Distributed Real-Time Audio Processing

Nelson Posse Lago

Abstract

Computer systems for real-time multimedia processing require high processing power. Problems that depend on high processing power are usually solved by using parallel or distributed computing techniques; however, the combination of the difficulties of both real-time and parallel programming has led the development of applications for real-time multimedia processing for general purpose computer systems to be based on centralized and single-processor systems. In several systems for multimedia processing, there is a need for low latency during the interaction with the user, which reinforces the tendency towards single-processor development.

In this work, we implemented a mechanism for synchronous and distributed audio processing with low latency on a local area network which makes the use of a low cost distributed system for this kind of processing possible. The main goal is to allow the use of distributed systems for recording and editing of musical material in home and small studios, bypassing the need for high-cost equipment.

The system we implemented is made of two parts: the first, generic, implemented as a middleware for synchronous and distributed processing of continuous media with low latency; and the second, based on the first, geared towards audio processing and compatible with legacy applications based on the standard LADSPA interface. We expect that future research and applications that share the needs of the system developed here make use of the middleware we developed, both for other kinds of audio processing as well as for the processing of other media forms, such as video.

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 (see Section 2).

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 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.

In this work, we developed a mechanism for distributed audio processing on a LAN with low latency which is compatible with legacy applications in the Linux environment using general purpose, low cost hardware equipment. Linux was chosen for several reasons beyond its (many) technical merits, such as (Kon, 2001; Raymond, 2001; Silveira and Cassino, 2003):

- its low cost;
- the full availability of the source code for both the operating system kernel and the applications that run on top of it;
- the community involvement in the development of the system;
- the social benefit that free software can bring about as part of a strategy for digital inclusion of the poorer population;

The system is based on two layers: the first, generic, is geared towards transparent real-time, low latency data routing between machines on a LAN; the second, specific, deals with the communication with legacy audio applications by means of the LADSPA specification to make use of distributed processing. The generic layer was developed as a reusable middleware (Tanenbaum and Steen, 2002, p. 36–42) for distributed processing of periodic tasks for continuous media in real time with low latency. The experiments showed excellent results with latencies of less than $7ms$, and other hardware configurations should allow latencies much lower still.

2 Low latency

The time a system takes to respond to a stimulus (or to offer some output that is based on some input) is called *latency*; variations in latency during processing is called *jitter*. Several systems, such as those that implement pattern recognition in continuous media (for instance, a system to control the

[1]Prices consulted at <http://www.gateway.com> and <http://www.protools.com> in April, 2004.

movements of a robot by means of computer vision) or such as multimedia interactive systems, must be able to do their processing not only in real-time, but also with low latency and with low jitter.

In non-interactive systems, the desirable characteristics of a system regarding latency and jitter usually can be defined with reasonable precision according to the system domain. For instance, the maximum possible speed of a moving robot in a known environment determines the maximum latency for the system to promote a route change in order to avoid collision. Still, this system might still be flexible: for instance, the robot may slow down if processing exceptionally takes longer than expected[2].

In interactive systems, adequate latency and jitter characteristics are determined by the interaction with the user (Shneiderman, 1984): high latency or jitter may impair the user's performance or, at least, offer a frustrating and tiring experience (Barber and Lucas, Jr., 1983). For instance, the user of a system with a GUI will have difficulties to use it if the on-screen mouse pointer reacts with a long or varying delay to the mouse movements. In a similar way, if the system takes too long to react to mouse clicks, the user may click more than once. So, in order to assess the quality of an interactive system regarding its latency and jitter characteristics, we need to understand their effects on the user's perception so that we can define maximum acceptable values for these parameters on such system.

The acceptable limits for latency and jitter on an interactive system may vary a lot (Miller, 1968). On the above example, the latency of the system with regard to the mouse movements should be very small; the latency for the system to offer visual feedback to the user for a mouse click may be longer; and the latency for the end of the processing started by the mouse click may be even longer. Interactive multimedia applications usually require the lowest latency and jitter values, since they usually involve at least one continuous media that may be modified by the user's interaction. But even in multimedia systems there are differences on the acceptable limits for latency and jitter: human hearing has a higher time precision than vision (Repp, 2003), and the time precision involving different stimuli types (such as visual and auditory or auditory and tactile) is usually lower than temporal precision with stimuli of the same kind (Levitin et al., 1999).

The higher timing precision of hearing and its relevance to music make the control of latency and jitter a very important part of the design of several systems for computer music. In many cases, systems are developed aiming at producing the lowest latency and jitter possible, which current cost-effective technologies put around a few miliseconds. However, many applications, especially those dealing with wide area network delays, cannot typically offer latencies under $10ms$, and may be limited to much higher latencies; still, they are obviously very interesting and are, therefore, developed in spite of the supposedly suboptimal latency and jitter characteristics they are able to offer.

While latency and jitter have been discussed a lot and much is already known about how we perceive them, we still lack experimental research that enables us to understand better the various tradeoffs between latency, jitter, human performance, and perception. For instance, it would be hard to argue that "pop" music requires more strict synchronization between performers than slow-moving textural music; but what are the acceptable limits for latency and jitter in each scenario? And, more importantly, what about other scenarios? When are latency and jitter perceivable? When are they influential on the performance of a musical instrument? When do they degrade the user experience? When do they seriously impair different kinds of human performance?

In this section, we try to show that there are a lot of aspects in human perception regarding latency and jitter that go beyond the usual "less is better" approach; that the naïve "perceptible is bad, not perceptible is good" approach to the problem may be inadequate in some circumstances; and suggest an experiment that could be carried out to help shed some light on the subject.

[2]In fact, that is exactly what a human does when driving a car: if we take a long time to identify the distance and speed of an object ahead of us, we slow down.

2.1 Effects of latency and jitter

Since we are able to use timing deviations as low as $20\mu s$ between ears as cues to determine spatial positioning (Pierce, 1999, p. 102), variations in the typical 44.1KHz sampling frequency may affect our spatial perception. However, since this kind of jitter comes from hardware imprecisions, there is not much that can be done about it but to improve the hardware precision and maybe increase the audio sampling rate. Besides that, this kind of jitter is not directly related in any way to the interactive aspect of a system, and therefore will not be further discussed here.

Timing may also affect the perception of timbre, such as in comb filtering or in tight drumming flams (Wessel and Wright, 2002). Comb filtering effects occur in situations in which an original sound is mixed with a corresponding delayed sound and both sounds are reasonably similar (for instance, the sound of an ordinary acoustical instrument and the same sound processed in order to increase its high frequencies). In most practical situations, however, comb filtering effects do not appear or may be avoided: often, the processed sound is used as a replacement for the original, eliminating the source for the effect. In other cases, it is possible to delay the original sound to eliminate the temporal difference between the original and the processed signals. Probably the only situation where comb filtering cannot be avoided is that in which a real-time acoustical signal is both received directly by the listener (such as what happens with acoustical instruments) and processed and reproduced by electronic means. In this case, however, the effect is similar to what occurs naturally in several acoustic ambients (D'Antonio, s.d.). Thanks to the difference in the spatial positioning of the acoustic sources[3], the the comb filtering effect varies according to the listener's position and is most significant during the sound attack, before the reverberation and other acoustic effects minimize it. This similarity with what occurs in ordinary acoustical ambients suggests that we may ignore the effect in this kind of scenario; in fact, there is not much that can be done about such effect of latency, since almost *any* reasonable latency value, high or low, will result in comb filtering in these situations.

