Regular Paper

Music Server System
—Distributed Music System on Local Area Network—

TATSUYA AOYAGI* and KEIJI HIRATA**

This paper describes the system architecture and design consideration of a *music server system*. The music server system is a platform for music applications under a distributed environment. It is designed by the server-client model, and offers special functions dedicated to musical applications.

The music server system is implemented on a workstation and a personal computer connected via a local area network. The system is made of several processes: a music server, a clock client, a device-driver client and application clients. They cooperate using the inter-process communication with a special protocol.

Model of the music server consists of the following abstractions: *event*, *track*, and *time map*. An event is defined as a pair of an action and time at which the action is issued. A track holds events, fires events punctually and converts server time to track time according to the time map.

The network delay and the process switching overhead may not only increase the response time of the system but also degrade the time accuracy. The algorithm basically incorporates deadline scheduling and event buffering for keeping the time accuracy high enough. We have evaluated the buffer delay and investigated the dominant parameters.

1. Introduction

Nowadays, many computer music systems are available [2, 5, 6, 9, 14, 15]. Almost all of those systems are built on personal computers (such as IBM-PC and Macintosh) with MIDI (Musical Instrument Digital Interface) [12]. Although these systems are intensively used for commercial recordings, research, or hobbies, we think that those systems have several serious problems. Two major ones are:

- poor computing capability and
- a low interface-level.

Indeed, many computer music applications (such as sound synthesis, sound analysis, improvisation generation, and human-machine ensemble) require a large amount of calculation. Thus, huge computing power is necessary to realize these applications and personal computers cannot provide such power. Moreover, many computer music applications have an inherently concurrent nature. For instance, when you write a program for a human-machine ensemble, it is natural to exploit concurrent processes to represent these musical activities: listening, score tracking, beat prediction, performance generation etc. Unfortunately, however, almost all major personal computers are single-process machines. Thus, the problem is the lack of both sufficient computing power and a multi-process capability.

The second problem is the low interface-level. Many computer music applications are built directly on MIDI-driver routines (Fig. 1) [4, 8]. Consequently, the highest common interface to musical instruments is the MIDI protocol level (almost the same as a hardware level). However, since the MIDI protocol level is too low for musical applications, this causes problems with flexibility and extensibility. For example, if we want to use another musical interface, such as a digital audio interface (DAI), it is difficult to integrate the interface for a new device with current application programs. So, a layer to abstract the sound of musical instruments and to provide a higher-level interface to musical applications is indispensable (Fig. 2).

Our idea to solve the first problem (poor computing

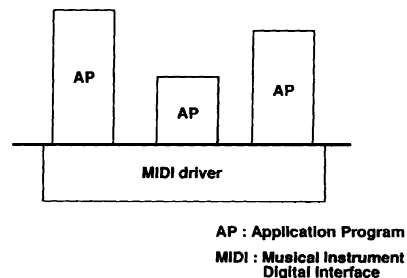


Fig. 1 Directly on MIDI Interface.

*The University of Electro-Communications.
**Institute for New Generation Computer Technology.

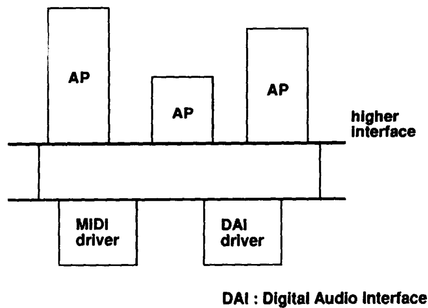


Fig. 2 High Interface Level.

capability) was to build a computer music system as a functionally-distributed system, i.e. a connection of personal computers and larger computers (e.g. workstations, minicomputers, and supercomputers) via a network. A workstation[1] has a large computing power and multi-processing capability, but lacks a realtime-control facility (in particular, UNIX-based computers suffer from this). Moreover, there are few workstations with a hardware interface to MIDI, a digital audio interface and so forth. On the other hand, it is easy to control the hardware interface of a personal computer in realtime. Thus, we expect that the combination of a personal computer and a workstation can satisfy the above requirements.

This distributed approach has the additional advantages of high availability and high extensibility. Workstations and personal computers connected via networks are very popular and easy to establish. This means that our approach requires no special hardware. So, any one can easily build our system, and use it as a platform for computer music research. Further, a distributed system can bring us high extensibility. That is, if a faster computer becomes available on a network somewhere else, it would be easy to incorporate it into our system via the network.

