Nelson Posse Lago

Processamento Distribuído de Áudio em Tempo Real

Dissertação apresentada ao Instituto de Matemática e Estatística da Universidade de São Paulo como requisito parcial para obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. Fabio Kon

Durante a elaboração deste trabalho, o autor teve apoio financeiro da CAPES

São Paulo, Abril de 2004

Sumário

1 Introdução **1**

1.1 Justificativa . . . 2

1.1.1 O computador e a linguagem musical . . . 2

1.1.2 O crescimento dos estúdios domésticos . . . 3

1.1.3 As dificuldades dos sistemas fechados e de alto custo . . . 4

1.1.4 Software livre . . . 4

1.2 Limitações das soluções correntemente em uso . . . 6

1.2.1 Soluções *ad hoc* . . . 7

1.2.2 Hardware dedicado . . . 7

1.2.3 Processamento paralelo . . . 8

1.3 O sistema DLADSPA . . . 9

1.3.1 Baixa latência . . . 10

1.3.2 Operação síncrona . . . 11

1.3.3 Sistema operacional . . . 11

1.3.4 Aplicações legadas . . . 12

1.3.5 Modularização . . . 13

1.3.6 Distribuição de algoritmos seqüenciais . . . 13

1.3.7 Middleware . . . 14

1.4 Trabalhos relacionados . . . 15

I Conceitos relevantes **19**

2 Latência e percepção **20**

2.1 Latência em sistemas interativos . . . 20

2.1.1 Origens da latência e da agitação em sistemas computacionais . . . 22

2.2 Efeitos da latência e da agitação sobre a percepção . . . 22

2.3 Latência, agitação e sincronização . . . 24

2.3.1 Percepção rítmica . . . 24

2.3.2 Percepção da simultaneidade entre sons . . . 25

2.3.3 Percepção da relação temporal entre ação e reação . . . 26
2.4 Desenvolvimento e baixa latência . . . 28

3 Processamento baseado em funções *callback* 30
3.1 Interação entre o computador e o hardware de E/S . . . 30
3.1.1 Sistemas de E/S baseados em interrupções . . . 30
3.1.2 Escalonamento de processos e E/S . . . 31
3.2 Latência em dispositivos de áudio . . . 32
3.3 *Callbacks* . . . 33
3.3.1 JACK . . . 34
3.3.2 LADSPA . . . 37

4 Sistemas distribuídos de tempo real 40
4.1 Um modelo para o processamento computacional . . . 40
4.2 Processamento paralelo e distribuído . . . 42
4.2.1 Sistemas distribuídos . . . 42
Middleware . . . 43
4.2.2 Processamento paralelo . . . 44
Abordagens para o processamento paralelo . . . 44
4.3 Sistemas de tempo real . . . 45
4.3.1 Prazos rígidos e flexíveis . . . 46
4.3.2 Sistemas de tempo real rígido e flexível . . . 47
4.3.3 Sistemas de tempo real firme . . . 48
4.3.4 Caracterização de sistemas de tempo real . . . 49
4.4 Processamento paralelo em tempo real . . . 52

II O sistema 55

5 O middleware 56
5.1 Mecanismos de funcionamento . . . 56
5.1.1 Tempo real . . . 57
5.1.2 Transmissão de blocos de dados . . . 58
5.1.3 Operação síncrona . . . 58
Protocolo de rede orientado a mensagens . . . 60
Agendamento de tempo real . . . 60
5.1.4 Janelas deslizantes . . . 61
Redes e tempo real . . . 61
Melhorias para a utilização da rede . . . 63
5.1.5 Processamento isócrono . . . 65

5.1.6 Integração com CORBA . . . 66
5.2 Implementação . . . 67
5.2.1 Abstração do protocolo de rede . . . 67
5.2.2 Encapsulamento dos *buffers* de dados . . . 69
Buffers de tamanho variável . . . 70
Ordenação de bytes (*endianness*) . . . 72
5.2.3 Gerenciamento e agrupamento dos *buffers* de dados . . . 72
5.2.4 Processamento remoto . . . 73

6 A aplicação DLADSPA **76**
6.1 Distribuição de módulos LADSPA . . . 76
6.1.1 Compatibilidade com aplicações legadas . . . 76
6.1.2 Conjuntos de módulos . . . 78
6.1.3 Editores . . . 79
6.1.4 Servidores de módulos LADSPA . . . 80
6.2 Implementação . . . 80
6.2.1 Encapsulamento de módulos LADSPA . . . 81
Processamento de mais de um fluxo de áudio . . . 81
Encadeamento de módulos . . . 82
6.2.2 Interação com aplicações LADSPA . . . 83
6.2.3 Módulos LADSPA remotos . . . 84
6.2.4 Servidores remotos e comunicação . . . 84
Proxy, manager e plugin_factory . . . 85
6.2.5 Reutilização do código . . . 86
6.3 Resultados experimentais . . . 87
6.3.1 Ambiente de software e hardware . . . 87
6.3.2 Limitações dos experimentos . . . 88
6.3.3 Resultados . . . 89
6.3.4 Discussão dos resultados . . . 91

7 Conclusão **93**
7.1 Trabalhos futuros . . . 94
7.1.1 Middleware . . . 94
7.1.2 Aplicação DLADSPA . . . 96

Bibliografia **97**

Lista de Figuras

3.1 A interface LADSPA 38

4.1 Grafos podem representar tarefas computacionais 41

5.1 Comunicação sem o uso de janelas deslizantes 62
5.2 Comunicação com o uso de janelas deslizantes 65
5.3 Classes para a comunicação entre as máquinas 68
5.4 *Buffers* de dados e as funções readv() e writev() 71
5.5 As classes do middleware 74

6.1 O processador de efeitos jack-rack. 79
6.2 IDL do servidor plugin_factory. 86
6.3 Relação entre período e utilização do processador 89
6.4 Eficiência do sistema sem janelas deslizantes 90
6.5 Utilização do processador nas máquinas remotas 91
6.6 Carga na máquina central em função do número de máquinas remotas 92

Resumo

Sistemas computadorizados para o processamento de multimídia em tempo real demandam alta capacidade de processamento. Problemas que exigem grandes capacidades de processamento são comumente abordados através do uso de sistemas paralelos ou distribuídos; no entanto, a conjunção das dificuldades inerentes tanto aos sistemas de tempo real quanto aos sistemas paralelos e distribuídos tem levado o desenvolvimento com vistas ao processamento de multimídia em tempo real por sistemas computacionais de uso geral a ser baseado em equipamentos centralizados e monoprocessados. Em diversos sistemas para multimídia há a necessidade de baixa latência durante a interação com o usuário, o que reforça ainda mais essa tendência para o processamento em um único nó.

Neste trabalho, implementamos um mecanismo para o processamento síncrono e distribuído de áudio com características de baixa latência em uma rede local, permitindo o uso de um sistema distribuído de baixo custo para esse processamento. O objetivo primário é viabilizar o uso de sistemas computacionais distribuídos para a gravação e edição de material musical em estúdios domésticos ou de pequeno porte, contornando a necessidade de hardware dedicado de alto custo.

O sistema implementado consiste em duas partes: uma, genérica, implementada sob a forma de um middleware para o processamento síncrono e distribuído de mídias contínuas com baixa latência; outra, específica, baseada na primeira, voltada para o processamento de áudio e compatível com aplicações legadas através da interface padronizada LADSPA. É de se esperar que pesquisas e aplicações futuras em que necessidades semelhantes se apresentem possam utilizar o middleware aqui descrito para outros tipos de processamento de áudio bem como para o processamento de outras mídias, como vídeo.

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.

Para a Tati,
que mudou tudo.

Gostaria de agradecer:

à CAPES, pelo apoio financeiro
que viabilizou este trabalho;

aos membros da banca, particularmente
ao Prof. Iazzetta, pelas valiosas sugestões;

aos meus pais, pela paciência;

ao Marko, pelo incentivo
em embarcar nessa jornada;

e, especialmente, ao Fabio,
pelo apoio excepcional.

Capítulo 1

Introdução

O uso do computador na produção de multimídia já há muito deixou de ser exclusividade de centros de pesquisa científica ou artística; seu uso é visível em várias áreas, como vinhetas e comerciais de televisão, estúdios de gravação de música popular, exposições multimídia para a demonstração de produtos, jogos computadorizados e novas formas de arte e entretenimento baseadas diretamente nas possibilidades abertas pelo uso do computador.