Also, the timbre of flams may be altered by timing differences as low as $1ms$; evidently, comb filtering plays a rôle in this case too, but here it is supposedly under the control of the interpreter. Therefore, a digital system must be able to control event onsets with jitter levels below $1ms$ if such events are to be reproduced faithfully. As we will see later, jitter values close to this are relevant in other scenarios as well, which suggests that trying to achieve low levels of jitter (perhaps by trading it for added latency) is usually a good strategy.

Outside of these extreme examples, the problem with latency and jitter is usually a problem of perceived synchronization: they may prevent us from perceiving events that should appear to be simultaneous as such. This, in turn, may affect our interaction with the system. We may divide pairs of events that may have to be perceived as simultaneous in a musical system in three categories: an external and an internal isochronous beat (that is, the relation of a beat-based musical structure and the corresponding induced beat on the user), pairs of external events (such as pairs of notes or flash lights and note onsets), and actions of the user and their effects (for instance, what happens while playing a musical instrument)[4].

[3] In the case of the effect caused by the ambient acoustical characteristics, each sound reflections act as a secondary audio source.

[4] A special case of synchronization between an internal and an external beat exists when the external beat adjusts to the user's internal beat. A simple example is the rhythm synchronization between music performers, where the internal beat of all performers must be synchonized and, for each performer, the other performer's beat is external. There is already some work on this area (Schuett, 2002), but the results were somewhat inconsistent; we will not address this subject here, but additional work must be done on this topic.

2.1.1 Synchronization in rhythm

One important characteristic of human perception is rhythm, and rhythm is obviously very important in several applications involving music. This is an area where human performance and perception show extremely high precision, although not always in a conscious manner. It was shown that we can tap a steady beat with typical variations in intertap intervals as low as $4ms$ (Rubine and McAvinney, 1990). Similarly, we can also adjust our tapping to compensate for variations of around $4ms$ in interstimuli intervals in an otherwise isochronous pulse sequence (Repp, 2000) and detect consciously timing variations of around $6ms$ (Friberg and Sundberg, 1995). If such variations are cyclic and a little higher, close to $10ms$, we even spontaneously perform together with them (and not only correct our tapping after each variation is detected) (Thaut, Tian and Azimi-Sadjadi, 1998). This kind of adjustment, however, is done subconsciously. Still, it is not unlikely that such variations are perceived not as timing variations, but as some kind of fuzzy musical characteristic like the so-called "feel". In fact, there are strong indications that performers do introduce such variations in performances according to musical context (Bilmes, 1993).

Experimentation suggests that this rhythmic perception is based on the comparison between the expected and actual time for each sound attack (Schulze, 1978); this hypothesis is reinforced by the fact that such precision in tracking rhythmic variations is not significantly affected if we tap out of phase (that is, on the "upbeat") (Repp, 2001). This in turn means that the perception of rhythmic variations of around 10–$20ms$ is not based on auditory cues related to the slight differences in attack moments of close sounds. Instead, such high precision regarding rhythm means we are able to assess time intervals and attack times with around $4ms$ of precision in a subconscious level, and that discrepancies of this magnitude may affect the feel of some kinds of music (those that are based on a very steady isochronous pulse, like many forms of "pop" music). This makes a strong point for the case of trying to minimize jitter as much as possible in a computer music system if such kinds of music are to be supported.

2.1.2 Synchronization in external events

It would be tempting to conclude that such precision in perception means we need to guarantee that events that should be perceived as simultaneous should indeed happen with no more than around $4ms$ of asynchrony between them. However, asynchronies of up to around $50ms$ in supposedly simultaneous notes are not at all uncommon during ordinary music performance. In fact, the percussion and horn sections of an orchestra may be over $10m$ farther from the audience than the violin section, which results in asynchronies around $30ms$ for the public beyond the ordinary asynchronies between instruments. Even in chamber music, asynchronies of up to $50ms$ are common (Rasch, 1979). In a similar way, dynamic differences between voices on pieces for the piano are responsible for what has been called *melody lead*: notes of the melody are tipically played around $30ms$ before other supposedly simultaneous notes[5]. In spite of the percussive characteristic of the piano sound (which results in short attack times and, therefore, very distinguishable attacks), these asynchronies are not perceived as such by performers or the audience. Finally, subjects asked to tap along with a metronomic stimulus virtually always tap about 10–$80ms$ ahead of time (typically $30ms$) without noticing it (Aschersleben, 2002). These facts suggest that latencies responsible for asynchronies in external events of up to at least $30ms$ may be considered normal and acceptable under most circumstances; music performance with traditional

[5]There has been some debate as to wether such effect is only a reflection of the dynamic differences between voices or if it is subconsciously introduced by the performer in order to highlight the melody line. Recent research (Goebl, 2001), however, leaves very little doubt that this effect is indeed a consequence of the dynamics; the perceptual effect of the melody lead effect also appears to be minor (Goebl and Parncutt et al., 2003).

instruments is not impaired by them. In fact, such asynchronies are used by the ear as strong cues for the identification of simultaneous tones (Rasch, 1978).

It may be argued that, even if such asynchronies are not consciously perceptible, they may have a musical role and be partly under the control of the performer; in fact, as just mentioned, they are at least responsible for better tone discrimination. Apparently, though, if such musical role exists, it is minor: not only perceptual experiments showed little impact of variations in artificially-induced asynchronies (Goebl and Parncutt et al., 2003), but also performers apparently do not have such high precision in controlling note asynchronies. This stems from the influence of tactile and kinesthetic (usually called *haptic*) sensations that accompany the action.

2.1.3 Synchronization in haptics

This brings us to the most interesting aspect of latency and jitter for multimedia and music applications: the perception of the latency between an user action and the corresponding reaction. In this respect, our perception once again shows a very high degree of precision: it was shown that variations in feedback delay of $20ms$ are, although not consciously noticed, compensated for in the same manner as we can adjust tapping to a slightly disturbed beat sequence (Wing, 1977). It is reasonable to expect similar mechanisms to be involved in both cases; in fact, it is most likely the same mechanism that is involved: subjects create an expectation for the moment in time for the feedback, detect the feedback disturbance and try to compensate for it.

In spite of the similarity, in such situation there are three elements at stake that make matters more complex: the user's motor commands, the user's corresponding haptic sensations, and their relation to the external feedback. These elements are important because there is very strong evidence suggesting that the moment we recognize as the moment of start of external feedback can be widely influenced by several factors, including the haptic sensations (which are themselves a form of feedback) (Aschersleben, 2002). This means that events that actually happen simultaneously may be perceived as asynchronous, even if only at a subconscious level.

