GRAME - Computer Music Research Lab.
Technical Note - 01-12-06

MidiShare Server
A proposed new architecture for the MidiShare Kernel

Dominique Fober Yann Orlarey Stephane Letz
Grame - Computer Music Research Laboratory
9, rue du Garet BP 1185
69202 LYON CEDEX 01
[fober, orlarey, letz]@grame.fr

MidiShare is a portable software architecture for musical applications, based on a client/server model. Up to now and along all the supported operating systems (GNU/Linux, MacOS, Windows), it has always been implemented at low level operating system layer. This choice was dictated by efficiency and time constraints. The main drawback of using low level layers is the lack of portability and the complexity of the kernel extensions design. Recent evolutions of operating systems, combined with important technology improvements, have made possible to consider a more portable architecture for MidiShare. This document presents a proposed new architecture, based on a user level design.

1 Introduction

The proposed new MidiShare architecture is only intended to report the currently provided services on a more portable implementation. Including new services is not an issue of the present document. However, a great care should be taken to design a new architecture capable of easily supporting such extensions. According to this objective, the main problems to be solved in a user level design of the MidiShare kernel may be summarized as follows:

- the time deterministic behavior of the global system (seen as the MidiShare kernel and its clients) has to be preserved,
- the new implementation additional cost has to be bounded in order to keep the system efficiency,
- the global system consistency has to be examined according to the possible failures intoduced by the new design.

The current services are discussed below in regard of the first two points. Consistency is discussed in the implementation part.

1.1 The client / server interactions

The basic services provided by the midishare kernel through its main components have been described many times [1] [2] [3]. These services are based on a limited set of interactions between the kernel and its clients. From the client point of view, services are available through the MidiShare API ie using function calls. From the server point of view, services are provided at client request (ie in reply to a MidiShare API call) or triggered by time dependant events. This set of interactions may be summarized as follows:

- *Callback based interactions*: they run in the client memory space but are triggered by the server. They may be viewed as events notifications and we can consider two types of them:
  - synchronous event notification: made at the MidiShare time resolution (receive alarms and real-time tasks),
  - asynchronous event notification: may occur at any time, they are triggered by client applications (applications alarms).

  For events, synchronous and asynchronous qualifiers denote wether the notification is made at the MidiShare time resolution or not. This distinction is relevant in regard of possible optimizations: grouping all the notifications at MidiShare time resolution for example.

- *Request based interactions*: they correspond to the set of MidiShare function calls and may run both in the client and server memory spaces. We can consider two types of requests:
  - synchronous request : represents a function call made by a client which requires a reply from the server,
  - asynchronous request : represents a one way function ie without reply. This kind of request generally operates by side effect.

  For requests, synchronous and asynchronous qualifiers denote wether the request is synchronous to the function call or not. This distinction is relevant in regard of the requests deterministic behavior. At present, requests are handled from client to kernel space using functions which may cross several operating system layers. The most direct requests are achieved by direct kernel call (MacOS) or shared library functions (Windows). A more complex system is used by the Linux implementation where requests transport from client to MidiShare requires to switch from user mode to kernel mode. In any case, when a client calls any MidiShare API, it can assume that the call has taken effect et return.

1.2 The MidiShare events communication scheme

The MidiShare events internal communication scheme is critical from efficiency point of view. MidiShare events are at the root of the inter-applications communication services provided by MidiShare. To implement these services, access to the events data is required for both the client and the kernel. Some implementations make use of shared memory segments: implicit shared memory in case of MacOS where the memory is not protected, explicit shared memory for Windows implementations. In this case, events transmission may be achieved using simple pointers exchange. On the contrary, the Linux implementation copy events data from client to kernel memory space (and the opposite) using low level memory access functions.

The next part of this document will focus on the possible technical solutions for the client/server interactions and the MidiShare events communication. Section 3 presents a proposed new architecture, section 4 focus on its implementation and section 5 deals with the expected performances.

