A Middleware System for Distributed Real-Time Multimedia Processing[*]

Nelson Posse Lago[1][†], **Fabio Kon**[1]

[1]Department of Computer Science
Institute of Mathematics and Statistics
University of São Paulo
`http://gsd.ime.usp.br/software/DistributedAudio`

`lago@ime.usp.br, kon@ime.usp.br`

Abstract. *The development of distributed applications for Multimedia processing with real-time and low-latency requirements faces a few key challenges. In this paper, we characterize such applications as "firm real-time applications", argue that they should be based on a callback architecture, and discuss the associated network limitations. We present a new middleware system addressing the challenges discussed and describe an application for distributed audio processing built on top of this middleware. Experimental results obtained with this application demonstrate the effectiveness of our approach.*

1. Introduction

In several computer systems for multimedia processing (such as interactive systems for the creation and edition of multimedia, particularly audio and music, or systems for pattern recognition in continuous media), it is highly desirable to be able to do the processing not only in real-time, but also with low latency. Low latency processing means that the time it takes for a change in the input data of the computer system to produce the corresponding output should be as small as possible; how small is enough, given the usual goal that the latency must not be perceptible by the user, varies a lot with the application and the user, but it seems that up to $5ms$ for the most critical interactive applications is reasonable (see a discussion on the subject in Steinmetz and Nahrstedt, 1995, p. 588–599).

In interactive systems, for instance, low latency processing serves the purpose of giving the user the illusion that the system performs the computations immediately, which is very important since the user generally adjusts his input to the computer system according to the output he receives from the system. Low latency may also be important if we want part of the data to be processed in real-time by external devices (for instance, we may want to route a previously captured audio signal into an analog effects processor and record the resulting sound without losing the timing information of the signal).

A simple example of a situation in which real-time low-latency processing is desirable is the recording of an acoustic musical instrument with some effect processing (for

[*]This research is supported in part by a grant from CNPq, proc. 55.2028/02-9.
[†]Nelson Posse Lago is partly supported by CAPES.

instance, an electric guitar processed by a custom digital distorter): while playing the instrument, the musician needs to hear the sound being produced; if the processing latency is too large, the musician will have difficulties to perform correctly.

Systems for multimedia processing in real-time, including those where low latency is desirable, usually demand large processing power from the computer system. Problems that demand large computing power (as multimedia does) are usually solved by parallel or distributed processing. However, the real-time and low latency requirements of most multimedia processing systems coupled with the need for sometimes strict synchronization between several media streams as well as the cost of multiprocessor systems have made most multimedia applications to be developed for single processor systems.

In the audio and music processing field, it is not uncommon for such systems to be coupled with dedicated, specialized hardware in order to boost the system performance. Such hardware, however, is usually proprietary and expensive: for example, a single processing board for the Pro Tools HD system costs 4 times as much as a complete mid-range desktop PC in the USA[1]. On smaller studios without access to high-end equipment, it is relatively common for the computing power during audio editing to be exceeded; when this happens, usually part of the processing is done in non-realtime mode and the processed result is saved to disk, which is inconvenient, since the possibility of interactive experimentation is lost.

Given the economic advantage and flexibility offered by general-purpose computer systems, being able to process multimedia data in a distributed system would be useful, allowing users to go beyond the performance limits of single processor systems in a more cost-effective way. Home and small music recording studios, which usually cannot afford the expensive proprietary solutions, would benefit from the use of small clusters of older and inexpensive computers to increase their processing power at a low cost.

This paper describes (1) the requirements and limitations of mechanisms for distributed multimedia processing in real-time with low-latency, (2) a simple middleware system which is a testbed for the ideas presented here, and (3) an application geared towards audio processing developed on top of this middleware. In section 2.1. we characterize systems for multimedia processing as firm real-time applications and argue, in section 2.2., that low-latency, firm real-time systems should be based on callback functions. In sections 2.3. and 2.4., we discuss the possible approaches for load distribution, address the limits presented by the networking medium, and propose a mechanism to solve them. Finally, in section 3.1., we describe the current implementation of the system and present experimental results in section 5.

2. Problems and mechanisms

When designing a method for distributed processing of multimedia in real-time with low latency, one must take into account several factors. The specific characteristics of multimedia with regard to real-time and low latency, which suggest the use of specialized real-time operating systems (such as RT-Linux – Barabanov and Yodaiken, 1996), must

[1]Prices consulted at <`http://www.gateway.com`> and <`http://www.protools.com`> in July, 2003.

be balanced with the needs for advanced user interfaces and wide hardware support, which suggest the use of general-purpose operating systems. The method by which the processing is to be distributed must be addressed, and the limitations of the networking system must be taken into account.

2.1. Firm real-time

Real-time systems are traditionally categorized as *hard real-time* systems or *soft real-time* systems. Multimedia applications are often used as examples of soft real-time systems. The definitions of soft and hard real-time systems, however, vary in the literature (Liu, 2000, p. 27–29). Hard real-time systems are usually defined as systems where a timing failure is absolutely unacceptable (it might threaten human life, for instance). On the other hand, most soft real-time systems are commonly characterized as systems where timing failures are a relatively common event and where the system is usually expected to adapt to such failures; for instance, a video playback application may skip or duplicate frames during execution if deadlines are missed[2].