As mentioned before, subjects typically tap together with a metronomic stimulus ahead of time. The amount of anticipation, however, is dependent on the characteristics of both auditory and haptic feedback. Auditory-only feedback produces perfect synchronization; haptic-only feedback produces reasonably large anticipations; both forms of feedback together produce relatively small anticipations; and finally, normal haptic feedback combined with delayed auditory feedback produce anticipations that grow in accordance with the amount of delay (Stenneken et al., 2003; Aschersleben and Prinz, 1997; Mates and Aschersleben, 2000). Excluding the very special cases of auditory-only feedback, such measured variations were of about $15ms$ for auditory feedback delays between zero and a little less than $30ms$ for subjects that proved to show very little variability due to previous training. Anticipations also tend to decrease in contexts where there is sound data in between beats. Such variations give further indication that, while asynchronies in note onsets are used as cues to tone discrimination, their role in musical expression is probably very limited.

The most important aspect of this is the fact that we can subconsciously adjust our performance to compensate for such different feedback conditions. During experiments with delayed feedback, subjects clearly altered their behavior according to the characteristics of each trial, forcing the researchers to introduce control trials between each pair of trials (Aschersleben and Prinz, 1997; Mates and Aschersleben, 2000). In piano performance, the time elapsed between pressing a key and the corresponding note onset is around $100ms$ for *piano* notes and around $30ms$ for *staccato*, *forte* notes (Askenfelt and Jansson, 1990). Even if we assume that the pianist expects the note onset to happen somewhere in the middle of the course of the key, it is very likely that latencies will be different for different dynamic

levels. Still, pianists have no problem dealing with such different latencies; since voices in pieces for the piano usually have dynamics that change continuously, the performer has the opportunity to adjust himself to the corresponding changes in latency. When there are abrupt changes in dynamics, they usually are related to some structural aspect of the music, which brings with it large interpretative timing variations. Finally, modern music may make use of dynamic changes that do not fit well with interpretative timing variations; however, such music is usually not based on a clear and steady beat, making the effects of sudden variations in latency much less perceptible.

In fact, since our motor system cannot react instantaneously, we must issue motor commands ahead of time in order to perform "on time"; it is not hard to believe that the various feedbacks for our actions are used to calibrate how much ahead of time commands are issued. In tapping experiments, latencies of up to around $30ms$ were adjusted for, resulting in final asynchronies between stimulus and response variations of about $10ms$ (Mates and Aschersleben, 2000), which, as previously stated, we believe are mostly irrelevant.

2.2 The future

We hope we have been able to argue convincingly that somewhat large latencies, maybe up to 20–$30ms$, are pretty much acceptable for most multimedia and music applications. Jitter, on the other hand, can be a bigger problem; but it is generally possible to trade jitter for added latency. This does not mean lower latencies are not of interest; quite on the contrary, since latency in different parts of a system accumulates. For instance, the mere positioning of loudspeakers at around 3–$4m$ of distance from a user adds $10ms$ to the total perceived latency of the system; many DSP algorithms add significant latency; etc. Therefore, aiming at the lowest possible latency in each part of a system helps keep the overall latency under control. Still, tradeoffs are acceptable.

Currently available data is still insufficient to determine clearer limits for latency and jitter as well as to confirm much of what was said here in a musical context, making it difficult to assess the quality of musical and multimedia applications regarding temporal precision. This is so because much of the current research on music and timing perception makes use of non-musical stimuli. Since timing is so tightly tied to still unmeasurable aspects of music such as "feel" and since at least part of our timing perception occurs outside of consciousness, we need more experiments performed on actual music. Such experiments would face many technical challenges, some new, some of which have already been dealt with before (Bilmes, 1993; Levitin et al., 1999).

As an example, one such experiment would be, on a small ensemble, to subject one of the instrumentists to different feedback latencies to assess their effect over the perfomer. This should be repeated with feedbacks provided by earphones, loudspeakers, with and without artificial reverberation. Also, different kinds of music (such as tonal classical music and percussion-rich "pop" music) might have different impacts. Finally, running such tests in different rooms would be useful, since the acoustical ambience may affect the performer.

3 Callback functions

The usual read/write mechanism offered by general purpose computers and operating systems 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 [ASIO] and, under Linux, by the JACK system [JACK]. 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 and the application would hardly be re-scheduled in time to complete the processing before the next interrupt.

3.1 Callback-based plugin systems

Many algorithms for multimedia processing are commonly used; it's the way by which they are combined with captured data and with each other that creates new material. Therefore, in a multimedia editing or processing application, each one of several available algorithms should be easily activated, deactivated, combined with others and applied to different data streams; also, new algorithms should be easily incorporated into the system. The most common solution to these needs is to implement each 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 the internals of the module, following the more general tendency towards component-based software development.

As we just saw, in order to perform the I/O data processing with low latency, the application must register a callback function with the system that performs all the necessary computations. If, however, most of the processing is to be done by independent software modules managed by the application, 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 accept pointers to data buffers that need to be processed, 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 [VST] under Windows and MacOS and the LADSPA specification [LADSPA] 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.

4 Distributed real-time systems

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 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.

4.1 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.

4.2 Real-time systems

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 in such a way as to make it clear that a timing failure is absolutely unacceptable in such systems (it might threaten human life, for instance). On the other hand, the definitions for soft real-time systems have very different meanings: they may be characterized as systems where timing failures are undesirable but tolerable, or by the ability to adapt to timing failures (for instance, a video playback application may skip or duplicate frames during execution if deadlines are missed[6]), etc.

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

Besides that, there are lots of aspects that define a real-time system; the terms "soft real-time" and "hard real-time", therefore, are not enough to characterize a real-time system. Nonetheless, hard and soft real-time systems have historically been associated to distinct fields of applications.

However, 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[7].

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.

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, Droettboom and Fujinaga, 2001).

4.3 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[8]. 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,

[7] 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.

[8] 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.

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).

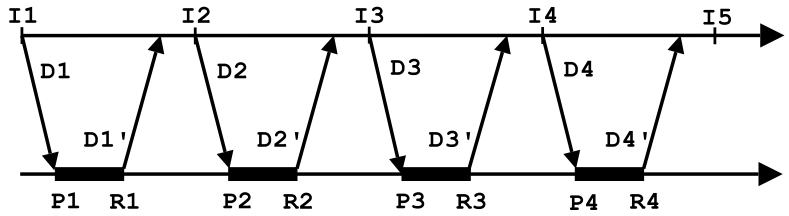


Figure 1: At each interrupt I_n, captured data (D_n) is sent out to the remote machine, processed (P_n), returned (R_n) to the originating machine already processed (D'_n), and output to the multimedia device before the next interrupt (I_{n+1}).