To cope with the second problem (a low interface-level), the server-client model is adopted. This model is suitable for distributed environments. Of course, the server-client model does not answer the second problem. Here, the important point is to define a clear interface between the server and clients. The server-client model encourages us to design a higher level interface independent of hardware.

The main purposes of our research [10] are:

- to define appropriate interfaces between music servers and clients, and
- to develop a mechanism which guarantees realtime property on a hybrid system of workstations and personal computers.

This paper presents the design and implementation of a distributed music server system; we call it the *Music Server* system. The structure of this paper is as follows. Section 2 describes the external interface of Music Server, and section 3 gives the internal design of Music Server. These two sections concern the research purposes listed above. A prototype of the Music Server system and its sample sessions are presented in section 4. Section 5 discusses several design considerations, and section 6 concludes this paper.

[1]We use *workstation* to mean any large computer.

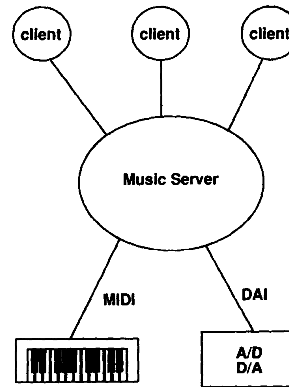


Fig. 3 Global Configuration.

2. External View of Music Server

2.1 Global Configuration

Figure 3 shows the global configuration of our distributed music system. The center of the system is Music Server. It accepts requests from clients and provides services. At the lowest level, most requests cause data transfers between Music Server and musical instruments. That is, Music Server sends/receives music data to/from musical instruments with precise timing as specified by the client's request.

2.2 Abstractions

Since the data transfer level is too low to be controlled directly by clients, Music Server has to provide an interface to the client at a more abstract level. Therefore, Music Server supports the following abstractions: an *event*, a *track*, and a *time map*. An event is defined as a pair of an action and the time at which that action is issued. You may think of an event as abstraction of the data transferred to/from musical instruments, such as MIDI, DAI, SMPTE (Society of Motion Picture and Television Engineers) time code. Since there are so many events in our music system, a grouping mechanism is indispensable. The track abstraction plays this roll. A track holds events. Roughly speaking, a track is analogous to one track of a multitrack tape or one staff of a music score. Each track has its own clock. On some tracks, time may proceed fast, while on others

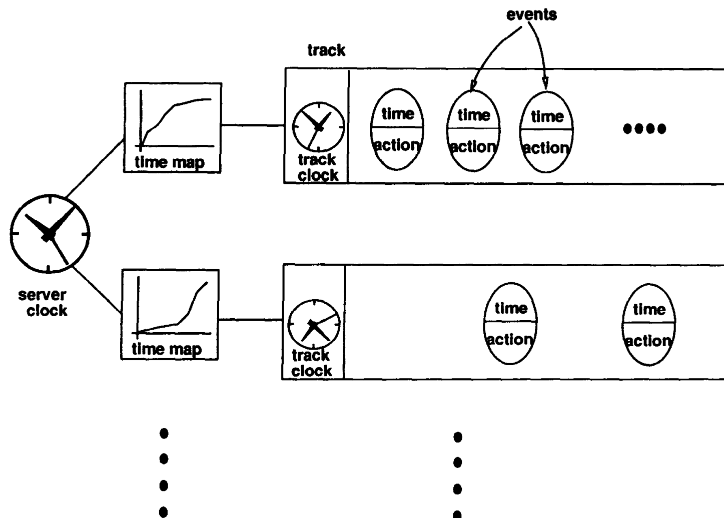


Fig. 4 Abstractions: Events, Tracks and Time maps.

it may not. So, the third abstraction, a time map, is necessary [11]. A time map associates track time (virtual time) with server time (real time). Each track has its own time map, and its clock proceeds dependent on that map. Music Server activates events on tracks according to the track time. Figure 4 shows these abstractions.

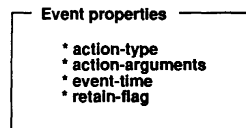


Fig. 5 Event Properties.

2.2.1 Event

Basically, an event is a pair of an action and the time. Figure 5 shows the properties of an event.

There are several kinds of actions specified by the property `action-type`. Figure 6 shows the classification of events depending on their action types.

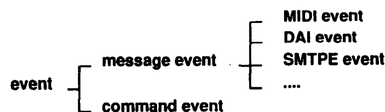


Fig. 6 Event Classification.