Uma aspecto fundamental para o uso de computadores em várias aplicações relacionadas à multimídia é a operação interativa em tempo real, ou seja, a possibilidade de interagir diretamente com o computador durante a produção ou reprodução de sons ou imagens, alterando suas características. A interação em tempo real, além de simplificar o uso dos sistemas computacionais por usuários cuja capacitação primária não é na área da Ciência da Computação, abre as portas para novos mecanismos de criação, execução e recepção de comunicações e artes baseadas em sistemas multimídia[1].

Um outro aspecto que se apresenta em aplicações interativas é a necessidade de funcionamento não apenas em tempo real, mas também com baixa latência, ou seja, com tempos pequenos de resposta entre a inserção de um novo dado no sistema computadorizado e a produção do resultado correspondente. Nesse contexto, o tempo de resposta de um sistema é pequeno o suficiente se for imperceptível para o usuário que interage com o sistema, garantindo a ilusão de que a resposta do sistema é instantânea.

Sistemas multimídia, de maneira geral, operam com pelo menos uma forma de mídia contínua (Steinmetz e Nahrstedt, 1995, p. 13, 17, 571) e, em geral, com várias, o que significa que o volume de dados processados é geralmente alto. Aplicações que dependem de grande volume de processamento são tradicionalmente abordadas com o uso de técnicas de programação paralela ou distribuída; no caso de sistemas de tempo real onde a baixa latência é um requisito essencial, no entanto, o uso de sistemas paralelos e distribuídos se mostra significativamente mais complexo, o que tem impedido que soluções desse tipo sejam adotadas em produtos de uso geral, não baseados em hardware dedicado.

Neste trabalho, desenvolvemos um mecanismo para o processamento distribuído de áudio em uma rede local em tempo real com garantias de baixa latência (nos experimentos, obtivemos boa eficiência

[1] Em Roads (1996, p. 105), o autor aborda diversas vezes as interfaces homem-máquina capazes de operação em tempo real, principalmente nos capítulos 14 e 15, e chega a dizer que "non-real-time software synthesis is 'the hard way' to make music".

com latências menores que $7ms$, mas outras configurações de hardware podem reduzir ainda mais a latência do sistema) compatível com aplicações legadas em ambiente Linux utilizando computadores e equipamentos genéricos e de baixo custo. O uso de um sistema distribuído dispensa equipamentos multiprocessados, que têm alto custo, e permite ao usuário dimensionar gradativamente seu sistema de acordo com suas necessidades e possibilidades simplesmente agregando novas máquinas ao sistema.

O sistema desenvolvido é baseado em duas camadas: a primeira, genérica, permite a distribuição transparente de dados entre máquinas em tempo real com baixa latência; a segunda, específica, aborda a comunicação com sistemas legados de edição de áudio, através da especificação LADSPA, para o processamento distribuído de áudio. Procurou-se desenvolver a camada genérica sob a forma de um middleware (Seção 1.3.7) reutilizável. Esse middleware permite o processamento distribuído de tarefas periódicas no processamento de mídias contínuas em tempo real com baixa latência. Outras pesquisas no futuro podem utilizar e aprimorar esse middleware ou mesmo determinar sua inadequação para contextos diferentes do processamento de áudio. Uma possível aplicação para esse middleware é o processamento de áudio e vídeo em ambientes multimídia interativos ou em *performances* envolvendo multimídia em tempo real.

Graças aos experimentos realizados, pudemos constatar que o sistema permite utilizar efetivamente pelo menos oito, e provavelmente mais de dez máquinas interligadas por uma rede local padrão Fast Ethernet com equipamentos de baixo custo. Essa solução oferece, portanto, uma excelente relação custo/benefício em comparação a outros sistemas disponíveis que buscam maximizar a capacidade de processamento de sistemas para a manipulação de áudio em tempo real.

1.1 Justificativa

O crescimento do papel do computador na linguagem musical, aliado à sua popularização como elemento básico para a criação de estúdios domésticos, suscita o interesse no desenvolvimento de sistemas de baixo custo voltados para o uso do computador nesses ambientes. De forma similar, o computador tem sido utilizado também em eventos ao vivo para o processamento de sinais de áudio (e também vídeo) em tempo real. Um dos aspectos mais importantes dos sistemas computacionais tipicamente usados nessas aplicações são suas limitações intrínsecas de desempenho e o alto custo das soluções disponíveis para contorná-las. Portanto, o uso de um sistema distribuído baseado em software livre, como alternativa de baixo custo, se mostra altamente interessante nesses cenários. Nosso sistema foi desenvolvido com base em um middleware (Seção 1.3.7) especificamente projetado para permitir a comunicação em rede com baixa latência e pode vir a ser reutilizado em outros projetos.

1.1.1 O computador e a linguagem musical

O computador tem sido usado em um grande número de aplicações voltadas para a música:

- Cada vez mais, mesmo a música realizada por instrumentos tradicionais (seja ao vivo ou em gravações) é processada, no mínimo, por módulos processadores de sinais como equalizadores,

reverberadores, compressores etc. Em estúdios de gravação, sejam de pequeno, médio ou grande porte, o computador já é uma das ferramentas de trabalho cotidianas, e muitas vezes central: operações de edição, que antigamente dependiam de manipulações diretas das fitas magnéticas de gravação, são rotineiramente realizadas através do computador; diversos tipos de correção do sinal gravado (como a eliminação de distorções pontuais ou a adequação do nível de gravação, por exemplo) podem ser executados rapidamente e praticamente sem degradação do sinal; e a paleta de efeitos disponíveis para processamento de áudio é facilmente expansível através da instalação de novos pacotes de software, dispensando boa parte dos dispositivos de hardware dedicados a tarefas específicas usados comumente há poucos anos.

- A síntese sonora através do uso de computadores e outros equipamentos digitais já há muito foi incorporada tanto pela música erudita quanto por diversos gêneros da música popular e de mercado.

- O próprio processo de composição musical foi alterado com a introdução de ferramentas computadorizadas para a composição musical, algumas baseadas em conceitos abstratos de programação, como o MAX ([MAX]; [JMAX]), Common Music ([COMMONMUSIC]) etc., outras baseadas em paradigmas musicais tradicionais, como seqüenciadores MIDI (***M****usical* ***I****nstrument* ***D****igital* ***I****nterface* — [MIDI]; Kientzle, 1998, p. 273–318).

Essa introdução do computador em tantos campos da criação e produção musicais tem por conseqüência a transformação contínua da linguagem musical[2]. Assim, embora a tecnologia digital evidentemente não seja uma condição *sine qua non* para a criação ou produção musical, sua disponibilidade é, sem dúvida, cada vez mais desejável.

1.1.2 O crescimento dos estúdios domésticos

A tecnologia para o processamento de áudio tem se difundido com a oferta crescente de computadores de uso geral com grande capacidade de processamento e equipados com dispositivos de áudio de excelente qualidade disponibilizados por preços razoavelmente baixos; esses computadores se tornaram acessíveis como ferramenta de uso diário para compositores e músicos em geral, que podem pagar por equipamentos desse tipo para usá-los em suas casas ou em apresentações ao vivo. Esse uso cotidiano, por sua vez, reforça a influência da tecnologia digital sobre a linguagem musical.

A disponibilidade desses equipamentos tem tornado possível, ao músico e a pequenos produtores, a criação de estúdios domésticos com custos de implantação e manutenção muito menores que os de um estúdio convencional e capazes de oferecer resultados com qualidade competitiva (Oppenheimer et al., 1999). Se, no passado, gravadores multipista baseados em fitas cassete possibilitavam ao músico produzir gravações "demo" de baixa qualidade, úteis apenas como um passo intermediário no caminho

[2]De fato, se tomarmos como exemplo a produção de artistas populares brasileiros importantes em atividade há bastante tempo, como Caetano Veloso, Chico Buarque e Gilberto Gil, poderemos observar mudanças na sonoridade de seus trabalhos que são fruto dos avanços nas tecnologias de gravação de áudio, já há algum tempo conduzidos por processadores digitais de sinal e, mais recentemente, por computadores.

para um estúdio profissional, estúdios domésticos atuais baseados em computadores têm possibilitado ao músico produzir gravações domésticas de diversos tipos de trabalho com qualidade suficiente para a distribuição e comercialização. De forma similar, a presença do computador em diversas produções musicais realizadas ao vivo (às vezes aliadas a outras formas de expressão, como vídeo, teatro etc.) é crescente. Com o crescimento da Internet e de seu potencial tanto como meio de divulgação quanto de distribuição, é de se esperar que esse novo contexto tecnológico incentive o mercado de música independente a crescer, com os conseqüentes crescimento e diversificação da cultura musical.