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):

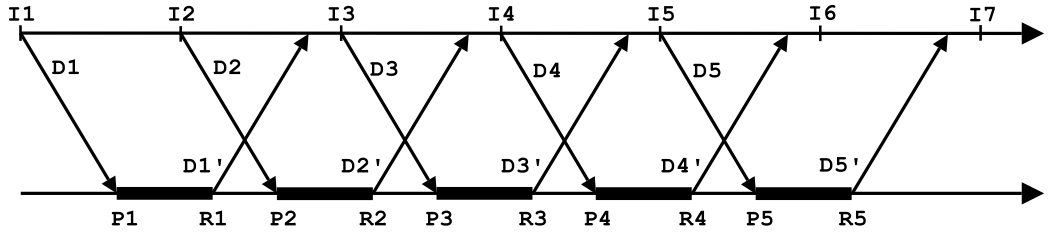
1. the data transfer from the originating machine to the remote machine;

2. the remote data processing;

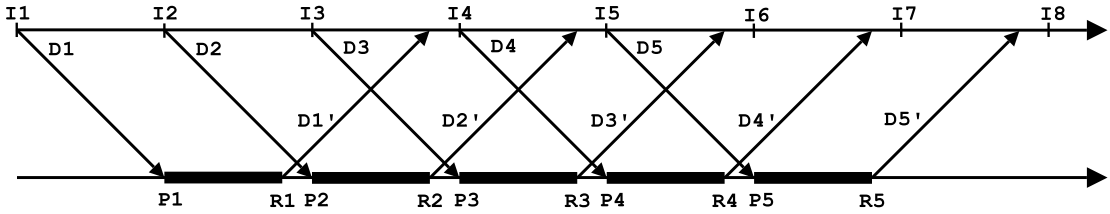
3. the data transfer from the remote machine back to the initial machine.

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 – 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 – 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[9].

[9]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}).

5 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 above.

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.

5.1 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[10]. 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.

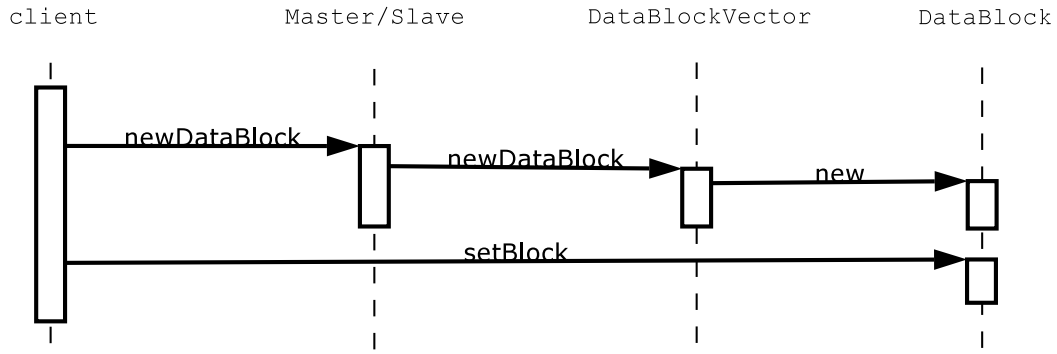


Figure 3: Interaction between the client and the Master/Slave classes for the definition of a new buffer that is to be sent/received through the network.

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

[10]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.

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.

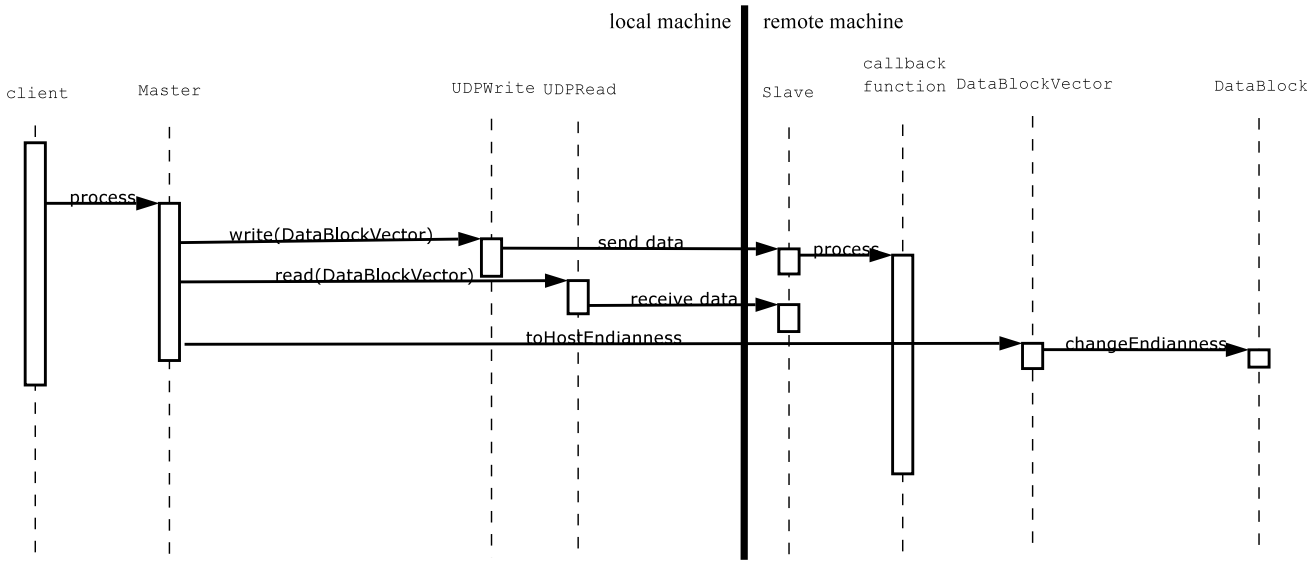


Figure 4: Interaction between the classes for the data exchange between machines.

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.

6 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 [VST] under Windows and MacOS and the LADSPA specification [LADSPA] 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.

7 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.

7.1 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 version 0.9.4. On top of jackd, we used the application jack-rack [JACKRACK], 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 Delta44. 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+[11].

7.2 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.

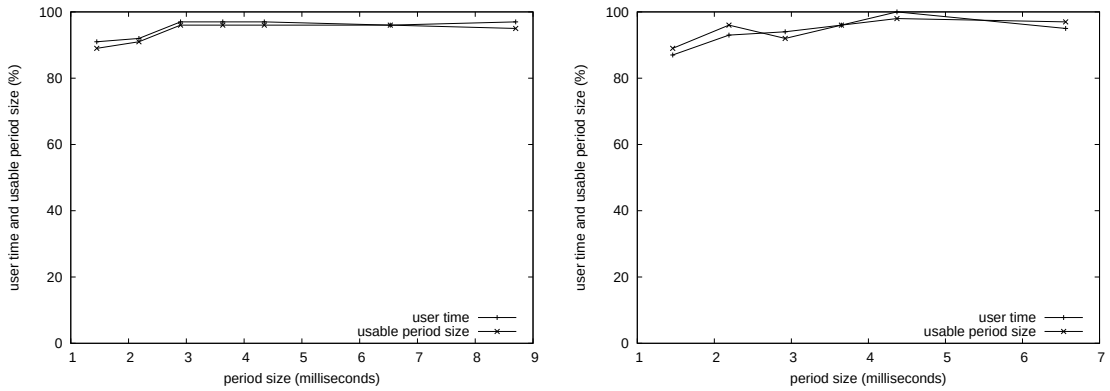
While jackd under Linux runs very well, there were occasional "xruns", that is, the system was unable to read or write a complete buffer to or from the sound card in time. Xruns of about 30–60μs occurred sporadically, always when the load of audio processing was relatively low; they even occurred when jackd was running without any 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 500μs, 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 500μs. 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.