A message event corresponds to a data transfer which occurs between Music Server and musical instruments. A MIDI event is a special message event in the case where a musical instrument is a MIDI instrument. When a MIDI event on a track is activated by Music Server, MIDI data included in the event's property, `action-arguments`, is sent to MIDI. When data comes from a MIDI interface to Music Server, a MIDI event is stored on tracks.

A command event is executed inside Music Server. All operations which Music Server accepts (see Section 2.3) can be realized as command events, as well. For instance, command events can create a track, manipulate the time map, or delete specified events.

2.2.2 Track

Figure 7 shows the properties of a track. A track holds events (`list-of-events`), its own time map (`time-map`), and the track time (`track-time`).

Fig. 7 Track Properties.

The `play-flag` and `record-flag` control data transfer to and from the track. If the `play-flag` is on and message events on the track are activated, the data are sent to the musical instrument. If the `record-flag` is on and data come from the musical instruments, a message event is put on the track. This implies that, if there are many tracks with `play-flags` that are on, data transfer from those tracks to the musical instruments may occur at the same time. Also, if there are many tracks with `record-flags` that are on, incoming data are copied for all the tracks.

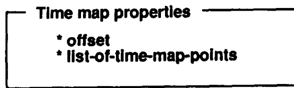


Fig. 8 Time Map Properties.

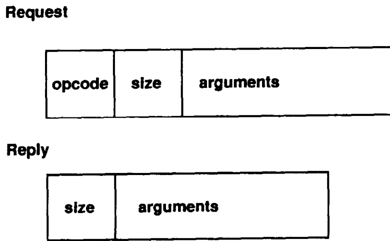


Fig. 9 Message Format.

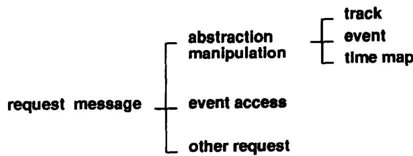


Fig. 10 Message Categories.

2.2.3 Time Map

Each track has its own time map, which associates the track time (virtual time) T_t with server time (real time) T_s. The time map consists of a (strictly) monotonically increasing function of time $f(t)$ and an offset b. The relationship of these parameters is represented by $T_t=f(T_s-b)$. The function must be monotonically increasing. Since Music Server requires the inverse of the function when calculating track time.

Figure 8 shows the properties of a Time Map. In the current prototype, the function f is given as a sequence of differential terms dT_s/dT_t. These terms represent the reciprocal of the event-execution rate.

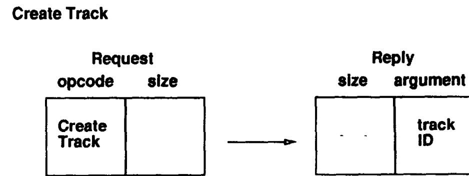
2.3 Protocol

A client requests a service to Music Server with inter-process communication. We assume a reliable byte-stream connection. Each request message forms a packet as shown in Fig. 9. Some request may have a reply message from Music Server. The request messages which Music Server can accept are categorized as shown in Fig. 10.

2.3.1 Abstraction Manipulation Request

Music Server provides the following functions:

- create/remove an abstraction (track, event, or time map)
- get/put the value of a property of an abstraction.



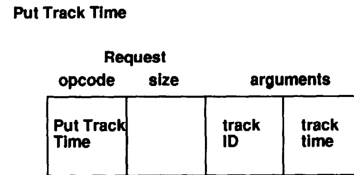


Fig. 11 Message Examples.

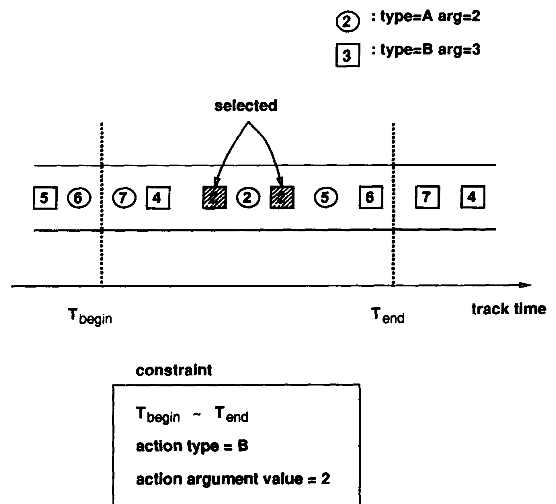


Fig. 12 Event-access Method.