2 Overview of the possible technical solutions

A user-level design of MidiShare means that the MidiShare kernel will run as a standard process, without any particular privilege. We'll later refer to this process as the *MidiShare Server*. Client/Server interactions and MidiShare events internal communication are critical from time and efficiency points of view.

2.1 Client/Server interactions

They should be based on the host operating system communication layers. Local socket communication is widely supported however and due to efficiency problems on some operating systems [4], we'll consider the most per-platform efficient communication way ie:

- local socket communication on Linux,
- mach messages on MacOS X,
- Windows messaging system on Windows.

It appears that the most common abstraction among all these platforms is to consider that the client/server interaction is message based: messages may be used both for client requests (replacing the function calls) and for events notifications. In the client to server direction, the messages typology should be mapped on the MidiShare API. But in the opposite direction and for events notifications, we can consider two different solutions concerning the messages typology:

- providing different messages for the limited set of server to client notifications: receive alarms, applications alarms, tasks and driver callbacks.
- using a unique "wake up" message and extending the MidiShare events typology.

Extending the MidiShare events typology to support new interactions between the client and the server is probably the most simple way to modify the kernel behavior as such extension is platform independant.

2.2 MidiShare events communication:

As in the current implementations, we can consider two solutions for the MidiShare events internal communication:

- copying events data between client and server protected memory spaces,
- making use of shared memory.

Events copy may be achieved through the client/server communication channel. In this case, events data are written and read from buffers sent and received from the communication channel. As these buffers are additionnaly copied through the system communication layers from source to destination process, global cost of events copy is far more expensive than events reference using shared memory.

Drawbacks:

- synchronization and accuracy problems to transmit large events,
- efficiency.

Advantages:

- memory spaces are always protected,
- applicable to distributed systems.

Shared memory segments should be only available to the owner process and the server. In this case, transmission of a simple event reference is sufficient to access the events data on both sides.

Drawbacks:

- memory protection: a client may crash the server with a wrong event reference,
- doesn't support distributed systems .

Advantages:

- efficiency

3 Proposed architecture overview

The proposed architecture is shown on figure 1. It is basically based on two different software components: the MidiShare Server and the MidiShare Library. The MidiShare Server is a separate process while the MidiShare Library is mapped in the client process memory space. The MidiShare drivers are considered as separate tasks but operates in the server memory space.

3.1 The MidiShare Server

It includes two global components: the core kernel and the communication layer.

The core kernel: implements the main components of MidiShare (Memory Manager, Time Manager, Task Manager, Communication Manager, Scheduler and Ports Manager). It provides also an abstract layer for client / server communication. It is platform independant.

The communication layer: is platform dependant. Its main purpose is to isolate the core kernel from the implementation particularities. This functionnality is mainly achieved through 3 different kind of tasks:

- it provides the implementation of the abstract communication layer,
- it normalizes the client requests to ensure that the core function calls are always achieved using the same interface among all the implementations,
- it ensures that transmitted events will be always presented to the core kernel using a unique high level format.

All these tasks are of course highly dependant of the choice made for the client / server interactions and for the MidiShare events internal communication.

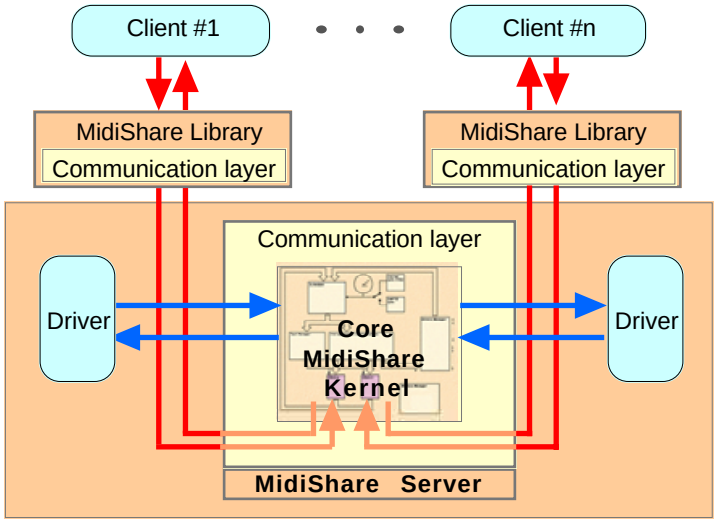

figure 1: global architecture

3.2 The MidiShare library

It provides the MidiShare API implementation which represents two different kinds of services:

- static services: to be run in the client space (like the client memory manager) and which don't require a server to run,
- dynamic services: which are provided by the server.