1.1.3 As dificuldades dos sistemas fechados e de alto custo

Diversas aplicações musicais de sistemas computadorizados envolvem o processamento do áudio em tempo real; no entanto, o processamento de um grande volume de dados em tempo real exige uma grande capacidade de processamento por parte do equipamento, impossibilitando, às vezes, a operação em tempo real.

A solução comumente adotada por estúdios de médio e grande porte é a aquisição de hardware específico e dedicado. Como veremos na Seção 1.2, tal solução tem diversas desvantagens, em particular o preço dos equipamentos e a incompatibilidade com sistemas de software livre. O uso de um sistema distribuído de processamento baseado em software livre é uma alternativa de baixo custo ao uso desses equipamentos.

1.1.4 Software livre[3]

O sistema aqui descrito foi desenvolvido com base no sistema operacional Linux e visa à compatibilidade com os softwares livres disponíveis para ele voltados para o trabalho musical. A adoção do Linux e do software livre foi influenciada pelos diversos benefícios (técnicos, sociais e econômicos) normalmente associados ao software livre, em particular no Brasil (Kon, 2001; Raymond, 2001; Silveira e Cassino, 2003), tais como:

- o software pode ser obtido e mantido atualizado por baixo custo, reduzindo custos em empresas e instituições de pesquisa e possibilitando o investimento em outras áreas, como, por exemplo, a compra de equipamentos (como microfones, amplificadores etc. ou instrumentos musicais no caso de pequenos estúdios);

- o software livre também pode representar uma grande economia na importação de software para o Brasil e, ao mesmo tempo, ampliar o espaço para o desenvolvimento local de software;

- no caso da pesquisa em computação, softwares livres já disponíveis podem muitas vezes evitar a necessidade de desenvolvimento das partes de um sistema periféricas ao objeto específico da pesquisa. Por exemplo, não é necessário desenvolver um sistema operacional completo para realizar experimentos com um novo algoritmo de agendamento de tempo real para multimídia: basta

[3]Para uma visão geral sobre o software livre e sobre as diversas licenças que se enquadram nessa definição, veja os sítios da *Free Software Foundation* ([FSF]) e da *Open Source Initiative* ([OPENSOURCE]).

adaptar um sistema já existente, como o Linux. A disponibilização do software desenvolvido durante a pesquisa, por sua vez, facilita seu uso por parte de outros pesquisadores;

- o uso do software livre em instituições de pesquisa no Brasil ainda pode elevar o conhecimento brasileiro na área da computação, graças ao contato facilitado com código-fonte de alta qualidade e à troca de informações e experiências entre pequisadores de diversos países atuando sobre softwares desenvolvidos de forma aberta;

- Tem se tornado claro que softwares livres são capazes de oferecer excelentes níveis de qualidade e desempenho, e o seu modelo de desenvolvimento colabora para uma grande velocidade na implementação de novos recursos aliada à manutenção de um baixo número de falhas de programação (*bugs*);

- o software livre, hoje, tem se configurado como uma das poucas alternativas viáveis ao monopólio atualmente existente no mercado de sistemas operacionais e aplicativos de escritório para microcomputadores; esse monopólio coloca a economia e o estado em situação fragilizada, em especial no caso de países em desenvolvimento como o Brasil;

- o fato de praticamente não haver restrições à redistribuição do software pode incentivar uma maior colaboração tanto entre programadores quanto entre usuários, facilitando o desenvolvimento e a troca de experiências no uso de sistemas computacionais e, talvez, promovendo um espírito comunitário que pode ter reflexos benéficos na sociedade de maneira geral;

- finalmente, o software livre tem sido apontado como peça fundamental no processo de *inclusão digital* das populações carentes, ou seja, nos processos de democratização do acesso aos sistemas computacionais e à informação disponibilizada através deles e da apropriação do universo digital enquanto ferramenta de trabalho e meio de comunicação individual e de massa por parte dessas populações.

Além disso, outras vantagens colaboraram para a escolha do Linux como plataforma básica para o sistema aqui descrito:

- dado o interesse intrínseco no software livre discutido acima, é interessante promover o crescimento do uso do software livre em áreas como a produção musical, pois isso pode colaborar para o seu crescimento em outras áreas;

- as implementações dos principais programas relacionados (inclusive, por exemplo, o servidor de áudio jackd) estão disponíveis sob licenças de software livre, possibilitando a alteração de algum desses programas caso necessário ou interessante;

- de forma similar, todos os padrões envolvidos são abertos e desenvolvidos pela comunidade de usuários; isso pode possibilitar a incorporação e adoção imediata das técnicas desenvolvidas na pesquisa, o que dificilmente aconteceria no caso de especificações controladas por uma empresa não diretamente envolvida com o movimento de software livre;

- as semelhanças entre os principais padrões de software usados para o processamento digital de sinais de áudio (tanto ASIO, CoreAudio e JACK quanto VST e LADSPA — Seções 3.3.1 e 3.3.2) sugerem que não deve ser demasiadamente difícil reimplementar as técnicas levantadas neste trabalho para os padrões usados em outras arquiteturas caso as empresas responsáveis se interessem. No entanto, mesmo que elas não o façam, o sistema já está à disposição dos usuários praticamente sem nenhum custo;

- o sistema operacional em si é altamente portátil, além de ser amplamente compatível com os padrões POSIX[4], viabilizando a implementação dessas mesmas técnicas em outras plataformas de hardware, seja através do próprio Linux, seja através da adaptação para outros sistemas operacionais semelhantes ao UNIX;

- o sistema operacional também é altamente estável e, com um dos *patches* de baixa latência disponíveis[5], oferece níveis de latência extremamente baixos (MacMillan, Droettboom e Fujinaga, 2001), comparáveis ao melhor que pode ser obtido com o MacOS X, que é um sistema tradicionalmente voltado para o mercado de artes gráficas e multimídia;

- finalmente, sistemas distribuídos naturalmente dependem de diversos computadores e, portanto, de diversas cópias do sistema operacional; logo, o uso de um sistema operacional não-livre teria um impacto significativo no preço de um estúdio doméstico baseado em um sistema distribuído como o descrito aqui.

1.2 Limitações das soluções correntemente em uso

Como vimos, boa parte do uso que se faz de computadores em aplicações musicais envolve o processamento do áudio em tempo real; de fato, a experimentação é fundamental na produção e criação musicais, o que depende diretamente do processamento em tempo real no caso de sistemas digitais. No entanto, o tratamento de um grande volume de dados em tempo real exige uma grande capacidade de processamento por parte do equipamento; alguns algoritmos em particular são tão pesados que mal podem ser executados em tempo real, e muitas aplicações dependem da execução de vários algoritmos simultaneamente sobre um grande número de canais de áudio, impossibilitando completamente o processamento em tempo real. A despeito da crescente velocidade dos equipamentos disponíveis no mercado, é de se esperar que esse problema não seja resolvido apenas pelo aumento na velocidade do hardware a médio prazo.

[4]POSIX é um conjunto de especificações para sistemas operacionais; seu objetivo é facilitar a transposição de aplicações entre diferentes sistemas operacionais compatíveis com essas especificações. As especificações são desenvolvidas por um grupo de trabalho do IEEE, cujo sítio é acessível em [POSIX].

[5]O *patch* de Ingo Molnar ([LLPATCH2.2]) foi desenvolvido para as versões 2.2.X do núcleo do Linux; o *patch* de Andrew Morton ([LLPATCH2.4]), por sua vez, foi desenvolvido para as versões 2.4.X do núcleo do Linux; o *patch* de Robert Love ([LLPATCH2.6]), que promove a operação em baixa latência por permitir que interrupções de hardware sejam capazes de forçar mudanças no contexto de execução do próprio código do núcleo do Linux, é compatível com as versões 2.4.X do Linux e foi incorporado ao Linux a partir das versões 2.6.X.