When Music Server creates an abstraction, Music Server returns its identification number, which identifies the newly created abstraction, in a reply to the creation request. Client uses this identifier in subsequent request messages to manipulate the abstraction. Figure 11 shows examples of the messages.

2.3.2 Event-Access Request

Clients can access events using the identifier. It is hard, however, for clients to manage every event with its identifier, especially since the number of events is quite large. Thus, Music Server provides an alternative event-access method other than access by identifier; that method uses the properties of events.

In this method, a client specifies constraints on each property of an event. Music Server selects only the events that simultaneously satisfy all constraints. Then,

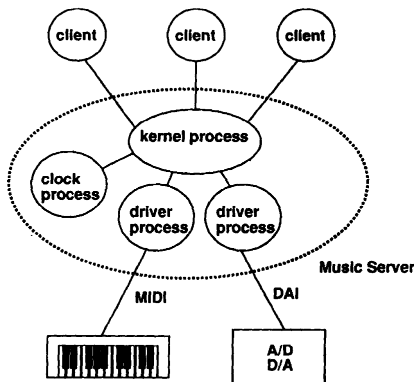


Fig. 13 Process Configuration.

the selected events are obtained/deleted according to the request message. The constraint consists of the following components:

- •Track-time Range ($T_{begin} \sim T_{end}$)
- •Action Type
- •Action Arguments

The constraint of the track time is a time range. In terms of the action type and the action arguments, their constraints are examined by equality to the arguments of the event-access request (i.e. exact match). Figure 12 shows an example of selection by the event-access request. For convenience, wildcards can be used, too. However, there are some patterns which this method cannot represent, such as events which does not satisfy a constraint. Further investigation is needed on expressiveness and readability.

2.3.3 Other Requests

In addition to the request messages described above, Music Server accepts the following requests:

- •Session
  Messages to open/close the connection to Music Server.
- •Global Information
  Messages to get information on Music Server, e.g. the message to get all the identifiers of available tracks.

3. Inside Music Server

We will focus on how to realize realtime control in the Music Server system.

3.1 Process Configuration

The Music Server system consists of a kernel process, a clock process and a driver process (Fig. 13). Interprocess communication occurs between the clock process and the kernel process, and between the kernel process and the driver process. This process configuration is statically determined at system startup. There are several types of messages transferred, such as request and reply.

The clock process has a real clock driver. The clock process is capable of issuing a special message (a ticktack message) every I_P *milliseconds*, that will increment the clock in the kernel process.

The kernel process periodically receives ticktack messages and increments its internal clock. Thus, the kernel process can synchronize the clock process. Upon reception of the ticktack message, the kernel process computes the activation time of each musical event, selects the events to be activated during the interval ($now \sim now + I_P$ *milliseconds*), and, then, transmits the events to the driver process.

The driver process works as the interface to musical instruments, connected externally. The driver process periodically receives a bunch of events. Since events are composed of an action and time as described earlier. The driver process performs the action exactly at that time, according to a clock within the driver process. Moreover, the driver process can receive musical events from the external musical instruments any time, too. These events are stamped with the receiving time, buffered, and sent together to the kernel process at appropriate regular intervals.

If you want to realize a musical application, you build a musical-application client, first. Then, the client makes a connection to Music Server, that is, the kernel process. The musical application exchanges various information with the kernel process to realize expected functions.

3.2 Two-level Realtime Control

The Music Server system adopts functionally-distributed approach as described in Chapter 1; the realtime capability of a personal computer compensates for the lack of a workstation. We can assume that the clock process and the kernel process run on workstations, and the driver process runs on a personal computer. To guarantee the realtime property of the whole system, our system introduces two-level time resolution management: coarse and fine grains.

The clock and kernel processes have a coarse grain time resolution since the time resolution of the clocks in those processes depends on the accuracy of a workstation's timer routine and the process switching overhead. If the interval of the ticktack messages is I_P milliseconds, the tolerance of the interval is $\pm I_P/2$. On the other hand, the driver process has fine grain time resolution since its time resolution is determined by the personal computer timer routine. As a result, the system can achieve time resolution to the order of one millisecond.