While some multimedia applications (such as video players) are easily characterized as soft real-time, some others (such as interactive multimedia editing and processing tools) are not: they cannot adapt to timing failures (since that would cause audio glitches, skipped video frames etc. which might be acceptable in other environments, but are not in a professional editing tool) and, therefore, need timing precision as close as possible to that of hard real-time systems. On the other hand, sporadic failures are not catastrophic, which means that statistical guarantees of timing are enough for such a system to be acceptable[3].

Since it is not possible to deal with timing errors graciously, these systems do not need to implement sophisticated mechanisms for error correction and adaptation as most soft real-time systems do; all they need is to detect errors, which may be treated as ordinary errors by the system. During the recording of a live musical event, for instance, a failure may be registered for later editing. Reducing the quality of the recording, on the other hand, is not acceptable. During the actual editing work, a failure may simply stop the processing and present an error message to the user, who can restart the operation. Neither one of these options can be characterized as "adaptation": the failures are, in fact, treated as processing errors, not as conditions to which the system can adapt.

Finally, such systems are ideally based on general purpose computers and operating systems, because they offer low cost, advanced user interfaces, and a rich set of services from the operating system. The use of general purpose operating systems also guarantees the compatibility of the application with a much wider range of general purpose multimedia hardware equipment, since these systems usually offer software drivers for such hardware, differently from operating systems designed specifically for real-time.

[2]Several research projects and commercial products have addressed this problem, particularly in distributed systems; c.f., for instance, Chen et al. (1995); Vieira (1999); Shepherd et al. (2001).

[3]In the case of interactive editing tools, the functionality of the application is not harmed much if eventual timing errors occur, say, once every half hour. If the user starts such real-time application and it runs correctly for some seconds, it is likely that it will work correctly for longer periods. Even such empirical evidence of "timing correctness" may be acceptable to the user in this case.

Researchers have classified systems with such "hybrid" needs as *firm real-time* systems (Srinivasan et al., 1998). Such systems are characterized by being based on general-purpose computers and operating systems, having statistically reliable timing precision, and treating timing errors as hard errors with no need for adaptation. Thanks to the increasing number of firm real-time applications, general purpose operating systems such as Linux and MacOS X have been greatly enhanced to offer good performance for this kind of application. With the proper combination of hardware and software, Linux, Windows, and MacOS are capable of offering latency behaviour suitable for multimedia: from 3 to 6 milliseconds (MacMillan et al., 2001).

2.2. Callback functions

Although we advocate the use of general purpose computers and operating systems, the usual read/write mechanism offered by them for I/O is not adequate for low latency processing. This mechanism depends on buffering at the operating system level, and such buffering increases the processing latency of the system by a significant amount.

In the audio processing field, this situation has been solved, under Windows and MacOS, by the ASIO specification `<http://www.steinberg.net/en/ps/support/3rdparty/asio_sdk/index.php?sid=0>` and, under Linux, by the JACK system `<http://jackit.sourceforge.net>`. Both define mechanisms in which, conceptually, a user space application can register a function (residing inside its address space) that is responsible for handling the I/O data. When an audio interrupt is generated by the audio hardware, the kernel interrupt handler writes and reads the data to and from buffers inside the application's address space and calls the user space function that was registered to produce and consume data before the next interrupt. In this way, the application can process input data as soon as it arrives and can output processed data as soon as it is ready, communicating with low latency with the hardware device without the need to have device-specific code inside itself. This method of using *callback functions* for low latency I/O operations can be seen as a transposition of the kernel space interrupt handler to the user space application, which allows for easy changing of the kind of processing that is to be carried on at each interrupt without the need for making any changes to the operating system kernel.

A very important point is that, in order to guarantee low latency operation, the application must not block in any way during the callback function; if it did, the operating system might schedule another process to run during the short period of time between each interrupt. Since Linux, with low latency patches applied, has a typical scheduling latency of about $500\mu s$ (Morton, 2001), the application would hardly be re-scheduled in time to complete the processing before the next interrupt.

2.3. Distributed processing

We are interested in the distribution of multimedia processing across a collection of computers in a network. There are several opportunities for this distribution:

- Development of new DSP algorithms capable of running concurrently;
- Parallel processing of different data streams (by allocating different streams to different machines);

- Parallel processing trough pipelining (the available machines are “chained” and each one processes, at time t, the data that was processed by the preceding machine at time $t-1$. The problem with this approach is that it results in an increase in the latency of the processing proportional to the number of machines in the pipeline. The maximum length of the pipeline must, therefore, be computed carefully).

In order to appraise these possibilities, we should note that multimedia processing, and specially music processing, is made of the combination of:

- multiple different processing algorithms applied to a single data stream (for example, an electric guitar may be subjected to compression, distortion, flanging, EQ and reverberation);
- multiple processed data streams grouped together (for example, 32 audio channels recorded by 32 microphones on a stage).

Developing parallel algorithms for multimedia processing could be interesting, but this solution is based on the creation of entirely new algorithms, aiming at achieving the same functionality already available with non-parallel ones. While there are computationally-intensive data transformations for multimedia that are particularly slow (to the point of not being able to be run in real-time) which could benefit from this approach, the vast majority of them (reverbs, distorters, compressors etc.) are relatively lightweight; it is the sum of several of them to process an entire multimedia composition that can exceed the computer’s capacity.