1.2.1 Soluções *ad hoc*

A solução comumente utilizada para contornar os limites do equipamento em ambientes onde soluções de maior custo não são viáveis é simplesmente abrir mão do processamento em tempo real e utilizar processamento em lote. O usuário define quais passos devem ser executados para obter um resultado e solicita o processamento em lote de todos os passos; o sistema executa o processamento e grava os resultados em disco (alguns editores de áudio oferecem a opção desse tipo de funcionamento). Essa solução, no entanto, é extremamente inconveniente, pois dificulta enormemente a experimentação e impossibilita o uso do sistema interativamente em eventos ao vivo.

O uso de sistemas biprocessados oferece ganhos de desempenho sem que haja necessidade de alteração significativa do código para operação paralela. Num sistema com dois processadores, tarefas como E/S (Entrada e Saída) de disco, atualização da interface com o usuário, escalonamento de tarefas pelo núcleo etc. podem ser executadas simultaneamente ao processamento do áudio, permitindo praticamente 100% de aproveitamento da capacidade de um processador para esse processamento. Para que esse aproveitamento seja maximizado, é interessante que as aplicações a serem executadas dividam seu processamento em *threads* voltadas para as tarefas de tempo real e as outras tarefas; felizmente, esse tipo de divisão é típico. Apesar dessa possibilidade, sistemas biprocessados não têm sido comumente utilizados, provavelmente porque os sistemas operacionais de uso geral mais comumente usados no mercado, Windows e MacOS, apenas recentemente incorporaram suporte a multiprocessamento. É de se esperar, no entanto, que seu uso venha a crescer; os desenvolvedores do editor Ardour ([ARDOUR]), para Linux, explicitamente recomendam o uso de sistemas biprocessados. Ainda assim, esses sistemas possibilitam a otimização do uso de um dos processadores para o processamento em tempo real, mas estão limitados à capacidade de processamento desse único processador. Para que haja benefícios na utilização de equipamentos com mais processadores, é necessário que o software seja desenvolvido especificamente para tirar proveito dos processadores adicionais, por exemplo dedicando uma *thread* de processamento para cada fluxo de áudio a ser processado. No entanto, a despeito dessa possibilidade, sistemas com mais de dois processadores ainda têm custo relativamente alto, o que inviabiliza seu uso em estúdios domésticos e inibe o desenvolvimento de programas capazes de operar eficientemente em sistemas multiprocessados.

1.2.2 Hardware dedicado

Outra solução para o problema, normalmente adotada em estúdios de médio e grande porte, é o uso de hardware específico e dedicado. Tais equipamentos, no entanto, apresentam algumas desvantagens, em particular para pequenos estúdios:

- Por tratar-se de produtos importados e especializados, o custo desse tipo de equipamento é bastante elevado; por exemplo, uma única placa processadora para o sistema Pro Tools HD (que é o produto líder de mercado em todo o mundo) custa, no país de origem (EUA), mais de quatro

vezes o valor de um PC de nível médio completo[6]. No Brasil, esse preço ainda sofre o acréscimo dos impostos de importação e custos com o transporte.

- Sistemas desse tipo são baseados em arquiteturas fechadas; salvo alguns periféricos, não há compatibilidade entre equipamentos, programas e acessórios de diferentes fabricantes. Assim, o comprador de um produto desse tipo está sempre sujeito aos interesses de mercado de um fabricante específico para obter outros produtos complementares a aqueles que ele já possui ou serviços, como assistência técnica.

- Embora esses sistemas ofereçam um excelente nível de desempenho dentro do âmbito dos problemas a que se destinam, eles são projetados com vistas a aplicações específicas. Computadores de uso geral, embora muitas vezes ofereçam níveis de desempenho inferior a soluções baseadas em hardware dedicado, podem ser programados mais facilmente e se adequar a aplicações mais diversas que eles. Além disso, fatores econômicos e tecnológicos têm conduzido ao crescimento exponencial na capacidade de processamento dos sistemas de uso geral, o que tem reduzido cada vez mais a diferença de desempenho entre sistemas genéricos e específicos.

- Até o momento, nenhum dos sistemas existentes é compatível com Linux ou outros sistemas operacionais de código livre; assim, seu uso implica no uso de sistemas de uso restrito (em geral, Windows ou MacOS). Isso, por sua vez, afasta os sistemas baseados em software livre tanto de desenvolvedores quanto de usuários, graças à indisponibilidade de equipamentos profissionais compatíveis com eles.

O uso de sistemas distribuídos de processamento baseados no Linux e em computadores de uso geral como alternativa a esses sistemas dedicados elimina essas deficiências, em particular o custo, permite o prolongamento da vida útil de computadores relativamente obsoletos e abre caminho para o crescimento do software livre em aplicações musicais, com os seus conseqüentes benefícios (conforme visto na Seção 1.1.4).

1.2.3 Processamento paralelo

As soluções comumente usadas para o aumento na capacidade de processamento de áudio descritas acima diferem das soluções tradicionalmente utilizadas para a solução de problemas que envolvem grandes capacidades de processamento ou E/S; em geral, problemas dessa natureza costumam ser resolvidos através da paralelização ou da distribuição da carga.

No caso do processamento de multimídia, existem diversas oportunidades para a paralelização do código:

- novos algoritmos poderiam ser desenvolvidos com vistas à paralelização ou à distribuição; esses algoritmos seriam especificamente otimizados para se beneficiar de múltiplos processadores.

[6]Preços consultados em <http://www.gateway.com> e <http://www.protools.com> em abril, 2004.

Algoritmos desse tipo poderiam aplicar diferentes operações sobre os dados paralelamente, em diferentes processadores, e combinar os resultados dessas operações. Por exemplo, um sinal de áudio poderia ser processado simultaneamente por diversos processadores, cada um encarregado de processar uma faixa de freqüências, caso isso se mostrasse eficiente para o algoritmo em questão;

- diferentes mídias, como áudio e vídeo, podem ser processadas paralelamente: basta alocar processadores ou máquinas independentes para cada mídia;

- muitas vezes a origem do problema de desempenho está no fato de diversos fluxos de dados independentes de uma mesma mídia, como os diversos canais de áudio em um estúdio, precisarem ser processados simultaneamente. O processamento desses fluxos também pode ser feito paralelamente, da mesma forma que o processamento de diferentes mídias;

- mesmo um conjunto de algoritmos aplicados seqüencialmente sobre um único fluxo de dados (um canal de áudio ou vídeo) pode ser distribuído em um conjunto de máquinas ou processadores e executado paralelamente: basta encadear as máquinas ou processadores de maneira que cada máquina ou processador atue sobre os dados já processados pela máquina ou processador imediatamente anterior a si, como em uma linha de produção (*pipeline* — Butenhof, 1997, p. 98–105). O resultado é simplesmente um aumento na latência do processamento em relação ao que haveria se todo o processamento fosse realizado em uma única máquina ou processador.

A despeito dessas possibilidades oferecidas para a paralelização, o uso de processamento paralelo em aplicações de áudio (e de multimídia em geral) em tempo real não é comum. Como vimos, o uso de sistemas multiprocessados foi até recentemente inibido pela incompatibilidade com os sistemas Windows e MacOS; mesmo com esse suporte presente, o alto custo de sistemas com mais de dois processadores sugere que eles continuarão a ser utilizados apenas em algumas poucas aplicações específicas durante muito tempo[7]. Já a distribuição entre máquinas requer a sincronização periódica entre as máquinas envolvidas; o volume de dados e a necessidade de baixa latência intrínsecas ao processamento de multimídia se apresentam como dificuldades reais para a sincronização nesse tipo de cenário.

1.3 O sistema DLADSPA

Com vistas a viablilizar o processamento distribuído de áudio em tempo real com baixa latência em uma rede local, desenvolvemos o sistema DLADSPA (Distributed LADSPA), que está disponível na web sob a licença GPL em http://gsd.ime.usp.br/software/DistributedAudio. Diversos aspectos foram levados em conta no desenvolvimento do sistema:

- A preocupação com a baixa latência é particularmente importante em sistemas interativos.

- O funcionamento do sistema deve seguir um modelo de processamento síncrono.

[7] Máquinas biprocessadas, que têm custo menos elevado, constituem um caso especial pois, como vimos, apresentam benefícios sem que haja a necessidade de algoritmos paralelos nas aplicações.