Distribution of the MidiShare API among this two service types are shown in tables 1 to 3. It appears clearly that whole MidiShare components may be lying in the client memory space: the memory manager for example is still implemented that way in the current Linux release.
The communication layer purpose is similar to the kernel communication layer: it isolates the client services from platform dependencies and takes in charge the particular implementation of client / server interactions and the MidiShare events internal communication.

3.3 The MidiShare drivers

The MidiShare drivers may be viewed as separate components, provided as shared libraries for example. They are mapped in the server memory space and therefore may bypass the whole communication layers. The problem is to decide wether the implementation will also support drivers as separate processes or not. If it does, the MidiShare library should then include the corresponding API and the communication layer should support the specific driver interactions (driver specific callbacks).

4 Proposed implementation

4.1 Communication layer

Global communication scheme is shown in figure 2. The communication layer design is similar to [5], it includes:

- a stub: in charge of the arguments packing and unpacking, including events buffering when necessary. It isolates the communication runtime from events internal representation.
- a communication runtime: in charge of the message transmission from process to process. It is build on top of different the operating system IPCs.

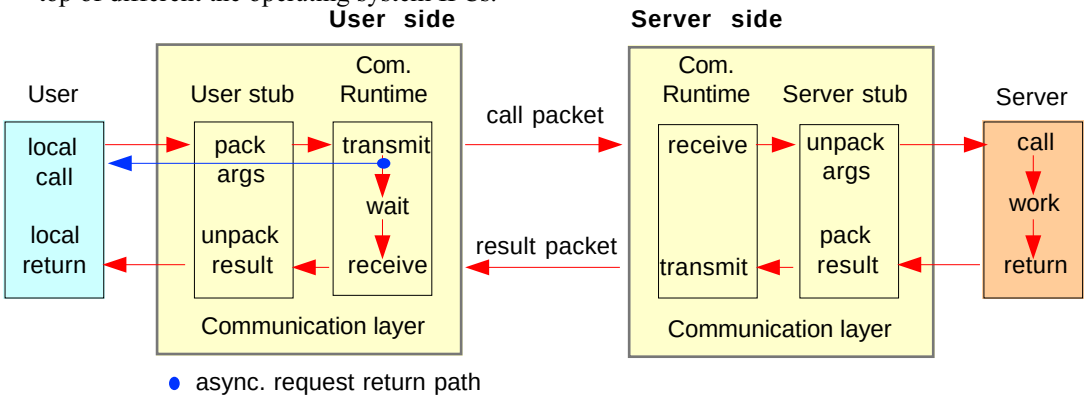

figure 2: communication scheme.

As in [5] we may consider generating the main part of the stubs automatically, starting from a high level description of the exported and imported interfaces. Static stub parts are dedicated to events linearization or de-linearization and to events references conversions.

4.2 Multi-threaded implementation

The different threads involved in the client / server interaction are shown ni figure 3 where request based interactions are in blue and callback based interactions in red. The server includes:

- a listening thread: dedicated to the incoming client requests. It guarantees that concurrent requests will be serialized.
- a time thread: in charge of the server time task. It is granted the highest priority in the system.

As with current implementations, two threads operate on the client side:

- the main thread: in charge of sending requests and receiving replies,
- the real-time thread: in charge of the application real-time tasks. Its execution is triggered by the server time thread.

All the threads operate through the communication layer and are associated with communication channels. We'll refer later to these communication channel as ports. On server side, the listening thread creates a listening port and make it publicly available to the clients. It is used to collect input requests. On client side, the listening port is collected at initialization and two additionnal ports are created:

- a reply port: associated with the main thread, it listens to the requests replies.
- a callback port: associated with the real-time thread, it listens to events notifications.