It seems, therefore, more interesting to base distributed systems for multimedia applications either on pipelining, where data is processed in sequence by several machines, or on the parallel processing of the streams, where each data stream is assigned to a CPU that performs the whole processing of the stream. Since pipelining increases the latency of the system, our approach is based on the parallel processing of different streams. It should be noted, however, that it would not be difficult to transpose this discussion (and its implementation) to pipelining and even to a combination of both.

2.4. Network limitations

Since the hardware interrupts generated by the multimedia hardware are responsible for driving the timing of the complete system, the machine that treats these interrupt requests in a distributed system must have a special role[4]. In order to process real-time multimedia data remotely, an application running on this machine must

1. receive the sampled data from the multimedia device after a hardware interrupt request,
2. send it out through the network,
3. have the data remotely processed,
4. get it back from the remote machine, and
5. output it to the multimedia device before the next interrupt (Figure 1).

[4]It is possible to have different machines perform I/O synchronously using hardware capable of operating with “word clock”, which is a hardware mechanism to synchronize multiple sound cards. This, however, relies on additional hardware on each node and, at the same time, alleviates the need to use the network for the purposes described here, since each machine is able to completely process incoming and outgoing streams of data independently. Therefore, this approach is a special case, which is out of the scope of this paper.

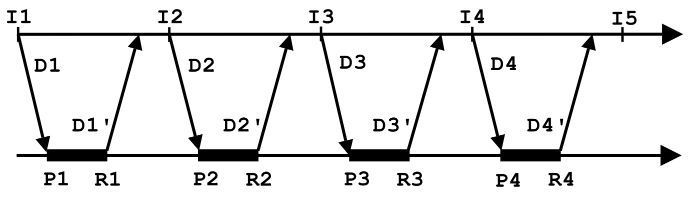


Figure 1: At each interrupt I_n, captured data (D_n) is sent out to the remote machine, processed (P_n), returned (R_n) to the originating machine already processed (D'_n), and output to the multimedia device before the next interrupt (I_{n+1}).

This mode of operation, however, uses the network in half-duplex mode, and uses both the network and the remote CPU for only a fraction of the time between interrupts. Given current commodity networking technologies (namely, Fast Ethernet), it is not hard to notice that such setup would offer the possibility of very little remote processing to be performed. If we are to seek better resource utilization, we should observe that we may distinguish three phases on the remote processing of data (for simplicity, only two machines will be considered):

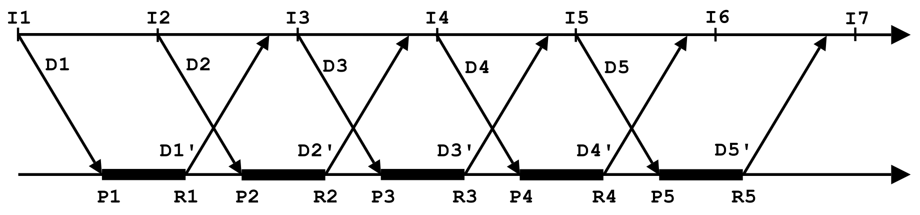
1. the data transfer from the originating machine to the remote machine;
2. the remote data processing;
3. the data transfer from the remote machine back to the initial machine.

We can benefit from a mechanism similar to the sliding windows technique used in several networking protocols (for instance, the use of sliding windows in the TCP/IP network protocol is described in Stevens, 1994, p. 280–284) to achieve better performance in a distributed application: at each callback function call (usually brought up by a hardware interrupt), the captured data is sent to the remote machine and the data that was sent out on the previous iteration and remotely processed is received. That is, the output data is reproduced one period "later" than it would have been normally, but this permits data to be sent, processed, and received back in up to two periods instead of just one (Figure 2 on the following page – a). If the amount of data is so large that two periods is not enough time to send, process, and receive the data, we may extend this method to make the delay correspond to two periods instead of one, which allows us to use three periods to perform these operations (Figure 2 on the next page – b). This is optimal in the sense that it allows us to use the full network bandwidth in full duplex and also to utilize all the processing power of the remote machine, but has the cost of added latency. These two modes of operation use the sliding windows idea with windows of size one and two respectively[5].

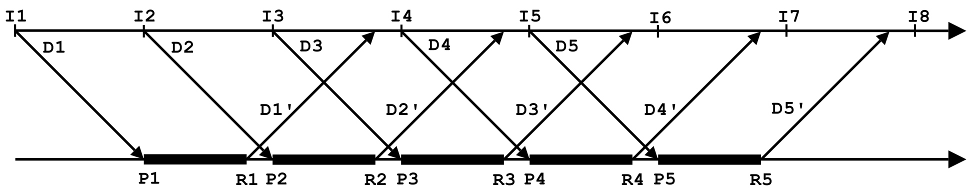
2.5. Summing up

Any system for the distributed processing of real-time applications with low latency could be seen as a distributed firm real-time application capable of operating with callback functions. Thanks to the fact that multimedia applications typically deal with multiple streams of data simultaneously, allocating different data streams to different machines provides

[5]We should note that we cannot extend this idea to window sizes larger than two because each one of the sending, processing, and receiving data phases cannot take longer than one period each. If any of them did, it wouldn't have finished its task when new data to be dealt with was made available on the next period.