The driver process has two buffers for playing and recording, a *play-buffer* and a *record-buffer*. When playing musical events (output), first, the clock and kernel

processes on workstations perform the coase-grain realtime control based on the deadline scheduling. Second, the more accurate realtime control is done on a personal computer. Further, the driver process stores the events using time window in the play-buffer to cancel the latent fluctuation of the event-transmission path. When recording musical events (input), the flow of musical events is rather simple. First, the driver process on a personal computer stamps them with the accurate internal clock, and stores them in the record-buffer. Second, these received events are transmitted to the kernel process periodically at a coarse-grain interval.

3.2.1 Deadline Scheduling

The kernel process basically adopts the deadline scheduling [1, 3, 13] to pick up the events to be activated. Our deadline scheduling algorithm finds the events with the earliest performance time. At that time, the real time is converted to the virtual time through the time map, and the algorithm works in virtual time. If the time advances, there might be more than one event which has to be performed within the duration. Since the time resolution of the kernel process is too coarse, our algorithm does *not* round up or down the performance time of each selected event, *but* calculates the performance time at the time resolution of fine grain even in the kernel process. Next, the virtual performance time is converted to the real time again, and those events are sorted in chronological order. Accordingly, while our algorithm obtains a bunch of the soonest events at the time resolution of coarse grain, each event separately has fine grain fine resolution performance. The driver process performs these events at the finer resolution, of course.

3.2.2 Buffering by Time Window

To absorb the deviations of network delay, process-switching overhead, and the calculation time of the kernel process, the driver process contains the play-buffer that receives messages from the kernel process. This buffer is managed by a time window (margin time): constantly buffering the events for more than W milliseconds from the actual moment (Fig. 14). That is, an amount of messages to be buffered is determined not by the number of the messages, but by the time window to be buffered. For instance, if the time of the soonest event is not in the time window, there are no events in the buffer. However, if the activation time of more than one event is within the time window, all of the events are stored in the buffer. In other words, the clock of the kernel process is advanced by the width of the time window of the driver process. Therefore, if the above deviations fit in the time window, it is guaranteed that the Music Server system works well. However, the buffering by the time window inevitably causes buffer delays.

3.3 Latency Analysis

This section examines the latency of the system.

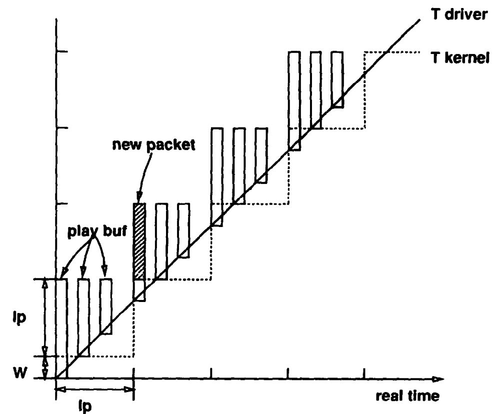


Fig. 14 Buffering in Driver Process.

When playing, the latency is the interval between the moment of posting a musical event to the kernel process and the moment of performing the musical event by the driver process. The sources of the latency are:

- the interval at which the ticktack messages are regularly issued, I_P,
- the buffer delay of play-buffer, W,
- network propagation,
- message composition/decomposition.

Without taking the third and the fourth factors into account, we can easily see that the minimal latency vibrates between W and $W+I_P$ (Fig. 15), and the arrival of a ticktack message can be deferred by W at most.

When recording, the latency means the interval between the moment an musical instrument is played and the moment the kernel process receives the performance data from the instrument. The record-buffer is flushed every I_R milliseconds, and the flushed events are sent to the kernel process together. The causes of latency are:

- the interval at which the driver process regularly transmits the recording data, I_R,
- network propagation,
- message composition/decomposition.

Also without taking the second and the third point into account, we can easily see that the minimal latency vibrates between 0 and I_R, similar to natural latency in playing.

4. Prototype System And Sample Sessions

4.1 Prototype Configuration

Figure 16 shows the prototype configuration of the Music Server system, which consists of a workstation (SUN3/280) and a personal computer (NEC PC9801) connected via a local area network (Ethernet). The per-

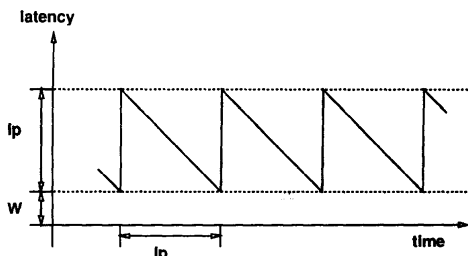


Fig. 15 Latency Vibration.