- O sistema deve ser compatível com sistemas operacionais de uso geral.
- A compatibilidade com aplicações legadas, bem como o uso de módulos (*plugins*) pré-existentes para o processamento de sinais (ao invés do desenvolvimento de novos algoritmos paralelos) através do uso de interfaces padronizadas, evitam a necessidade de alteração do código das aplicações já existentes e facilitam sua adoção por desenvolvedores e usuários atuais.
- Finalmente, a arquitetura em camadas do sistema permite que a camada mais baixa possa ser usada como um middleware para o desenvolvimento de outras aplicações distribuídas com necessidades de tempo real e baixa latência.

1.3.1 Baixa latência

Em diversos sistemas computacionais para o processamento de multimídia, como sistemas para edição e criação de conteúdo multimídia, para *performance* interativa ao vivo ou para reconhecimento de padrões em mídias contínuas, é altamente desejável que o processamento seja realizado não apenas em tempo real (ou seja, o processamento é realizado simultaneamente à produção do sinal original), mas também com baixa latência. Em sistemas de baixa latência, a diferença temporal entre o sinal de entrada e o resultado correspondente na saída é "pequena"; o tempo máximo que ainda pode ser considerado "pequeno" o suficiente depende da aplicação (Capítulo 2).

Um exemplo simples onde o processamento em tempo real com baixa latência é desejável é a gravação de um instrumento musical tradicional processado por algum tipo de efeito digital (por exemplo, uma guitarra elétrica processada por um distorcedor digital). Ao tocar o instrumento, o músico precisa ouvir o som que está sendo produzido; se a latência do processamento for demasiadamente grande, o músico terá dificuldades para a execução.

Garantias de baixa latência também são importantes se se deseja que parte do processamento de um sistema seja realizado em tempo real por outros dispositivos além do computador. Por exemplo, um sinal de áudio previamente capturado pode ser direcionado para um processador analógico de efeitos e o resultado pode ser gravado novamente pelo computador sem que haja diferença perceptível na disposição temporal da informação.

No caso de sistemas interativos multimídia, a latência do sistema geralmente pode ser considerada pequena o suficiente se não for perceptível para o usuário, garantindo a ilusão de que o sistema responde imediatamente (ou quase). Essa ilusão é extremamente importante em sistemas desse tipo, já que nesses sistemas o usuário modifica sua interação com o computador de acordo com os estímulos que recebe dele. Esse limiar varia fortemente tanto entre diferentes usuários quanto entre diferentes tipos de aplicações; ainda assim, como a latência das diversas partes de um sistema se somam, é interessante procurar garantir que cada uma delas seja a menor possível. O sistema desenvolvido neste trabalho funcionou eficientemente nos experimentos com latências menores que $7ms$, mas outras configurações de hardware provavelmente podem reduzir ainda mais esse nível de latência.

1.3.2 Operação síncrona

Diversos sistemas para o processamento e a comunicação via rede de multimídia implementam mecanismos para garantir a sincronização entre diferentes mídias ou diferentes fluxos de uma mesma mídia. Por exemplo, uma aplicação para vídeo-conferência ou para a apresentação de vídeo e som simultâneos normalmente implementa mecanismos para garantir a sincronização entre o áudio e o vídeo. Esses mecanismos são necessários graças às imprecisões no hardware de E/S de mídia ou aos atrasos inerentes à comunicação via rede.

Normalmente, podemos encarar cada um dos fluxos de dados em um sistema multimídia como uma série de eventos dispostos no tempo (por exemplo, quadros de vídeo ou amostras de áudio) de maneira que, para cada um desses fluxos de dados, a passagem do tempo está associada ao número de eventos processados. Assim, em função da taxa de amostragem e do número de eventos anteriores, pode-se definir com precisão o momento no tempo esperado para que um determinado evento ocorra[8].

No entanto, a taxa de amostragem nominal de um fluxo de mídia qualquer não corresponde exatamente à taxa de amostragem com a qual essa mídia é processada, graças a imprecisões no hardware. De forma similar, atrasos na comunicação via rede podem influenciar o momento em que um evento é efetivamente processado. Portanto, é necessário realizar correções periodicamente no processamento desses fluxos de dados para compensar as eventuais diferenças entre o momento em que um evento deveria ser processado e o momento em que ele efetivamente é processado; caso isso não seja feito, fluxos diferentes podem divergir entre si no tempo. Essa compensação envolve a modificação na taxa de amostragem usada durante o processamento dos diversos fluxos de dados, por exemplo, através da inserção ou remoção de amostras.

A necessidade desse tipo de compensação tem um efeito indesejável: ao executar um mesmo processamento mais de uma vez, diferentes resultados podem ser gerados. Em grande parte das aplicações multimídia, isso não afeta a qualidade da aplicação; no entanto, em sistemas para a edição profissional de mídia, essa característica não é aceitável. É preciso que haja sincronização entre todos os fluxos de dados no nível das amostras. Para que isso seja possível, os diversos fluxos de dados devem ser processados de acordo com uma única referência para a passagem do tempo; ou seja, o processamento de todos os fluxos de dados deve ser síncrono. Como veremos adiante, isso é possível com o uso de funções *callback*.

1.3.3 Sistema operacional

A necessidade de flexibilidade e de facilidade de uso em sistemas para o processamento de áudio sugerem que, embora algumas tarefas específicas possam ser realizadas por hardware dedicado (como conversores analógico-digitais), um computador de uso geral com um sistema operacional também de uso geral oferece diversas vantagens para esse tipo de aplicação. Tais sistemas:

[8] Em sistemas de vídeo-conferência esse mecanismo é mais complexo, mas a necessidade de correções temporais, discutida a seguir, permanece.

- possibilitam que o sistema seja baseado em módulos de software (como discutido abaixo na Seção 1.3.5) e permitem atualizações de forma simplificada;
- permitem que novos algoritmos possam ser desenvolvidos com linguagens de alto nível, o que por sua vez facilita aos usuários do sistema trocar módulos entre si (como acontece entre os usuários de MAX);
- oferecem interfaces sofisticadas com o usuário, geralmente baseadas em sistemas gráficos que fazem excelente uso dos recursos disponíveis (normalmente tela, mouse e teclado) com convenções bem definidas e conhecidas para a interação;
- oferecem um grande conjunto de serviços do sistema operacional para o programador;
- costumam ter custos menores que sistemas baseados em hardware dedicado, como, por exemplo, os produtos da família Pro Tools (que são, de fato, híbridos, já que se utilizam de um computador de uso geral para armazenamento de dados e interface com o usuário);
- podem oferecer excelentes desempenho e latência para aplicações voltadas para multimídia, que possuem requisitos temporais na casa dos milissegundos, apesar de não poderem competir com sistemas dedicados a tarefas de tempo real com requisitos temporais na casa dos microssegundos.

Assim, por causa dessas razões e pelas razões descritas na Seção 1.1.4, optamos por desenvolver o nosso sistema com base no sistema operacional Linux.

1.3.4 Aplicações legadas

A popularização recente do uso do computador em aplicações voltadas para a área musical se deu principalmente através de aplicativos disponíveis para as plataformas MacOS e Windows (tais como o MAX [MAX], PatchWork [PATCHWORK], Pro Tools [PROTOOLS], etc.) mas, com o crescente interesse pelo sistema operacional Linux, este também começa a contar com diversos aplicativos voltados à produção e pesquisa musicais (Metts, 2004; Phillips, 2000)[9]. Uma parte dessas ferramentas tem origem em sistemas experimentais desenvolvidos originalmente para plataformas baseadas em UNIX (como o Ceres [CERES], MixViews [MIXVIEWS] e outros), mas há diversos esforços recentes voltados especificamente para o Linux (como o Ardour [ARDOUR], Muse [MUSE], Gstreamer [GSTREAMER] e outros).

Adaptar essas diversas aplicações diretamente para operação em sistemas distribuídos envolveria um trabalho relativamente extenso por parte de seus desenvolvedores. Muito pouco desse trabalho poderia ser feito de forma genérica ou implementado em bibliotecas comuns a várias delas, o que levaria a uma parcela significativa de código adicional em cada uma delas. Devido a esses problemas, é de se esperar que vários desenvolvedores optariam por não dar suporte a processamento distribuído em seus projetos.

[9]Phillips também mantém uma lista bastante ampla de aplicações voltadas para áudio e música compatíveis com o Linux, disponível em [SOUNDAPPS].