Such a design allows preemption of the requests by the Time and Real-time threads which is conformant to the current system behavior.

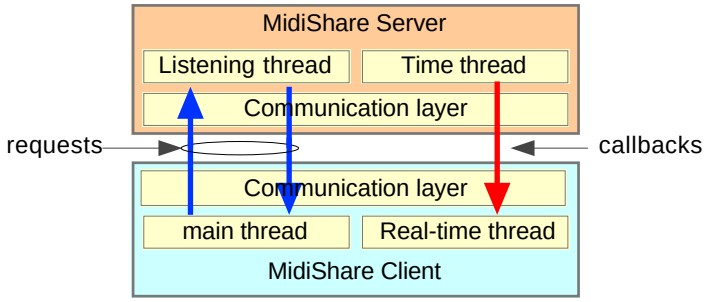

figure 3: threads involved in the clent/server interaction

`

4.3 States and transitions

From a client point of view, the server may be seen as:

- down: the server is not running because none of its services has been requested.
- up: the server is running. It corresponds to the normal state of current implementations where the server is generally loaded at system initialization time.
- unreachable: corresponds to an exception case. While client sessions are running, the server is suddenly unreachable due to either a server crash or a broken communication channel.

These views correspond respectively to 3 different client states: a passive, an active and an exception states.
It's the client responsability to activate the server when it is down. It's the server responsability to quit when there is no more active session.
Transitions between the client states are handled by a specific component as shown in figure 4.

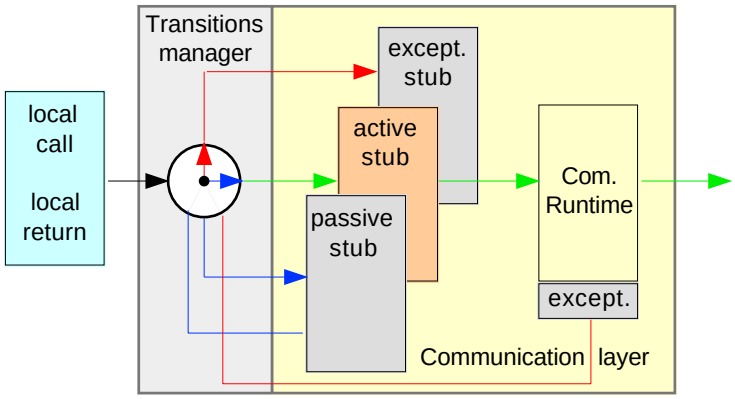

figure 4: transitions management

Every client call is directed to the component associated to the current state. The passive stub tries to access the server (and possibly launchs it). In case of success, it is replaced with an active stub which corresponds to the normal communication stub. Communication exceptions generated by the communication runtime induce the replacement of the current stub with an exception stub.

4.4 Exceptions and consistency

From client point of view, exceptions may happen when the server crash or when the communication channel is broken. Both of them should be detected by the communication runtime. The problem then is to provide a consistent behavior through the exception stub in order to let the client application take the appropriate decision.
The problem is more critical if a function never returns (the server is running an infinite loop for example): the client will be never informed of the failure. It could be solved with a time-out mechanism limiting the duration of a call. However, local procedure calls behave similarly (they have no time-out mechanism) and we may choosing a close semantic between remote and local calls.

From the server point of view, consistency may be defined as correctness of its information about clients. For example:

- at time t_1 3 clients are running and the server clients set is (A, B, C)
- at time t_2, the client B crashes but the server clients set remains (A, B, C)

then the server state is unconsistent. The problem may occur with the Macintosh and Windows current

implementations of MidiShare. As it is critical for the whole running clients, we are tempting to find a solution. In fact, as the server is linked to a client using its callback port, failure of the client may be catched by getting errors on write operations to this this port.