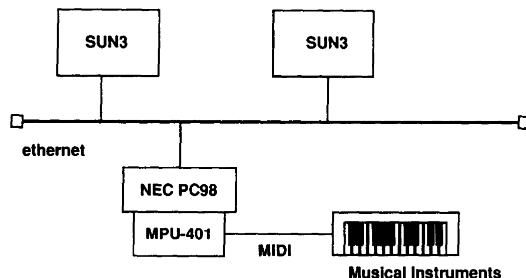


Fig. 16 Prototype Configuration.

sonal computer has a MIDI-interface board (MPU-401). The kernel and clock processes run on the workstation, and the driver process runs on the personal computer. For simplicity, we implement only the MIDI drive process currently, and not the DAI drivers yet. Musical applications may run on any workstation or personal computer in the network.

The current Music Server is written entirely in the C language. The kernel process has about 2500 lines, the driver process about 1500 lines, and the clock process about 300 lines. To write application client programs in prolog, foreign predicate routines written in C (500 lines) and SICStus Prolog (700 lines) [7] are provided.

We now mention the latency parameters introduced in Section 3. In the current prototype of the Music Server system, I_P is set at 300 milliseconds, W at 150 milliseconds, and I_R at 200 milliseconds. The time spent for network propagation and message composition/decomposition is evaluated to be an average of 30 ~ 40 milliseconds. The fluctuation of the time is usually within 10 milliseconds, although this value depends on the running conditions and the network configuration.

4.2 Sample Sessions

This subsection demonstrates how to play and record musical events while using the interface provided by Music Server.

4.2.1 Play

Let us give you a sample session of playing musical events. First, you create an application client and make a connection to Music Server. The application client issues the following commands in turn:

(1) create a track
A new track with its `play-flag` off is created in Music Server. The identifier of the newly created track returns.

(2) post musical events
The client can post musical events to be performed in any order by indicating the target track. The identifiers of the musical events posted may return.

(3) set a time map
A time map is combined with the target track. The time map can be newly allocated or can be reused among the existing time maps. The client can modify the contents of the time map as well.

(4) turn on the play-flag
When the `play-flag` is on, the track is active. Music Server is executing the musical events one by one along with advancing the time.

(5) post musical events
Any musical event whose time is after the current time of Music Server, will be executed eventually. Events whose times are before the current time, however, are never executed and are simply held in Music Server.

The clients, of course, can own more than one track, and turn on more than one `play-flag` simultaneously. In such a case, every active track sends musical events. Moreover, clients can dynamically toggle the `play-flag` and alternate between play and pause.

4.2.2 Record

The procedure of recording is almost the same as playing. Next, we show you a sample session of recording musical events. First, you create an application client and make a connection. Then, the application client issues the following commands in turn:

(1) create a track

(2) set a time map
Do the same as you would to play (see above).

(3) turn on the record-flag
When the `record-flag` of the track is on, the track is active. Music Server alternatively outputs received data in the active track and time stamps it.

If more than one track is active, Music Server duplicates the data for each active track and puts the data into each track. The clients can dynamically toggle the `record-flag` and alternate between record and pause.

Both the `play-flag` and `record-flag` can be on at the same time. Then, while performing the current events

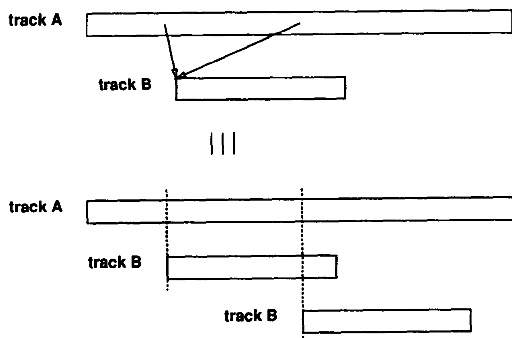


Fig. 17 Hierarchical Tracks.

one after another, Music Server receives incoming data from the external instruments.

5. Problems and Future Research

5.1 Hierarchy of Tracks

Usually, music forms a hierarchy: e.g. form, part, period, phrase, chord, note. Thus, it is natural that we introduce a hierarchy into our model. That is, we think that it is useful for tracks to be able to hold not only events but also tracks. In the current model, a track holds only events.

In Fig. 17, track *A* holds track *B* twice at different time. This track structure means a repeated phrase. In the current model, a track has its own clock and time map. When realizing the shared and hierarchical tracks, there might be more than two current points on a track simultaneously. Since shared track *B* needs multiple clocks and multiple time maps, we have to modify our model in order to manipulate those multiple time control.