No entanto, várias dessas aplicações (a maioria das aplicações ativamente desenvolvidas) podem ser expandidas pelo uso de módulos adicionais (*plugins*) através de módulos LADSPA; de fato, em várias delas, a maior parte do processamento realizado é efetivamente delegada para diversos módulos LADSPA. Assim, um mecanismo para o processamento distribuído e transparente acessível através dessa interface pode permitir que todas as aplicações compatíveis com ela se beneficiem do processamento distribuído sem alteração em seu código. Dentre essas aplicações, uma das mais interessantes é o programa Ardour ([ARDOUR]), que oferece características sofisticadas de edição de áudio, similares às oferecidas pelos sistemas da família Pro Tools, que são líderes de mercado.

1.3.5 Modularização

Uma tendência que tem se consolidado recentemente nos aplicativos voltados para o tratamento do áudio é a modularização: de fato, diversas especificações padronizadas para a criação de módulos (*plugins*) para o processamento de áudio existem, e aplicações complexas que possuam suporte a essas especificações, como gravadores ou seqüenciadores, podem se beneficiar dos novos módulos que têm sido criados regularmente. Essa tendência segue um movimento geral em direção ao desenvolvimento baseado em componentes, que possibilita a criação de aplicações mais flexíveis e expansíveis (Szyperski, 2002).

No ambiente Linux, a especificação LADSPA (Seção 3.3.2) define uma interface padrão para o encapsulamento de algoritmos de DSP (***D****igital* ***S****ignal* ***P****rocessing*, ou processamento digital de sinais) para áudio (como sintetizadores, processadores de efeitos como reverberação ou distorção etc.). Módulos escritos de acordo com essa especificação são carregados como bibliotecas dinâmicas pelos aplicativos e são executados pelo processador do computador no próprio contexto de execução do aplicativo que carregou o módulo; o processamento é geralmente feito em tempo real. Diversas aplicações que executam em Linux têm suporte a módulos que seguem essa especificação; um grande número de módulos já existe e muitos outros estão em desenvolvimento.

Como veremos a seguir, a adoção da interface especificada no padrão LADSPA oferece diversas vantagens para o sistema aqui descrito.

1.3.6 Distribuição de algoritmos seqüenciais

Dentre os mecanismos para o processamento paralelo elencados na Seção 1.2.3, apenas um depende do desenvolvimento de algoritmos especificamente paralelos; todos os outros são baseados na composição de algoritmos seqüenciais para atingir o resultado desejado.

Embora o desenvolvimento de novos algoritmos paralelos para o processamento de áudio seja interessante, essa não é a melhor forma de garantir a paralelização na maioria dos casos e muito menos de forma genérica, pois:

- o desenvolvimento de algoritmos paralelos é mais complexo que o desenvolvimento de algoritmos seqüenciais;

- já existe um grande número de algoritmos seqüenciais para o processamento de áudio desenvolvidos e implementados; certamente muitos mais que algoritmos paralelos. Muitos deles provavelmente não podem ser eficientemente paralelizados;

- problemas diferentes em geral são resolvidos por diferentes abordagens para a paralelização (Chaudhuri, 1992, p. 52–56; Gibbons e Rytter, 1988, p. 6–19; Carriero e Gelernter, 1990, p. 14–20); isso significa que o desenvolvimento de um algoritmo paralelo provavelmente não simplifica o desenvolvimento de outros;

- diferentes abordagens para a paralelização são comumente associadas a diferentes sistemas de hardware capazes de implementar eficientemente as abordagens definidas no nível do software. Portanto, uma única topologia de rede pode não ser flexível o suficiente para permitir implementações eficientes de algoritmos paralelos quaisquer;

- na maioria dos casos, os limites do hardware são atingidos não pela execução de um único algoritmo extremamente complexo, mas sim pela execução de um grande número de algoritmos sobre um grande número de fluxos de dados. Nesses casos, a paralelização de algoritmos individuais não traria benefícios em relação à mera distribuição de diferentes tarefas independentes entre diferentes processadores ou máquinas, mas traria um aumento desnecessário de complexidade;

- no caso do processamento em tempo real, quaisquer algoritmos paralelos dependeriam de algum mecanismo de comunicação e sincronização temporal entre suas partes; no caso do processamento distribuído em diferentes máquinas, seria necessário desenvolver uma infra-estrutura para permitir esse tipo de comunicação. Essa mesma infra-estrutura pode ser usada para permitir a distribuição de tarefas independentes entre máquinas de forma sincronizada; ou seja, mesmo a busca de paralelização de algoritmos depende do desenvolvimento de uma infra-estrutura que poderia ser usada para a mera composição de algoritmos.

Por essas razões, antes de investigar a paralelização e distribuição de algoritmos individuais, optamos por procurar soluções que permitam a distribuição sincronizada de algoritmos já existentes para o processamento paralelo de fluxos de áudio distintos. Tais mecanismos devem permitir a distribuição da carga sem a necessidade de algoritmos complexos. Com a grande quantidade de módulos LADSPA já disponíveis, um mecanismo para processamento distribuído que seja capaz de fazer uso desses módulos pode oferecer, de imediato, todas as funcionalidades desses módulos conjugadas aos benefícios de um sistema distribuído.

1.3.7 Middleware

O sistema aqui descrito procura viabilizar o processamento distribuído de áudio em uma rede local em tempo real e com baixa latência; é de se esperar, no entanto, que o processamento distribuído de outras mídias contínuas possa se beneficiar de mecanismos similares aos desenvolvidos aqui. Também em outras áreas não diretamente ligadas a sistemas interativos, como o reconhecimento de padrões de

imagens em tempo real, o processamento distribuído com baixa latência pode se mostrar uma abordagem viável. É interessante, portanto, que seja possível reutilizar ao máximo os resultados deste trabalho em outros contextos.

O processamento distribuído envolve interações entre sistemas computacionais independentes; juntamente com as interações esperadas com os recursos de hardware disponíveis localmente, protocolos de comunicação de rede estão necessariamente envolvidos em sistemas desse tipo. O desenvolvimento de aplicações que levam em conta as características da rede, no entanto, aumenta a complexidade dessas aplicações. Como conseqüência, para facilitar o desenvolvimento de aplicações distribuídas, tem se tornado cada vez mais comum o uso de camadas de abstração de software que visam a tornar a interação entre máquinas indepententes num sistema distribuído razoavelmente transparente. Essas camadas são comumente designadas *middleware* (Tanenbaum e Steen, 2002, p. 36–42), por consistirem em uma camada intermediária (ou seja, que fica no "meio" — *middle*) entre a aplicação e o sistema operacional.

Assim, com vistas a possibilitar a reutilização do código desenvolvido neste trabalho, o sistema foi desenvolvido em duas partes:

1. um middleware genérico para o processamento distribuído de mídias contínuas em tempo real com baixa latência;

2. uma aplicação voltada para o processamento de áudio baseada nesse middleware.

A aplicação desenvolvida demonstra a viabilidade do middleware, e é de se esperar que ele venha a ser utilizado para o desenvolvimento de outros tipos de aplicação, abrindo espaço para mais pesquisas em sistemas distribuídos de tempo real onde a latência é um fator importante.

1.4 Trabalhos relacionados

Diversos produtos, sistemas e protocolos envolvendo redes e multimídia já existem. Boa parte deles são voltados para a comunicação de dados, como sistemas para transmissão (*streaming*) de áudio e vídeo ou vídeo-conferência. Os sistemas que mais se aproximam do trabalho aqui desenvolvido são:

VST System Link: VST System Link (Carlson, Nordmark e Wiklander, 2003, p. 639–655) é uma especificação, desenvolvida pela empresa Steinberg, para a interconexão de computadores em estúdios para o processamento distribuído de áudio e MIDI. O sistema permite que diversas máquinas operem sincronizadamente sobre diversos fluxos de áudio e MIDI simultaneamente e que esses fluxos sejam enviados de uma máquina para outra. Em cada uma das máquinas é executada uma aplicação completa, com sua interface de usuário etc. No entanto, o VST System Link automaticamente envia comandos como "play", "stop", "rewind" etc. entre as máquinas conectadas pelo sistema, minimizando a necessidade de interação do usuário com a interface de todas as máquinas do sistema para as tarefas mais simples.