(a) window size 1



(b) window size 2

Figure 2: At each interrupt I_n, captured data (D_n) is sent out to the remote machine to be processed (P_n); remotely processed data from a previous iteration (D'_{n-1}or D'_{n-2}) is returned (R_{n-1}or R_{n-2}) and output to the multimedia device before the next interrupt (I_{n+1}).

for a simple and efficient method of load distribution. Given the commodity networking hardware available today, in order to promote better resource usage, such distribution must make use of the sliding windows idea with windows of size one or two, which yield respectively one or two additional network buffers and the corresponding additional latency.

3. A middleware system for distributed real-time, low-latency processing

The data communication between applications on a network may involve many different kinds of data; industry standards such as CORBA (Henning and Vinoski, 2001; Siegel, 2000) promote a high level of abstraction for this kind of communication, allowing for virtually any kind of data to be sent and received transparently over networks of heterogeneous computers. However, mechanisms such as CORBA introduce an overhead that may hinder low latency communication. In order to achieve better real-time and low latency performance, we developed, in C++ under Linux, a simple middleware system (approximately 2000 lines of code) geared towards distributed multimedia processing taking into consideration the aspects discussed in section 2.

This middleware does not intend to compete with systems like CORBA in terms of features, abstraction level, or flexibility; on the contrary, the intent is only to establish a simple and efficient method for the transmission of synchronous data in real-time with low latency in a local network. For this reason, it deals only with preallocated data buffers, not with data organized in an object-oriented way; and, for the same reason, it deals only with buffers of data types native to the C programming language: integers, floats, doubles, and characters.

Coupled with systems like CORBA, however, the present middleware may offer a blend of the most interesting characteristics of such systems, such as flexibility, transparency etc. with good performance on applications with real-time and low latency needs. Our real-time communication mechanism can be used by CORBA objects to achieve low latency communication. The development of distributed applications with real-time and low latency needs can, therefore, use CORBA for its non-real-time aspects and the middleware described here for its real-time aspects.

3.1. Implementation

For each data stream that is to be processed remotely, the central machine creates an instance of the Master class. Objects of this class maintain a pair of UDP "connections" with the remote machine that is responsible for the processing of that stream, one to send and the other to receive the data[6]. Instances of this class have an attribute of type DataBlockSet, which is an (initially empty) collection of instances of the DataBlock class. As can be seen in Figure 3, every time the application needs to register a new buffer of data to be remotely processed, it asks Master for the creation of a new DataBlock; Master delegates this operation to the DataBlockSet. After receiving the reference to the DataBlock, the application can register a pointer to a memory buffer and its size in this DataBlock, as well as define if this buffer must be only read (sent to the remote machine), written to (received from the remote machine) or both. At each iteration, the application then only has to call the `process()` method of the Master object to have the data processed; this method sends and receives the data buffers that are contained in the DataBlocks that are part of the DataBlockSet, taking care not to block when reading from the network, but to use busy-wait instead if needed. Data types sensible to the byte ordering have their bytes rearranged (if necessary) by the DataBlock object that contains them after they are received by either side of the communication link.

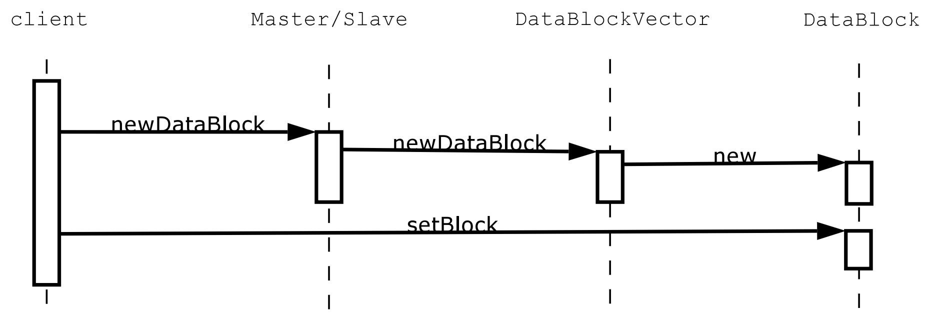


Figure 3: Interaction between the client and the Master/Slave classes for the definition of a new buffer that is to be sent/received through the network.

On the remote machine, an object of the Slave class keeps corresponding UDP "connections" with the central machine and a DataBlockSet with DataBlocks corre-

[6]While the current implementation uses UDP for communication, the connections between the machines are handled by independent classes, which may be substituted in order to support other network protocols. In fact, UDP does not have real connections: in this implementation, the connections exist only on a conceptual level. Also, all data to be sent/received is encapsulated in a single UDP datagram, which limits the size of the data to be transferred at each iteration to around 60KBytes, which is the maximum size of a UDP datagram.

sponding to the DataBlocks created on the central machine. This class also keeps a reference to a callback function, defined by the application, that is called every time a new block of data is received from the network; at the end of the processing, the resulting data is sent back to the central machine (Figure 4). Differently from the Master class, Slave blocks when reading from the network: the availability of network data is the result of the interrupt generated on the central machine, which triggers the operation on the other machines.