4.5 Time dependent behaviors

The MidiShare kernel aims to provide a completely deterministic time behavior to its clients: it means essentially that a sequence of time ordered events will be always rendered with the same scheduling. This time consistency is maintained internaly by making use of appropriate objects such as LIFO and FIFO. On a single communication channel (from server to client), this consistency should be maintained as transmissions are serialized. However, if we consider a sequence of time ordered events distributed on several clients, the problem is more complex as shown in figure 5 where considered events are MidiShare real-time tasks:

- if the kernel is allowed to directly call a task for a client (fig. 5a), then the time behavior is completely deterministic,
- if it is not allowed to do so (fig. 5b), the final scheduling is entrusted to the host operating system. This is the case with the current Linux implementation where the example below produces the following result: [A, B, C], [C, B, A], [A, B, C] etc...provided that each task reschedules itself using a common time offset.

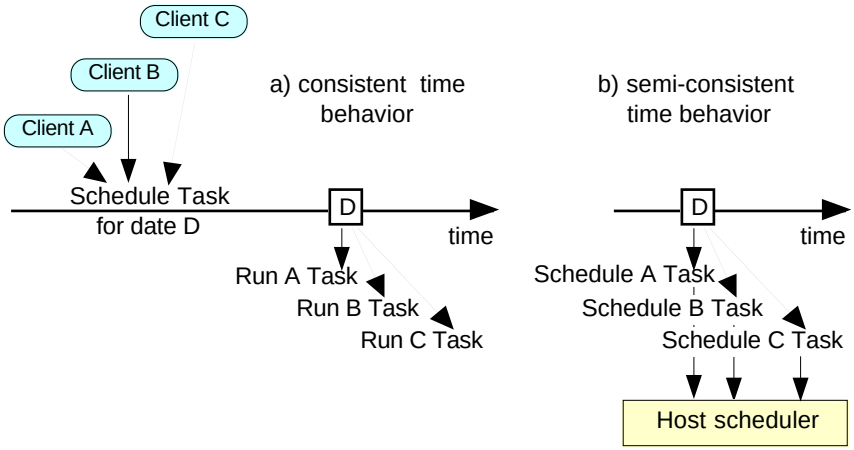

figure 5: time consistency

5 Expected performances

Experiments made on [4] show that triggering a time tasks every millisecond in a client / server environment may consume from 2 to 30 % of the CPU depending on the clients count and on the underlying operating system:

- about 2 % represents the best case: a single client running one task every millisecond on Linux, Windows 2000 or NT.
- about 30 % represents the worst case: 10 clients, each one running a separate task every millisecond on MacOS X

We have made additionnal measurements to evaluate RPC cost on different operating systems. Results are presented on table 1.

processor	OS	RPC cost (µs)	Comm. system
Pentium II 350 Mhz	Linux 2.4.3	38	sockets
	Windows 2000	25	win msg
PowerPc G4 350 Mhz	MacOS 10.1	25	mach msg

table 1: additionnal RPC cost

According to these results, the global number of RPC calls supported by the system at the MidiShare time resolution (ie each millisecond) is shown on figure 6. It assumes that each client is scheduled for execution every tick. In this context, every RPC count represents a 95% load of the CPU. As above, operating systems are associated with the processor indicated in table 1.

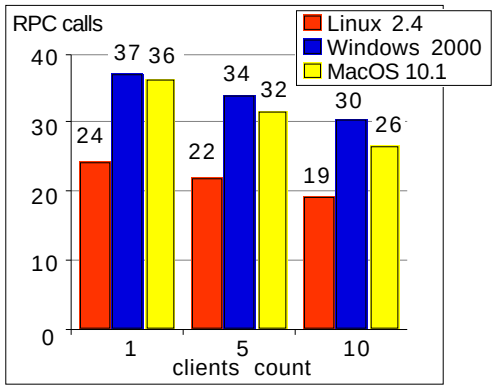

figure 6: supported RPC calls according to the clients count.

