bits/s, a 30% reduction from the open-loop bandwidth estimate of 6880 bits/s.

B. DETECTING LATE PACKETS

In this Appendix, we describe how the NMP client decides if an incoming packet is late. The client models the RTP timestamp value an incoming packet should have if it arrives on time. When a new packet arrives, the client compares the packet timestamp t_p against the model time t_m. If

$$t_m - t_p > \texttt{maxlate} \qquad (B.1)$$

the packet is considered late, and the flag variable `late` is set. The `maxlate` value controls the tolerance of the system to late notes; the current NMP client uses a value of 40 ms.

To model t_m, we record the arrival time of the first RTCP sender report as t_f, and record the RTP timestamp field in the sender report as t_o. To evaluate t_m at some later time t, we compute

$$t_m = (t - t_f) + t_o. \qquad (B.2)$$

When subsequent RTCP sender reports arrive, we use Equation B.1 to check the RTP timestamp field of the report. If the check indicates an on-time arrival, we use the sender report to update the model parameters t_f and t_o.

This algorithm is tolerant to clock skews between sender and receiver, and avoids t_m model corruption due to late RTCP sender reports. However, an unforeseen event (such as a network routing change or a large clock slip) could corrupt the model, causing all packets to be marked as late.

To safeguard against this condition, the client resets t_f and t_o if all incoming packets during an interval `late_window` are marked late. The current NMP client uses a `late_window` value of 3.5 seconds.