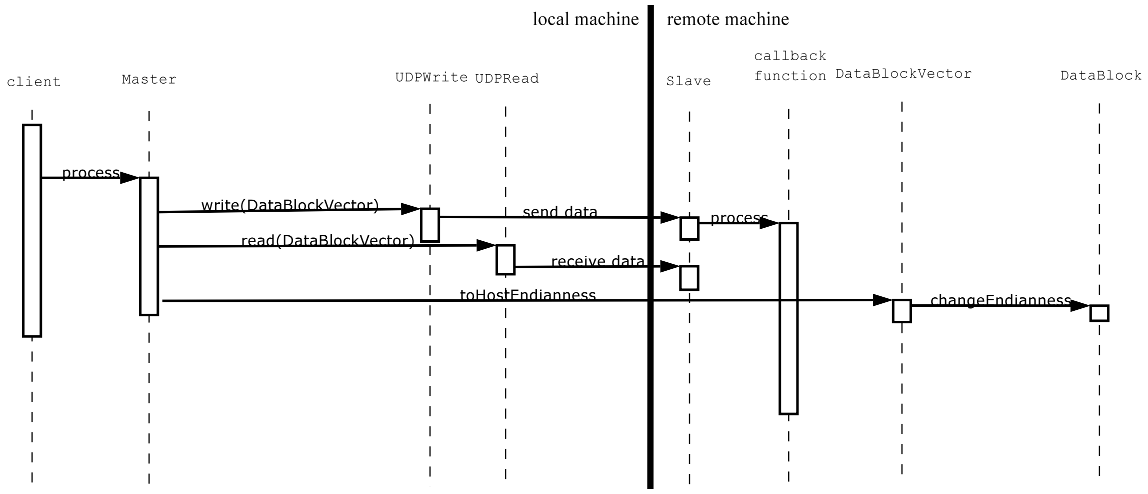


Figure 4: Interaction between the classes for the data exchange between machines.

When created, objects of the Master class define the window size that should be used for the connection. Besides the data buffers allocated by the application, Master also sends and receives a counter, used to detect errors on the ordering of the data received by Master.

4. An application: distributed LADSPA

In order to process the data with the least possible latency, we need to do all the processing inside the callback function. On the other hand, we want to be able to perform several different kinds of processing inside the callback function: in a multimedia editing or processing application, each one of several available processing algorithms should be easily activated, deactivated, combined with others, and applied to different data streams. Also, new algorithms should be incorporated into the system easily.

The most common solution to these requirements is to implement each processing algorithm as an independent software module (usually called a *plugin*) that is loaded by a generic application that deals with the modules without knowing anything about its internals, following the tendency towards component-based development. These modules must be compatible with the operation inside a callback function and must, if possible, avoid the memory allocation needed by pass-by-value function calls; that is, they must process data buffers previously allocated, just as the main callback function does.

In the audio processing field, two specifications for the development of plugins that follow this design have gained widespread use: the VST specification <http://www.steinberg.net/en/ps/support/3rdparty/vst_sdk/index.php?sid=0> under Windows and MacOS and the LADSPA specification <http://www.ladspa.org> under Linux. Any application compatible with either specification can make use of any plugin available under that specification without the need to know anything about the internals of the module and still be able to work with low latency.

In order to make the distributed processing of audio easier for a larger audience of free software users, as well as to stimulate the use of free software in music applications, we decided to implement a mechanism for distributed audio processing compatible with the LADSPA specification, presenting the system to any audio application as an ordinary LADSPA plugin; this way, applications designed to make use of LADSPA plugins are able to use the system unmodified. At the same time, our middleware supports distribution of all the algorithms already available as LADSPA plugins (such as flangers, reverbs, synthesizers, pitch scalers, noise gates etc.), simply by delegating the processing to them. The current version of this implementation is available for download under the LGPL license at <http://gsd.ime.usp.br/software/DistributedAudio>.

The layer responsible for the interaction with the application is simply an application of the Adapter design pattern (Gamma et al., 1994): it presents itself to the application with the interface of a LADSPA plugin but, on the inside, creates a Master object responsible for passing the data to be processed to a remote machine. On the remote machine, the application creates data buffers in which the received data is written; a Slave object receives the data and invokes the callback function that the application registered, which just calls the appropriate function of the real LADSPA plugin. A mechanism for the dynamic creation of "meta-plugins", i.e., LADSPA plugins that are actually compositions of several other LADSPA plugins, is in the works.

Several aspects of our system need to be configured without a need for low latency processing, such as selecting which machine is to perform which processing on which data stream, defining the window size to use etc. For these, another layer, based on CORBA, allows the easy creation and destruction of data processing chains, definition of network connections etc. Machines capable of operating as "slaves" are able to register themselves in a CORBA trader to facilitate the instantiation of the remote plugins.

5. Experimental results

In order to verify the viability of our middleware, we performed experiments with the LADSPA distributed system. We created a LADSPA plugin (the waste_time plugin) that simply copies its input data to its output and then busy-waits for a certain time. When it starts operating, it busy-waits, at each iteration, for as long as necessary to make the whole processing time of that iteration $100\mu s$; after $10s$ of operation, it busy-waits to make the iteration take $200\mu s$; after another $10s$, $300\mu s$; and so on. When the JACK server, jackd, starts issuing timing errors, we know the system cannot cope with the amount of processing time used by the plugin and so we register the largest time an iteration can take before the system stops functioning properly. The plugin also gathers data from the Linux /proc filesystem about the system load in terms of user time and system time and, when the experiment is over, saves the data to a file. With this setup, we were able to determine the maximum percentage of the period that is available for any LADSPA plugin to process and, at the same time, the system load generated during this processing. With this data, we are able to determine the overhead of our middleware system. While $10s$ may seem a small amount of time, since each iteration takes less than $5ms$, this is sufficient to run more than 2000 iterations of each configuration.