Apparently, these numbers seems to be rather limited but we have to consider them in regard of the concret use of the MidiShare API. Let's take the example of the application "msEcho" which is part of the standard MidiShare Suite: an endless echo of one note at the full MIDI data rate (ie every millisecond) makes use of 2 requests at every tick: MidiSendAt and MidiTask. Moreover, these requests may be considered as asynchronous which means that their cost is to be divided by 2. Therefore, the system could easily support 10 clients generating the same data flow.

References

[1] Y. Orlarey, H. Lequay - MidiShare : a Real Time multi-tasks software module for Midi applications - *Proceedings of the International Computer Music Conference 1989*, Computer Music Association, San Francisco, pp.234-237

[2] Grame - MidiShare Developer Documentation - Grame, 1990 ftp://ftp.grame.fr/pub/MidiShare/Documentation/MidiShare.pdf

[3] Grame - MidiShare Kernel Development Guide, Grame, 2000 http://cvs.grame.fr/cgi-bin/midishare-cvs/src/common/DevGuide/msKernelDevGuide.pdf

[4] D. Fober, Y. Orlarey, S. Letz. Real-Time IPC on a client / server model: Multiple OS performances benchmark. Technical Report #---- Grame 2001

[5] A.D. Birrel, B.J. Nelson - Implementing Remote Procedure Calls - ACM Transactions on Computer Systems, Vol. 2, No. 1, Feb. 1984, pp.39-59

Appendix A - MidiShare API Distribution

section	synchronous req.	asynchronous req.	remarks
MidiShare Environment	MidiGetVersion MidiCountAppls		using shared memory using shared memory
Application configuration	MidiGetInfo MidiGetFilter MidiGetRcvAlarm MidiGetApplAlarm		
Drivers management	MidiCountDrivers		using shared memory
Memory management	MidiNewCell MidiTotalSpace MidiGrowSpace MidiNewEv MidiCopyEv MidiGetField MidiCountFields	MidiFreeCell MidiFreeEv MidiSetField MidiAddField	
Sequence management	MidiNewSeq	MidiAddSeq MidiFreeSeq MidiClearSeq MidiApplySeq	
Time	MidiGetTime		using shared memory
Receving	MidiCountEvs MidiGetEv MidiAvailEv	MidiFlushEvs	events are pushed into the client fifo by the server asynchronously
Mail boxes	MidiReadSync MidiWriteSync		
Filters	MidiNewFilter MidiIsAcceptedPort MidiIsAcceptedChan MidiIsAcceptedType	MidiFreeFilter MidiAcceptPort MidiAcceptChan MidiAcceptType	filters make use of shared memory segments
Task Management	MidiCountDTasks	MidiFlushDTasks MidiExec1DTask	see events above

table 1: requests handled in client memory space

section	synchronous req.	asynchronous req.	remarks
MidiShare Environment	MidiGetIndAppl MidiGetNamedAppl		
Application configuration	MidiGetName	MidiSetName	
Connections management	MidiIsConnected	MidiConnect	
Sending		MidiSendIm MidiSend MidiSendAt	
Slots management	MidiGetIndSlot MidiGetSlotInfos MidiIsSlotConnected	MidiSetSlotName MidiConnectSlot	

table 2: requests handled by the server

Appendix A - MidiShare API Distribution

section	synchronous req.	asynchronous req.	remarks
Open / close application	MidiOpen	MidiClose	
Application configuration		MidiSetFilter MidiSetRcvAlarm MidiSetApplAlarm	
Task Managing	MidiTask MidiDTask	 MidiForgetTask	
MidiShare	MidiShare		

table 3: requests handled by the client and the server

section	synchronous req.	asynchronous req.	remarks
SMPTE synchronization	MidiGetSyncInfo MidiGetExtTime MidiInt2ExtTime MidiExt2IntTime MidiTime2Smpte MidiSmpte2Time	MidiGetSyncInfo MidiSetSyncMode MidiTime2Smpte	
Drivers management	MidiRegisterDriver	MidiUnregisterDriver	drivers may be considered as running in the server space using shared libraries
Slots management	MidiAddSlot	MidiRemoveSlot	as above
Task Managing		MidiCall	obsolete

table 4: requests with undefined status
(not yet distributed)