As diversas máquinas são conectadas entre si através de um anel, onde a saída de áudio digital

de uma máquina é ligada à entrada da máquina seguinte até que o anel seja fechado. Um dos canais de áudio é usado, juntamente com a transmissão de áudio, para a transmissão dos dados de sincronismo, localização e MIDI do sistema. Dados de áudio produzidos em uma máquina podem ser enviados para serem processados em outra; de forma similar, o sistema oferece a possibilidade do envio de mensagens MIDI entre as máquinas. Todo o processamento é realizado de maneira efetivamente síncrona: como o sistema depende da presença de placas de som capazes de utilizar sincronismo externo em todas as máquinas do sistema, as requisições de interrupções de todas as máquinas ocorrem simultaneamente. No entanto, a comunicação de áudio entre as máquinas utiliza um esquema similar ao mecanismo de janelas deslizantes descrito na Seção 5.1.4; assim, como o sistema interconecta as máquinas em um anel, o aumento no número de máquinas intermediárias entre duas máquinas que trocam dados entre si aumenta proporcionalmente o acréscimo de latência do sistema.

Por um lado, o VST System Link oferece a possibilidade de fazer uso de um grande número de máquinas com um grande número de canais de áudio em cada máquina, permitindo que um volume de dados que não poderia ser gerido por uma única máquina seja utilizado pelo sistema. Isso é possível porque cada máquina do sistema pode operar de maneira independente, já que todas possuem dispositivos para E/S de áudio. Nesse tipo de utilização, no entanto, o VST System Link opera apenas como um mecanismo para a sincronização entre as máquinas. Quando utilizado como um mecanismo para a comunicação de dados de áudio entre máquinas, o VST System Link oferece a possibilidade de comunicação entre aplicações completas; a mesma funcionalidade poderia ser oferecida pelo sistema aqui descrito adaptando-se o middleware para operação com o sistema JACK. Por outro lado, o VST System Link não permite a operação de modo mais transparente para o usuário, como o nosso sistema, além de depender do uso de hardware com várias entradas e saídas de áudio digitais, que são de alto custo. Finalmente, o uso de interconexões baseadas em dispositivos de áudio digital limita o número de canais que podem ser transferidos entre as máquinas em função do hardware de áudio disponível, além de ter um efeito significativo sobre a latência do sistema se um número relativamente grande de máquinas (5–10) for utilizado.

OSC: OSC (***O****pen* ***S****ound* ***C****ontrol* — Wright e Freed, 1997; Wright, Freed e Momeni, 2003) é uma especificação para a transmissão de mensagens, similar ao MIDI mas muito mais sofisticada, voltada para o uso em redes de velocidade razoavelmente alta (a partir de 10Mbps). O mecanismo básico oferecido pelo sistema OSC é o de envio de mensagens independentes com níveis razoavelmente baixos de latência. Uma mensagem é composta por um endereço de destino e um conjunto de argumentos, juntamente com seus tipos. Alguns tipos padrão são definidos na especificação (como cadeias de caracteres, inteiros de 32 bits etc.), mas outros tipos podem ser usados. Isso significa que os tipos de mensagens que podem ser enviadas não são definidos *a priori* pela especificação, tornando o sistema extensível.

Uma entidade que recebe mensagens é um servidor; cada servidor tem um endereço e oferece um espaço de endereçamento interno, ou seja, uma árvore de pontos de destino em potencial para uma

mensagem. Assim, o endereço de uma mensagem especifica um servidor de destino e um ponto de destino dentro desse servidor. Cada um desses pontos possui um nome simbólico, de maneira que é possível usar nomes descritivos para os endereços, como "/voice/3/freq" para definir um ponto de destino de uma mensagem. Além disso, é possível que uma mensagem seja enviada a múltiplos destinatários através do uso de expressões regulares no endereço.

Cada mensagem também leva consigo uma marca temporal, que é usada para definir o momento em que o efeito causado pela mensagem deve ocorrer; OSC permite que a informação temporal referente aos eventos representados por mensagens seja quão precisa quanto necessário. Portanto, o uso das marcas temporais permite que a latência do sistema de comunicação seja compensada pela aplicação. No caso de aplicações interativas de tempo real, evidentemente há um limite para essa precisão causado pela latência tanto na comunicação via rede quanto no processamento necessário ao envio e recebimento das mensagens. No entanto, em redes locais baseadas em padrões de alta velocidade, como Fast Ethernet, esses atrasos são pequenos o suficiente (da ordem de poucos milissegundos) para serem considerados irrelevantes. Além disso, graças às marcas temporais, variações na latência (que, como veremos no Capítulo 2, podem ser problemáticas) podem ser compensadas com um pequeno aumento na latência do sistema. Por outro lado, o uso das marcas temporais torna o sistema dependente de um mecanismo, como o uso de NTP (Mills, 1992), para a manutenção da coerência temporal entre as máquinas do sistema. Tipicamente, um mecanismo desse tipo precisa promover ajustes periódicos no horário de um sistema; esses ajustes podem ter efeitos sobre a precisão temporal da aplicação.

Os desenvolvedores da especificação oferecem uma implementação de referência que pode ser usada como base para o desenvolvimento de aplicações que façam uso de OSC; nesse sistema, o desenvolvedor define as entidades que farão parte da árvore de endereços do servidor e define funções *callback* que devem ser executadas quando do recebimento de uma mensagem.

Assim, OSC é voltada para o desenvolvimento de aplicações musicais distribuídas com baixa latência; no entanto, a especificação não é voltada especificamente para o uso com mídias contínuas e, portanto, não oferece nenhum mecanismo similar ao mecanismo de janelas deslizantes descrito na Seção 5.1.4 nem oferece mecanismos para a operação síncrona. O agrupamento de dados em mensagens completas (normalmente encapsuladas em pacotes UDP) é similar ao mecanismo usado em nosso sistema; no entanto, as outras funcionalidades oferecidas têm pouco a acrescentar ao sistema aqui descrito.

RTP: RTP (***R**eal-**T**ime **P**rotocol* — Schulzrinne et al., 2003; Schulzrinne e Casner, 2003) é um protocolo voltado para a transmissão de multimídia em redes, em particular mídias contínuas em redes de grande abrangência geográfica, como a Internet. A aplicação principal do protocolo é em sistemas para vídeo-conferência, mas diversas outras aplicações utilizam RTP. A especificação define um formato de pacote genérico que deve ser usado para transmitir praticamente qualquer tipo de dado e utiliza especificações adicionais para definir como os diferentes tipos de mídia que podem ser transmitidos são encapsulados nesse formato de pacote genérico. RTP também

define um outro protocolo, RTCP (***R**eal-**T**ime **C**ontrol **P**rotocol*), usado para diversas tarefas relacionadas à comunicação efetivada pelo protocolo RTP, como negociar o início e fim de uma conexão ou oferecer a identificação de um usuário. Um dos principais usos desse protocolo é avaliar a qualidade do serviço oferecido a cada um dos clientes, mas ele também pode ser usado para oferecer serviços adicionais, como a listagem de nomes dos membros de uma vídeo-conferência ou a possibilidade de avançar ou retroceder a apresentação de uma mídia pré-gravada.

Pacotes RTP carregam consigo duas informações temporais: um contador, que identifica a posição do pacote em relação aos outros pacotes da seqüência, e uma marca temporal, que indica o momento no tempo em que o pacote deve ser reproduzido. Esse mecanismo garante precisão temporal e permite que pacotes que não contenham informação relevante sejam suprimidos, ao mesmo tempo em que oferece a possibilidade de detecção de perdas de pacotes. Através do protocolo RTCP, a detecção de perdas de pacotes pode ser usada para que sejam usados mecanismos adaptativos para adaptar a comunicação às condições da rede disponíveis.

O protocolo RTP tem sido bastante usado na comunicação de multimídia em sistemas de rede; no entanto, ele não é adequado para facilitar o processamento distribuído de multimídia. Como trata-se de processamento onde não deve haver perda de pacotes, os mecanismos oferecidos para controle da passagem do tempo e administração da qualidade de serviço não são de interesse para essa aplicação.

MPI: MPI (***M**essage **P**assing **I**nterface* — Message Passing Interface Forum, 1995, 1997; Foster, 1995, Capítulo 8) é uma especificação de um sistema para a comunicação voltada para o processamento paralelo. No caso de sistemas computacionais interligados por uma rede local, MPI permite que diversos nós troquem mensagens para o processamento distribuído de dados. MPI oferece diversas funções que simplificam o desenvolvimento de aplicações paralelas baseadas em troca de mensagens, gerindo automaticamente a criação de processos, oferecendo mecanismos para definir pontos de sincronização, permitindo o agrupamento de processos em categorias e, principalmente, oferecendo diversos mecanismos para a troca de mensagens entre os diversos nós responsáveis por uma tarefa. No entanto, MPI dificilmente poderia ser usada de forma compatível com aplicações legadas; além disso, não é voltada para o processamento em tempo real e não contempla o uso de funções *callback*, podendo oferecer dificuldades para o processamento com baixa latência.