5.1. Software and hardware testbed

To run the experiments, we used the JACK daemon, jackd, version 0.72.4; it allowed us to communicate with the sound card with low latency and experiment with several different interrupt frequencies; the sound driver used was ALSA `<http://www.alsa-project.org>` version 0.9.4. On top of jackd, we used the application jack-rack `<http://pkl.net/~node/jack-rack.html>`, version 1.4.1, to load and run our plugins. This application is an effects processor that works as a jackd client loading LADSPA plugins and applying them to the data streams provided by jackd. We first made measurements with the waste_time plugin running on the local machine, just as any other LADSPA plugin. We then performed experiments in which jack-rack would load instances of our proxy plugin, which would then handle the data to be processed to remote instances of the waste_time plugin. At the same time, we measured (using the /proc filesystem) system load on the central machine. The experiments were run partly at 44.1KHz, partly at 96KHz audio sampling rate; LADSPA and JACK treat all samples as 32 bit floats.

The central machine was an Athlon 1.4GHz PC with 256MB RAM, running Debian GNU/Linux with Linux kernel version 2.4.20 with low latency patches applied; the audio card was an M-Audio Delta 44 `<http://www.m-audio.com/products/m-audio/delta44.php>`. The other machines ran a GNU/Linux system with the bare minimum to run the slave application, under the same Linux kernel, version 2.4.20 with low latency patches applied. The remote machines were: another athlon 1.4GHz PC with 256MB RAM, two athlon 1.1GHz PCs with 256MB RAM, one AMD K6-2 400MHz PC with 192 MB RAM, two AMD K6-2 450MHz PCs with 192MB RAM, and one AMD K6-2 350MHz PC with 192MB RAM. The athlons had onboard SiS900 network hardware, while the K6s used low-cost 8139-based PCI network cards. The machines were interconnected by an Encore switching hub model ENH908-NWY+[7].

5.2. Experiment limitations

The waste_time plugin, being very simple, probably allows for the memory cache on the machines where it runs on to be filled with the code needed by the kernel to perform both network communication and task switching; that probably would not be the case with a real application. Therefore, we should expect real applications to have a higher overhead than what was measured. On the other hand, as we will see, the measured overhead was almost nonexistent.

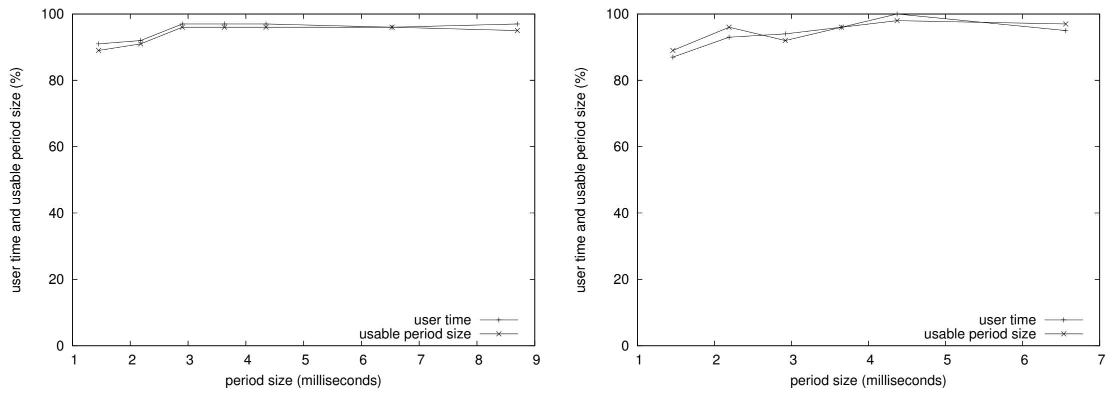
While jackd under Linux runs very well, there were occasional “xruns”, that is, the system was unable to read or write a complete buffer to or from the sound card in time. Xruns of about 30–$60\mu s$ occurred sporadically, always when the load of audio processing was relatively low; they even occurred when jackd was running without any

[7]Many thanks to the folks at *fonte design* for providing us with the test environment.

clients attached and the machine was not performing any audio processing. The probable reason is that, while the typical scheduling latency of the patched kernel is about $500\mu s$, it can sometimes be higher, causing longer delays. When the processing load is higher, the kernel most likely schedules less processes between each interrupt, reducing other I/O activity and, therefore, staying closer to the typical scheduling latency of $500\mu s$. This problem prevented us from experimenting with periods shorter than $1.45ms$, when these random xruns became too common. It is probably possible to reduce or eliminate these xruns by configuring jackd to use three buffers instead of two to communicate with the sound card (at the cost of additional latency) or with a faster machine, but we did not try to do that; instead, we just ignored these sporadic xruns, since they were not related to our system and were easily discernible from the xruns caused by system overload.

5.3. Results

We first tried to determine the maximum time the waste_time plugin could spend at each period when running at the local machine; this experiment serves as a comparison for the distributed processing in which we are interested (Figure 5).



(a) at 44.1KHz audio sampling rate

(b) at 96KHz audio sampling rate

Figure 5: On a single machine, longer periods allow the plugin to use more CPU time, increasing efficiency.