7.3 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).

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 500μs to schedule the application to run and process the data, which is approximately 30% of the

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



(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.

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).

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).

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.

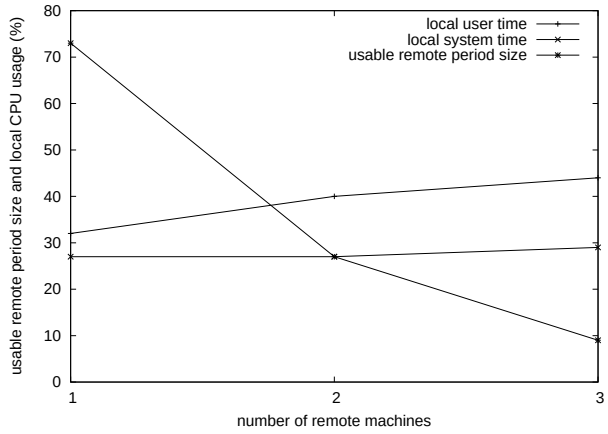


Figure 6: Remote usable CPU time decreases rapidly with more machines when using window size zero.

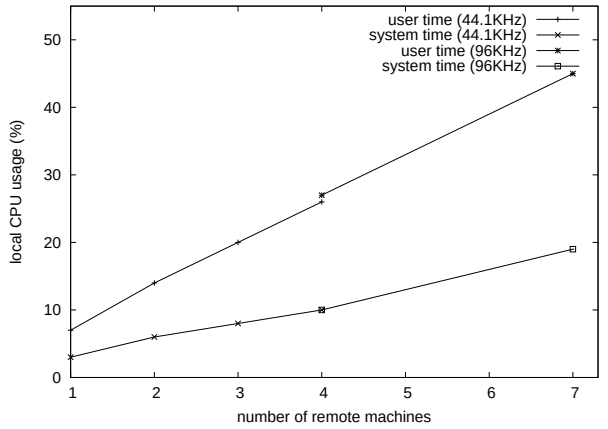


Figure 7: The load on the central CPU increases linearly with the number of remote machines.

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). 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, 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.

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

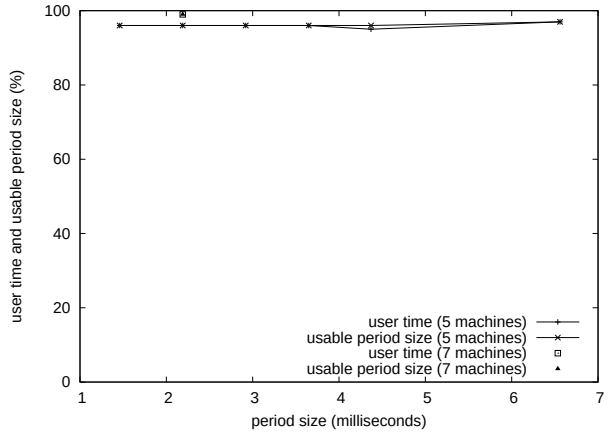


Figure 8: CPU time available for the plugin on the remote machines is similar to that on a single machine.

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.

8 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.

Bibliography

ADELSTEIN, Bernard D. et al. Sensitivity to haptic-audio asynchrony. In: *Proceedings of the 5th International Conference on Multimodal Interfaces*. New York, NY, USA: ACM Press, 2003. p. 73–76. Available at: <http://human-factors.arc.nasa.gov/ihh/spatial/papers/pdfs_bda/Adelstein_2003_Haptic_Audio_Asynchrony.pdf> Accessed at: 10 may 2004.

[ALSA] <http://www.alsa-project.org>. *Website of the ALSA system*. Accessed at: 8 feb 2004.

[ARDOUR] <http://www.ardour.org>. *Website of the Ardour program*. Accessed at: 8 feb 2004.

ASCHERSLEBEN, Gisa. Temporal control of movements in sensorimotor synchronization. *Brain and Cognition*, v. 48, p. 66–79, 2002.

ASCHERSLEBEN, Gisa; PRINZ, Wolfgang. Delayed auditory feedback in synchronization. *Journal of Motor Behavior*, v. 29, n. 1, p. 35–46, 1997.

[ASIO] <http://www.steinberg.net/en/ps/support/3rdparty/asio_sdk/index.php?sid=0>. *Website of the ASIO specification*. Accessed at: 24 jul 2003.

ASKENFELT, Anders; JANSSON, Erik V. From touch to string vibrations. I: Timing in the grand piano action. *Journal of the Acoustical Society of America*, v. 88, n. 1, p. 52–63, 1990.

BARABANOV, Michael; YODAIKEN, Victor. Real-time linux. *Linux Journal*, n. 23, março 1996. Available at: <http://www.fsmlabs.com/articles/archive/lj.pdf> Accessed at: 5 apr 2004.

BARBER, Raymond E.; LUCAS, JR., Henry C. System response time operator productivity, and job satisfaction. *Communications of the ACM*, ACM Press, New York, NY, USA, v. 26, n. 11, p. 972–986, 1983.

BERNAT, Guillem; BURNS, Alan. Combiming (n m) hard deadlines and dual priority scheduling. In: *Proceedings of the IEEE Symposium on Real-Time Systems*. [S.l.: s.n.], 1997. p. 46–57.

BERNAT, Guillem; BURNS, Alan; LLAMOSI, Albert. Weakly hard real-time systems. *IEEE Transactions on Computers*, v. 50, n. 4, p. 308–321, 2001. Available at: <http://citeseer.nj.nec.com/bernat99weakly.html> Accessed at: 2 feb 2004.

BILMES, Jeffrey Adam. *Timing is of the Essence: Perceptual and Computational Techniques for Representing, Learning, and Reproducing Expressive Timing in Percussive Rhythm*. Dissertation (Masters) — MIT, Cambridge, Massachusetts, 1993. Available at: <http://www.icsi.berkeley.edu/~bilmes/mitthesis/mit-thesis.pdf> Accessed at: 10 may 2004.

BURNETT, Theresa A. et al. Voice f0 responses to manipulations in pitch feedback. *Journal of the Acoustical Society of America*, v. 103, n. 3, p. 3153–3161, 1998.

BUTENHOF, David R. *Programming with POSIX Threads*. Reading, Massachusets: Addison-Wesley, 1997.

CARLSON, Ludvig; NORDMARK, Anders; WIKLANDER, Roger. *Nuendo 2.1 Operation Manual*. [S.l.]: Steinberg Media Technologies GmbH, 2003. Available at: <ftp://ftp.pinnaclesys.com/Steinberg/download/Documents/Nuendo_2/english/Nuendo_21_Operation_Manual_en_11082K.pdf> Accessed at: 5 apr 2004.