5.2 Time Map

Although the time map concept has been found to be essential and very useful, it still has several problems.

Firstly, a more powerful timing mechanism is needed. That is, there are time progression patterns which the time map cannot express: expressing a vibration of infinite duration, expressing reverse time progression, and implementing tapedeck operations (e.g. fast-forward, rewind, and pause), to name a few.

Next, there is the question of at what level the time map should be supported. The purpose of the time map mechanism is to control the event-execution rate. It is possible to support it at higher application program levels, that is, at the application client level. It is thought to be the lowest level that Music Server supports the time map. The optimal point between redundancy and functionality must be investigated further.

5.3 Action Express

In the current design, where several processes are synchronized by interprocess communications, it may not be sufficient to catch realtime events and to respond to them quickly (within a very short term). To solve this problem, we are developing a technique called an *action express*. If an action (an event with no time) with an action express tag is posted in Music Server, Music Server transparently relays the action to the driver process. This processing of an action express can be viewed as an execution with the highest priority. As soon as the driver process receives the action express, the action is written to the MPU board with zero waits. Thus, the time when an action express is actually executed is determined by the sum of the response time of Music Server, the network delay, and the response time of the driver process This sum is much smaller than the buffer delay, which was evaluated in Section 3.3, but fluctuates with current conditions.

6. Concluding Remarks

The design and the architecture of the Music Server system were described. We have proven that the two-level realtime control works well under the condition of usual system load in terms of network traffic and processes. Since our realtime control mechanism is rather simple, we can predict the system latency. Starting from the Music Server system, we will investigate a distributed musical system more suitable for real-time use.

From the experience gained through the development of more sample applications, the functionality of Music Server's command structure must be examined and refined.

Acknowledgement

We would like to thank Mr. Shigeki Goto, Mr. Yutaka Ogawa, Mr. Hirohide Mikami, and Mr. Toshi Takada for their helpful advice and valuable discussion.

References

1. ANDERSON, D. P. and KUIVILA, R. Accurately Timed Generation of Discrete Musical Events, *Computer Music Journal*, **10**, 3 (1986).
2. ANDERSON, D. P. and KUIVILA, R. A System for Computer Music Performance, *ACM Trans. Comput. Syst.*, **8**, 1 (1990).
3. ANDERSON, D. P. and KUIVILA, R. A Model of Real-time Computation for Computer Music, *Proc. of International Computer Music Conference 1986.*
4. BOYNTON, L., LAVOIE, P., ORLAREY, Y., RUEDA, C. and WESSEL, D. MIDI-LISP A LISP-Based Music Programming Environment for the Macintosh, *Proc. of International Computer Music Conference 1986.*
5. COINTE, P. and RODET, X. Formes: an Object & Time Oriented System for Music Composition and Synthesis, *ACM Symposium on LISP and Functional Programming* (1984).
6. COLLINGF, D. J. and SCHEIDT, D. J. Moxie for the Atari ST, *Proc. of International Computer Music Conference 1988.*

7. CARLSSON, M. and WIDÉN, J. SICStus Prolog User's Manual, *SICS Research Report* R88007 (1988).
8. DANNENBERG, R. B. *The CMU MIDI Toolkit Manual*, Center for Art and Technology, College of Fine Arts, CMU (Aug. 1986).
9. DANNENBERG, R. B., MCAVINNEY, P. and RUBINE, D. Arctic: A Functional Language for Real-Time Systems, *Computer Music Journal*, **10**, 4 (1986).
10. HIRATA, K. and AOYAGI, T. Music Server, *Proc. of International Computer Music Conference 1989*.
11. JAFFE, D. Ensemble Timing in Computer Music, *Computer Music Journal*, **9**, 4 (1985).
12. Japan MIDI Standard Committee MIDI-1.0 Specification (1986).
13. KUIVILA, R. and ANDERSON, D. P. Timing Accuracy and Response Time in Interactive Systems, *Proc. of International Computer Music Conference 1986*.
14. POPE, S. T. A Smalltalk-80-based Music Toolkit, *Proc. of International Computer Music Conference 1987*.
15. THOMPSON, T. and BARAN, N. The NeXT Computer, *Byte* (Nov. 1988), 158–175.

(Received November 28, 1989; revised December 5, 1990)