As expected, the smaller periods make the system less efficient, because they imply a higher interrupt rate and, therefore, a higher overhead. Besides that, the scheduling latency is proportionally higher with shorter periods: after the interrupt is issued by the sound card, the kernel takes approximately $500\mu s$ to schedule the application to run and process the data, which is approximately 30% of the time between interruptions when the period size is $1.45ms$. Finally, jitter in the scheduling latency become more troublesome with shorter periods. At 44.1KHz with a period of $1.45ms$, 89% of the period size was usable by the plugin; the load on the system was 91% of user time (the CPU time spent by ordinary processes) and 1% of system time (the CPU time spent by the kernel performing general tasks). Therefore, we could not use more than 92% of the CPU time with this period size. At 96KHz, the maximum usable time in each period and the maximum CPU load obtainable were a little lower. At 44.1KHz with period sizes of $3.0ms$ or more, around 96% of the period size was usable by the plugin; the load of the system was around

97% of user time and 0% of system time. For these period sizes, therefore, we could use nearly all the CPU time to do the processing.

The next experiment was to run the waste_time plugin on remote machines and have them communicate with the central machine with window size zero, i.e., the master machine would busy-wait until the processed data arrived (Figure 6).

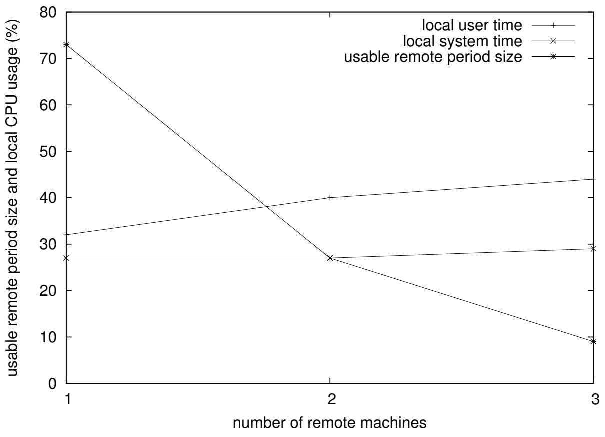


Figure 6: Remote usable CPU time decreases rapidly with more machines when using window size zero.

This proved how impractical that approach would be: there is simply no gain in this setup. The user and system times measured on the central machine can be misleading: they correspond mainly to busy-waiting, which involves a user-space loop that performs a system call (`read()`); much of this time could be used by a local plugin instead, processing other data in parallel with the remote machine. Still, the maximum percentage of the period size usable for processing by a single remote machine is 73%; for two remote machines, this drops to 27% at each machine; for three machines, this drops even more, to 9% at each machine. These gains do not justify distributed processing.

After that, we wanted to measure the local overhead introduced by the system without busy-waiting; in order to do that, we set up the communications layer to use a window of size two. Then we ran the experiment with a period of $2.18ms$ at 44.1KHz with 1, 2, 3, and 4 remote machines and at 96KHz with 4 and 7 machines (Figure 7 on the following page).

The load generated by the system on the central machine, while significant, is totally acceptable; it also grows approximately linearly, which was expected. We also verified that the load on the central machine generated by 4 remote machines with 96KHz sample rate is almost the same as the load generated by 4 remote machines with 44.1KHz sample rate, which actually came as a surprise.

Finally, we wanted to determine the maximum time a plugin could spend at each period when running at several remote machines using a window of size one. We ran the experiment with 4 remote machines at 96KHz audio sample rate using different period sizes and with 7 remote machines using a period of $2.19ms$ (Figure 8 on the next page). This showed that the remote CPU usage is excellent with a relatively low local overhead, even with this many machines: with a period size of $2.19ms$ and a sample rate of 96KHz,

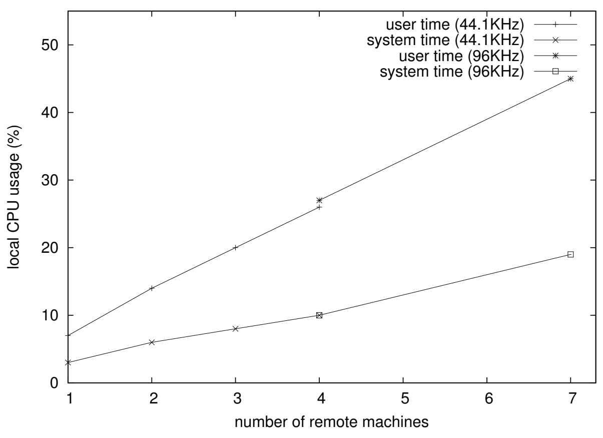


Figure 7: The load on the central CPU increases linearly with the number of remote machines.

the waste_time plugin could run remotely for 96% of the period, yielding 99% of user time CPU usage. The CPU usage on the central machine was about the same as on the previous experiment, 45% user time and 20% system time.

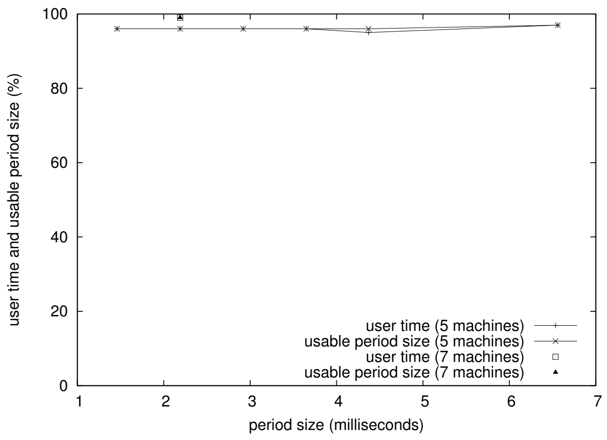


Figure 8: CPU time available for the plugin on the remote machines is similar to that on a single machine.