CARRIERO, Nicholas; GELERNTER, David. *How to write parallel programs: a first course*. Cambridge, Massachusetts: The MIT Press, 1990.

[CERES] <http://www.notam02.no/arkiv/src>. *Website of the Ceres program*. Accessed at: 5 apr 2004.

CHAUDHURI, Pranay. *Parallel algorithms: design and analysis*. Brunswick, Victoria: Prentice Hall of Australia, 1992.

CHEN, Zhigang et al. Real Time Video and Audio in the World Wide Web. In: *Fourth International World Wide Web Conference*. Boston: [s.n.], 1995. Also published in World Wide Web Journal, Volume 1 No 1, January 1996.

CHOUDHARY, Alok N. et al. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems*, v. 5, n. 4, p. 439–445, 1994. Available at: <http://citeseer.nj.nec.com/choudhary94optimal.html> Accessed at: 15 feb 2004.

CLARK, David D.; SHENKER, Scott; ZHANG, Lixia. supporting real-time applications in an integrated services packet network: architecture and mechanism. *Computer Communication Review*, ACM Press, New York, NY, USA, v. 22, n. 4, p. 14–26, October 1992.

CLYNES, Manfred. Entities and brain organization: Logogenesis of meaningful time-forms. In: PRIBRAM, K. H. (Ed.). *Proceedings of the Second Appalachian Conference on Behavioral Neurodynamics*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1994. Available at: <http://www.superconductor.com/clynes/entitie.htm> Accessed at: 10 may 2004.

CLYNES, Manfred. Microstructural musical linguistics: Composer's pulses are liked best by the best musicians. *COGNITION, International Journal of Cognitive Science*, v. 55, p. 269–310, 1995. Available at: <http://www.superconductor.com/clynes/cognit.htm> Accessed at: 10 may 2004.

[COMMONMUSIC] <http://www-ccrma.stanford.edu/software/cm/doc/cm.html>. *Website of the Common Music program*. Accessed at: 8 feb 2004.

COOK, Perry (Ed.). *Music, Cognition, and Computerized sound: an Introduction to Psychoacoustics*. Cambridge, Massachusetts: MIT Press, 1999.

[COREAUDIO] <http://developer.apple.com/audio/macosxaudio.html>. *Website of the CoreAudio specification*. Accessed at: 5 apr 2004.

D'ANTONIO, P. Minimizing acoustic distortion in project studios. Available at: <http://www.rpginc.com/cgi-bin/byteserver.pl/news/library/PS_AcD.pdf> Accessed at: 5 apr 2004. [s.d.].

[DELTA] <http://www.m-audio.com/products/m-audio/delta44.php>. *Informações sobre a placa de som Delta-44 da M-Audio*. Accessed at: 1 jul 2003.

FARINES, Jean-Marie; FRAGA, Joni da S.; OLIVEIRA, Romulo S. de. *Sistemas de Tempo Real*. São Paulo: IME/USP, 2000. (Escola de Computação).

FOSTER, Ian. *Designing and Building Parallel Programs (Online)*. Reading, Massachusets: Addison-Wesley, 1995. Available at: <http://www-unix.mcs.anl.gov/dbpp> Accessed at: 15 feb 2004.

FRIBERG, Anders; SUNDBERG, Johan. Time discrimination in a monotonic, isochronous sequence. *Journal of the Acoustical Society of America*, v. 98, n. 5, p. 2524–2531, 1995.

[FSF] <http://www.fsf.org>. *Website of the Free Software Foundation*. Accessed at: 8 feb 2004.

GALLMEISTER, Bill O. *Posix.4: Programming for the Real World*. Sebastopol: O'Reilly & Associates, Inc., 1995.

GAMMA, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusets: Addison-Wesley, 1994.

GIBBONS, Alan; RYTTER, Wojciech. *Efficient Parallel Algorithms*. Cambridge: Cambridge University Press, 1988.

GILLESPIE, Brent. Haptics. In: COOK, Perry (Ed.). *Music, Cognition, and Computerized sound: an Introduction to Psychoacoustics*. Cambridge, Massachusetts: MIT Press, 1999a. p. 229–245.

GILLESPIE, Brent. Haptics in manipulation. In: COOK, Perry (Ed.). *Music, Cognition, and Computerized sound: an Introduction to Psychoacoustics*. Cambridge, Massachusetts: MIT Press, 1999b. p. 247–260.

GOEBL, Werner. Melody lead in piano peformance: Expressive device or artifact? *Journal of the Acoustical Society of America*, v. 110, n. 1, p. 563–572, 2001.

GOEBL, Werner; PARNCUTT, Richard. Asynchrony versus intensity as cues for melody perception in chords and real music. In: KOPIEZ, R. et al. (Ed.). *Proceedings of the 5th Triennial ESCOM Conference, September 8–13*. Hanover, Germany: [s.n.], 2003. p. 376–380. Available at: <http://www.oefai.at/cgi-bin/get-tr?paper=oefai-tr-2003-11.pdf> Accessed at: 10 may 2004.

[GSTREAMER] <http://www.gstreamer.net>. *Website of the gstreamer system*. Accessed at: 10 feb 2004.

HAMDAOUI, Moncef; RAMANATHAN, Parameswaran. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE transactions on computers*, v. 44, n. 12, p. 1443–1451, December 1995.

HENNING, Michi; VINOSKI, Steve. *Advanced CORBA Programming with C++*. Reading, Massachusets: Addison-Wesley, 2001.

HU, X. Sharon et al. *Firm real-time system scheduling based on a novel QoS constraint*. [s.d.]. Available at: <http://www.nd.edu/~lemmon/rtss03_submitted.pdf> Accessed at: 2 feb 2004.

IYER, Vijay S. *Microstructures of Feel, Macrostructures of Sound: Embodied Cognition in West African and African-American Musics*. Thesis (Doctorate) — University of California, Berkeley, Berkeley, California, 1998. Available at: <http://cnmat.cnmat.berkeley.edu/People/Vijay/%20THESIS.html> Accessed at: 10 may 2004.

[JACK] <http://jackit.sourceforge.net>. *Website of the JACK specification*. Accessed at: 5 apr 2004.

[JACKRACK] <http://arb.bash.sh/~rah/software/jack-rack/>. *Website of the Jack-Rack program*. Accessed at: 5 apr 2004.

[JMAX] <http://freesoftware.ircam.fr/rubrique.php3?id_rubrique=14>. *Website of the jMAX program*. Accessed at: 8 feb 2004.

JOSUTTIS, Nicolai M. *The C++ Standard Library: a Tutorial and Reference*. Reading, Massachusets: Addison-Wesley, 1999.

KIENTZLE, Tim. *A Programmer's Guide to Sound*. Reading, Massachusets: Addison-Wesley, 1998.

KING, Chung-Ta; CHOU, Wen-Hwa; NI, Lionel M. Pipelined data-parallel algorithms – concept and modelling. In: *Proceedings of the 2nd international conference on Supercomputing – St. Malo, France*. New York, NY: ACM Press, 1988. p. 385–395.

KON, Fabio. *O Software Aberto and a Questão Social*. São Paulo, may 2001. Available at: <http://www.ime.usp.br/~kon/papers/RT-SoftwareAberto.pdf> Accessed at: 5 apr 2004.

KOPETZ, Hermann; VERÍSSIMO, Paulo. Real time and dependability concepts. In: MULLENDER, Sape (Ed.). *Distributed Systems*. New York, NY: ACM Press, 1993. p. 411–446.

KUMAR, Vipin et al. *Introduction to Parallel Computing: design and analysis of algorithms*. Redwood City, California: The Benjamin/Cummings Publising Company, Inc., 1994.

KUNG, H. T. et al. Network-based multicomputers: an emerging parallel architecture. In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing — Albuquerque, New Mexico, United States*. New York, NY, USA: ACM Press, 1991. p. 664–673.

[LAD] <http://www.linuxdj.com/audio/lad/>. *Website of the Linux audio developers mailing list*. Accessed at: 5 apr 2004.

[LADSPA] <http://www.ladspa.org>. *Website of the LADSPA specification*. Accessed at: 5 apr 2004.

LARGE, Edward W.; FINK, Philip; KELSO, J. A. Scott. Tracking simple and complex sequences. *Psychological Research*, v. 66, p. 3–17, 2002.

[LATENCYTEST] <http://www.gardena.net/benno/linux/audio>. *Website of the latencytest program*. Accessed at: 8 feb 2004.

LEVITIN, Daniel J. et al. The perception of cross-modal simultaneity. Available at: <http://ccrma-www.stanford.edu/~lonny/papers/casys1999.pdf> Accessed at: 10 may 2004. 1999.

LEYDON, Ciara; BAUER, Jay J.; LARSON, Charles R. The role of auditory feedback in sustaining vocal vibrato. *Journal of the Acoustical Society of America*, v. 114, n. 3, p. 1575–1581, 2003.

LIU, Jane W. S. *Real-Time Systems*. Reading, Massachusets: Addison-Wesley, 2000.

LIU, Jane W. S. et al. Imprecise computations. *Proceedings of the IEEE*, v. 82, n. 1, p. 83–94, January 1994.

[LLPATCH2.2] <http://people.redhat.com/mingo/lowlatency-patches>. *Website of the low latency patch for Linux 2.2*. Accessed at: 8 feb 2004.

[LLPATCH2.4] <http://www.zip.com.au/~akpm/linux/schedlat.html>. *Scheduling latency in Linux*. Website of the low latency patch for Linux. Accessed at: 8 feb 2004.

[LLPATCH2.6] <http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/v2.4>. *Website of the Linux kernel preemption patch*. Incorporated in Linux 2.6. Accessed at: 8 feb 2004.

MacMillan, K.; Droettboom, M.; Fujinaga, I. Audio latency measurements of desktop operating systems. In: *Proceedings of the International Computer Music Conference*. [S.l.: s.n.], 2001. p. 259–262. Available at: <http://gigue.peabody.jhu.edu/~ich/research/icmc01/latency-icmc2001.pdf> Accessed at: 5 apr 2004.

Mates, Jirí; Aschersleben, Gisa. Sensorimotor synchronization: the impact of temporally displaced auditory feedback. *Acta Psychologica*, v. 104, p. 29–44, 2000.

[MAX] <http://www.cycling74.com/products/maxmsp.html>. *Website of the MAX/MSP distributor*. Accessed at: 8 feb 2004.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. 1995. Available at: <http://www.mpi-forum.org/docs/mpi-11.ps> Accessed at: 5 apr 2004.

Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. 1997. Available at: <http://www.mpi-forum.org/docs/mpi-20.ps> Accessed at: 5 apr 2004.

Metts, Allan. Sounds from another planet. *Electronic Musician*, January 2004. Available at: <http://emusician.com/ar/emusic_sounds_planet/index.htm> Accessed at: 08 feb 2004.

[MIDI] <http://www.midi.org>. *Website of the industry consortium responsible for the MIDI specification*. Accessed at: 10 feb 2004.

Miller, Robert B. Response time in man-computer conversational transactions. In: *AFIPS Conference Proceedings — fall joint computer conference, San Francisco, CA*. Washington: Thompson Book Company, 1968. v. 33, part 1.

Mills, David L. *RFC1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis*. 1992. Available at: <http://www.faqs.org/rfcs/rfc1305.html> Accessed at: 5 apr 2004.

[MIXVIEWS] <http://www.ccmrc.ucsb.edu/~doug/htmls/MiXViews.html>. *Website of the MiXViews program*. Accessed at: 5 apr 2004.

[MUSE] <http://muse.seh.de>. *Website of the Muse program*. Accessed at: 5 apr 2004.

Ng, Jim M.; Yu, Norris T. C. Transport protocol for real-time multimedia communication. In: Halang, W. A. (Ed.). *IFAC/IFIP Workshop on Real Time Programming*. Oxford, UK: Elsevier Science Ltd., 1994. (Annual Review in Automatic Programming, v. 18), p. 15–20.

Norman, Michael G.; Thanisch, Peter. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, ACM Press, New York, NY, v. 25, n. 3, p. 263–302, September 1993.

OMG. *CORBA services: Common Object Services Specification*. Framingham, MA, 1998. OMG Document 98-12-09.

[OPENSOURCE] <http://www.opensource.org>. *Website of the Open Source Initiative*. Accessed at: 8 feb 2004.

Oppenheimer, Steve et al. The complete desktop studio. *Electronic Musician*, v. 15, n. 6, p. 48–104, junho 1999.

[PATCHWORK] <http://www.ircam.fr/produits/logiciels/log-forum/patchwork.html>. *Website of the Patchwork program*. Accessed at: 5 apr 2004.

Phillips, Dave. *Linux music & sound*. San Francisco, CA: No Starch Press, 2000.

Pierce, John. Hearing in time and space. In: Cook, Perry (Ed.). *Music, Cognition, and Computerized sound: an Introduction to Psychoacoustics*. Cambridge, Massachusetts: MIT Press, 1999. p. 89–103.

[PLUGINS] <http://plugin.org.uk>. *Website of the Steve Harris LADSPA plugin collection*. Accessed at: 5 apr 2004.

[POSIX] <http://www.pasc.org>. *Website of the Portable Application Standards Comitee*. Responsable for the IEEE POSIX standards. Accessed at: 8 feb 2004.

[PROTOOLS] <http://www.digidesign.com>. *Website of the Pro Tools system manufacturer*. Accessed at: 10 feb 2004.

Rasch, R. A. The perception of simultaneous notes such as in polyphonic music. *Acustica*, v. 40, p. 21–33, 1978.

Rasch, R. A. Synchronization in performed ensemble music. *Acustica*, v. 43, p. 121–131, 1979.

Raymond, Eric S. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol: O'Reilly & Associates, Inc., 2001. there is a version online at: <http://www.catb.org/~esr/writings/cathedral-bazaar> Accessed at: 5 apr 2004.

Repp, Bruno H. Compensation for subliminal timing perturbations in perceptual-motor synchronization. *Psychological Research*, v. 63, p. 106–128, 2000.