As we just saw, truly synchronous distributed processing (using window size zero) is not a practical approach to the problem of distributing multimedia processing with low latency. Using the sliding windows idea, though, proved to be an efficient approach: with window size one, it made it possible for us to distribute 96KHz audio to be processed by 7 remote machines with very little overhead on these remote machines, allowing the plugin on them to use almost 100% of the CPU time and with an acceptable load on the central machine. It is reasonable to expect the maximum number of remote machines to be even higher with the hardware used, since the network bandwidth and the central machine have not achieved their maximum usage during the experiments. With a faster processor and a higher-end network interface (which presumably reduces the CPU usage for networking) on the central machine, the limit would probably be imposed by the network physical speed.

6. Conclusions and future work

Distributed multimedia processing with low latency offers some special difficulties; this paper presents a simple approach to overcome these difficulties. The ability to perform multimedia processing in distributed systems opens several possibilities; while this paper addresses only performance aspects, there may be applications that benefit from distributed processing in other ways. For instance, a distributed interactive multimedia application that uses multiple displays in a room and reacts to user input may use a mechanism similar to the one presented here to handle the user input and the corresponding distributed output with low latency.

It would be interesting to be able to process multiple data streams on each of the remote machines; we must perform more experiments to investigate the impact of the resulting additional context switches to the performance of the system. Extending the middleware presented here to operate with pipelined processing might prove useful, since there is an upper limit to the number of parallel machines that the central machine can coordinate.

While the middleware system presented here intends to be generic, the current work conducted with it was heavily based on audio processing. An investigation of the applicability of this system to other forms of multimedia should be conducted in order to investigate limitations, possibilities, and further enhancements. High-resolution video, in particular, involves too much data to be transferred uncompressed in a Fast Ethernet in real-time, which suggests the use of Gigabit Ethernet to perform the communication.

References

ALSA project website. <`http://www.alsa-project.org`>. Accessed on: Jul 24 2003.

ASIO specification website <`http://www.steinberg.net/en/ps/support/3rdparty/asio_sdk/index.php?sid=0`>. Accessed on: Jul 24 2003.

Barabanov, M. and Yodaiken, V. (1996). Real-time linux. *Linux Journal*, (23). Available at: <`http://www.fsmlabs.com/articles/archive/lj.pdf`> Accessed on: Jul 24 2003.

Chen, Z., Tan, S.-M., Campbell, R. H., and Li, Y. (1995). Real Time Video and Audio in the World Wide Web. In *Fourth International World Wide Web Conference*, Boston. Also published in World Wide Web Journal, Volume 1 No 1, January 1996.

M-Audio Delta 44 soundcard information. <`http://www.m-audio.com/products/m-audio/delta44.php`>. Accessed on: Jul 1 2003.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusets.

Henning, M. and Vinoski, S. (2001). *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, Massachusets.

JACK system website. <http://jackit.sourceforge.net>. Accessed on: Jul 24 2003.

Jack-rack website. <http://pkl.net/~node/jack-rack.html>. Accessed on: Jul 1 2003

LADSPA specification website. <http://www.ladspa.org>. Accessed on: Jul 24 2003.

Liu, J. W. S. (2000). *Real-Time Systems*. Addison-Wesley, Reading, Massachusets.

MacMillan, K., Droettboom, M., and Fujinaga, I. (2001). Audio latency measurements of desktop operating systems. In *Proceedings of the International Computer Music Conference*, pages 259–262. Available at: <http://gigue.peabody.jhu.edu/~ich/research/icmc01/latency-icmc2001.pdf> Accessed on: Jul 24 2003.

Morton, A. (2001). Linux scheduling latency. Home of the low latency patch for the Linux kernel. <http://www.zip.com.au/~akpm/linux/schedlat.html> Accessed on: Jul 24 2003.

Shepherd, D., Finney, J., Mathy, L., and Race, N., editors (2001). *International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, volume 2158 of *Lecture Notes in Computer Science*, Berlin. Springer-Verlag.

Siegel, J. (2000). *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2 edition.

Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. (1998). A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium, 3–5 June, 1998*, page 112, Denver, Colorado, USA. IEEE Computer Society. Available at: <http://www.ittc.ku.edu/kurt/papers/conference.ps.gz> Accessed on: Jul 24 2003.

Steinmetz, R. and Nahrstedt, K. (1995). *Multimedia: Computing, Communications & Applications*. Prentice-Hall, Upper Saddle River, NJ.

Stevens, W. R. (1994). *TCP/IP Illustrated, Vol.1: The Protocols*. Addison-Wesley, Reading, Massachusets.

Vieira, J. E. (1999). LINUX-SMART: Melhoria de desempenho para aplicações real-time soft em ambiente LINUX. Master's thesis, IME/USP, São Paulo.

VST specification website. <http://www.steinberg.net/en/ps/support/3rdparty/vst_sdk/index.php?sid=0>. Accessed on: Jul 24 2003.