Repp, Bruno H. Phase correction, phase resetting, and phase shifts after subliminal timing perturbations in sensorimotor synchronization. *Journal of Experimental Psychology: Human Perception and Performance*, v. 27, n. 3, p. 600–621, 2001.

Repp, Bruno H. Rate limits in sensorimotor synchronization with auditory and visual sequences: The synchronization threshold and the benefits and costs of interval subdivision. *Journal of Motor Behavior*, v. 35, n. 4, p. 355–370, 2003.

Rhyde, Randall. The art of assembly language programming. Available at: <http://webster.cs.ucr.edu/AoA/DOS/> Accessed at: 5 apr 2004. 1996.

Roads, Curtis. *The Computer Music Tutorial*. Cambridge, Massachusetts: The MIT press, 1996.

Rubine, Dean; McAvinney, Paul. Programmable finger-tracking instrument controllers. *Computer Music Journal*, v. 14, n. 1, p. 26–40, 1990.

Schuett, Nathan. The effects of latency on ensemble performance. Available at: <http://www-ccrma.stanford.edu/groups/soundwire/performdelay.pdf> Accessed at: 10 may 2004. 2002.

Schulze, Hans-Henning. The detectability of local and global displacements in regular rhythmic patterns. *Psychological Research*, v. 40, p. 173–181, 1978.

Schulzrinne, H.; Casner, S. *RFC3551: RTP Profile for Audio and Video Conferences with Minimal Control*. 2003. Available at: <http://www.faqs.org/rfcs/rfc3551.html> Accessed at: 5 apr 2004.

SCHULZRINNE, H. et al. *RFC3550: RTP: A Transport Protocol for Real-Time Applications*. 2003. Available at: <http://www.faqs.org/rfcs/rfc3550.html> Accessed at: 5 apr 2004.

SEVCIK, Kenneth C. Characterizations of parallelism in applications and their use in scheduling. In: *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems – Oakland, California, United States*. New York, NY: ACM Press, 1989. p. 171–180.

SHNEIDERMAN, Ben. Response time and display rate in human performance with computers. *ACM Computing Surveys*, ACM Press, New York, NY, USA, v. 16, n. 3, p. 265–285, 1984.

SIEGEL, J. *CORBA 3 Fundamentals and Programming*. 2 ed. New York: John Wiley & Sons, 2000.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer. *Sistemas Operacionais: conceitos*. São Paulo: Prentice Hall, 2000.

SILVEIRA, Sérgio Amadeu da; CASSINO, João (Ed.). *Software livre and inclusão digital*. São Paulo: Conrad Editora do Brasil, 2003.

[SOUNDAPPS] <http://sound.condorow.net>. *Website with a very comprehensive and up-to-date list of music and audio programs for Linux*. Accessed at: 8 feb 2004.

SRINIVASAN, B. et al. 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*. Denver, Colorado, USA: IEEE Computer Society, 1998. p. 112. Available at: <http://www.ittc.ku.edu/kurt/papers/conference.ps.gz> Accessed at: 5 apr 2004.

STANKOVIC, John A. Misconceptions about real-time systems. *IEEE Computer*, v. 21, n. 10, p. 10–19, October 1988.

STANKOVIC, John A. Real-time computing. *BYTE*, p. 155–160, August 1992. Available at: <http://www.idt.mdh.se/kurser/ct3200/regular/ht03/download/articles/rt-intro.pdf> Accessed at: 25 jan 2004.

STEINMETZ, Ralf; NAHRSTEDT, Klara. *Multimedia: Computing, Communications & Applications*. Upper Saddle River, NJ: Prentice-Hall, 1995.

STENNEKEN, Prisca et al. Anticipatory timing of movements and the role of sensory feedback: Evidence from deafferented patients. Available at: <http://jacquespaillard.apinc.org/deafferented/pdf/stenneken-et-al-ms-03.pdf> Accessed at: 10 may 2004. 2003.

STEVENS, W. Richard. *TCP/IP Illustrated, Vol.1: The Protocols*. Reading, Massachusets: Addison-Wesley, 1994.

STROUSTRUP, Bjarne. *The C++ Programming Language*. Reading, Massachusets: Addison-Wesley, 1997.

SUBHLOK, Jaspal; VONDRAN, Gary. Optimal latency–throughput tradeoffs for data parallel pipelines. In: *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures — Padua, Italy*. New York, NY: ACM Press, 1996. p. 62–71.

SZYPERSKI, Clemens. *Component Software: Beyond Object-Oriented Programming*. Reading, Massachusets: Addison-Wesley, 2002.

TANENBAUM, Andrew S.; STEEN, Maarten van. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ: Prentice-Hall, 2002.

THAUT, M. H.; TIAN, B.; AZIMI-SADJADI, M. R. Rhythmic finger tapping to cosine-wave modulated metronome sequences: Evidence of subliminal entrainment. *Human Movement Science*, v. 17, p. 839–863, 1998.

THE AUSTIN GROUP. *IEEE Std 1003.1–2001, 2003 Edition*. 2003. Available at: <http://www.unix-systems.org/version3/online.html> Accessed at: 5 apr 2004.

TRIGGS, Thomas J.; HARRIS, Walter G. *Reaction Time of Drivers to Road Stimuli*. [S.l.], 1982. Available at: <http://www.general.monash.edu.au/muarc/rptsum/hfr12.htm> Accessed at: 15 feb 2004.

VIEIRA, Jorge Euler. *LINUX-SMART: Melhoria de desempenho para aplicações real-time soft em ambiente LINUX*. Dissertation (Masters) — IME/USP, São Paulo, 1999.

[VST] <http://www.steinberg.net/en/ps/support/3rdparty/vst_sdk/index.php?sid=0>. *Website of the VST specification*. Accessed at: 24 jul 2003.

[VSTSERVER] <http://www.notam02.no/arkiv/src>. *Website of the VSTserver program*. Accessed at: 5 apr 2004.

WESSEL, David; WRIGHT, Matthew. Problems and prospects for intimate musical control of computers. *Computer Music Journal*, v. 26, n. 3, p. 11–22, 2002.

WING, Alan M. Perturbations of auditory feedback delay and the timing of movement. *Journal of Experimental Psychology: Human Perception and Performance*, v. 3, n. 2, p. 175–186, 1977.

WRIGHT, Matthew; FREED, Adrian. *Open SoundControl: A New Protocol for Communicating with Sound Synthesizers*. 1997. Available at: <http://www.cnmat.berkeley.edu/ICMC97/papers-html/OpenSoundControl.html> Accessed at: 5 apr 2004.

WRIGHT, Matthew; FREED, Adrian; MOMENI, Ali. OpenSound Control: State of the art 2003. In: *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME—03)*. Montreal, Canada: [s.n.], 2003. Available at: <http://www.cnmat.berkeley.edu/Research/NIME2003/NIME03_Wright.pdf> Accessed at: 5 apr 2004.

[XORG] <http://www.x.org>. *Website of the X Consortium, responsible for the X Windowing System*. Accessed at: 5 apr 2004.