Parte I

Conceitos relevantes

Capítulo 2

Latência e percepção

O tempo que um sistema qualquer demora para apresentar algum tipo de reação ou resultado correspondente a um dado ou estímulo recebido é chamado de *latência* do sistema; variações na latência de um sistema são chamadas de *agitação* (*jitter*). Latência e agitação são características importantes em diversos tipos de sistemas computacionais; vários deles, como sistemas voltados para o reconhecimento de padrões em mídias contínuas ou sistemas interativos multimídia, precisam ser capazes de operar com baixa latência e baixa agitação no processamento. Por causa dessa característica, tais sistemas são sistemas de tempo real (Seção 4.3) .

Neste capítulo, discutiremos os papéis da latência e da agitação em sistemas interativos. Nesse tipo de sistema, latência e agitação se relacionam com a percepção do usuário do sistema; assim, abordaremos a percepção dessas grandezas. Em particular, falaremos sobre a percepção auditiva em geral e em contextos musicais, já que a percepção auditiva tem grande precisão temporal e o sistema objeto deste trabalho é voltado para aplicações musicais.

É importante observar que os dados experimentais disponíveis oferecem uma visão geral das relações entre latência e percepção, mas ainda não são suficientes para que se possa definir limites mais claros para a latência e a agitação em diversas situações. Além disso, muitos dados foram coletados fora de contextos musicais, dificultando a tarefa de avaliar a qualidade de sistemas multimídia e musicais com relação à precisão temporal. Como as qualidades temporais da música estão tão intimamente ligadas a aspectos da música ainda pouco compreendidos como a "pegada" e como parte de nossa percepção temporal ocorre de forma subconsciente, é necessário que novos experimentos sejam realizados em contextos musicais.

2.1 Latência em sistemas interativos

No caso de sistemas não-interativos, as características desejáveis de um sistema em relação à latência e à agitação podem normalmente ser definidas com razoável precisão em função das características da aplicação desse sistema. Por exemplo, um sistema de visão computacional para o controle dos movimentos de um robô móvel deve ser capaz de promover alterações na trajetória desse robô para

impedir colisões; as características de latência e agitação desse sistema estão vinculadas à velocidade máxima que esse robô pode desenvolver em um determinado ambiente. Ainda assim, essa relação pode ser flexível; por exemplo, se o processamento em algum momento demora mais tempo que o esperado, o robô pode temporariamente diminuir sua velocidade[1].

No caso de sistemas interativos, as características de latência e agitação adequadas são determinadas pela interação com o usuário (Shneiderman, 1984): latências ou agitações muito altas (ou, em alguns casos, muito baixas) podem atrapalhar o desempenho do usuário ou, no mínimo, oferecer uma experiência frustrante e cansativa (Barber e Lucas, Jr., 1983). Por exemplo, o usuário de um sistema que se utiliza de uma interface gráfica vai ter dificuldades em utilizar esse sistema se o apontador na tela reagir aos movimentos do mouse com um atraso significativo ou com grandes variações nesse atraso. De forma similar, se esse sistema não reagir rapidamente aos cliques do mouse, o usuário terá a tendência a realizar o mesmo clique mais de uma vez. Assim, para analisar a qualidade de um sistema interativo em relação a suas características de latência e agitação, é necessário compreender os seus efeitos sobre a percepção do usuário.

Os limites aceitáveis para a latência e a agitação de um sistema interativo podem variar grandemente (Miller, 1968). No exemplo acima, a latência do sistema aos movimentos do mouse deve ser bastante pequena; a latência para que o sistema ofereça uma reação visível a um clique do mouse pode ser maior; e a latência para o término da tarefa solicitada pelo clique (por exemplo, abrir uma nova janela) pode ser ainda maior. Aplicações interativas voltadas para multimídia são normalmente as que exigem valores mais baixos para a latência e a agitação, já que normalmente envolvem pelo menos uma mídia contínua que pode ser modificada pela interação com o usuário. Mas mesmo em sistemas multimídia há variações nos limites aceitáveis para a latência e a agitação: a audição humana tem maior precisão temporal que a visão (Repp, 2003), e a precisão temporal envolvendo estímulos de tipos diferentes (como visual e auditivo ou auditivo e tátil) geralmente é menor que a precisão temporal envolvendo um único tipo de estímulo (Levitin et al., 1999).

A maior precisão temporal da audição e sua relevância para a música fazem da latência e da agitação aspectos importantes no desenvolvimento de diversos sistemas para computação musical. Em muitos casos, tais sistemas são desenvolvidos de forma a produzir latências e agitações as mais baixas possíveis, o que corresponde hoje a alguns milissegundos em sistemas de custo médio. No entanto, diversas aplicações, em particular envolvendo os atrasos provocados por redes de abrangência geográfica ampla (WANs), não podem oferecer latências típicas abaixo de $10ms$, e é muito comum que estejam limitadas a latências muito maiores. Ainda assim, essas aplicações são de grande interesse e são, portanto, desenvolvidas a despeito das características de latência e agitação supostamente sub-ótimas que oferecem.

Embora muito já se saiba sobre a percepção da latência e da agitação, ainda são necessárias mais pesquisas para que possamos compreender melhor as diversas relações entre latência, agitação, ação e

[1] De fato, isso é exatamente o que um ser humano faz quando dirige um automóvel: se um motorista demora um tempo maior que o normal para identificar a distância e velocidade de um objeto à sua frente, ele reduz a velocidade.

percepção humanas. Por exemplo, seria difícil discordar que a música "pop"[2] tipicamente precisa de um nível de sincronização entre instrumentistas maior que a música baseada em texturas que são alteradas gradativamente ao longo do tempo. Mas, ainda assim, não há dados definitivos que permitam dizer com segurança quais os limites adequados para a latência e a agitação em cada um desses cenários ou em outros. É preciso definir com clareza, em diferentes situações, os limites típicos para que latência e agitação sejam perceptíveis, afetem o desempenho de um instrumentista, degradem a experiência de um usuário de sistema multimídia, afetem o desempenho humano de diferentes tarefas etc. Esse conhecimento, por sua vez, possibilitará que seja possível avaliar a qualidade de aplicações musicais em relação à latência e à agitação. A seguir, ofereceremos uma visão geral sobre o que já se sabe hoje sobre esse assunto.

2.1.1 Origens da latência e da agitação em sistemas computacionais

Um sistema interativo precisa ser capaz de processar dados que emanam do usuário (ou do ambiente) ao longo do tempo; isso geralmente significa que ele deve ser capaz de capturar amostras desses dados de alguma maneira com freqüência adequada. De acordo com os dados capturados, o sistema deve realizar algum tipo de processamento para produzir o resultado correspondente e, finalmente, apresentá-lo para o usuário. A soma do período da taxa de amostragem, do tempo de processamento e do tempo para a saída dos dados corresponde à latência máxima do sistema.

Diversos fatores podem contribuir para que haja variações nessa latência, ou seja, para que haja agitação:

imprecisões do hardware: se há variações na taxa de amostragem do sistema, por exemplo, essas variações podem se refletir em variações no tempo total de processamento do sistema;

variações no tempo de processamento: se o tempo de processamento dos dados capturados não é constante (por exemplo, varia com os dados capturados ou de acordo com o agendamento de tarefas do sistema), a latência de processamento do sistema também pode variar;

taxa de amostragem: eventos que ocorrem em momentos diferentes podem ser capturados num mesmo instante pelo sistema se o tempo entre eles for menor que o período da taxa de amostragem; assim, a taxa de amostragem do sistema é um limite inferior para a agitação do sistema.

2.2 Efeitos da latência e da agitação sobre a percepção

Aparentemente, diversos mecanismos são envolvidos na percepção auditiva de eventos no tempo, vinculados a diferentes níveis de processamento e com diferentes resultados para a percepção consciente (Pierce, 1999, p. 95). Assim, as diversas formas de latência e agitação de um sistema computacional podem ter diversos efeitos sobre a percepção; alguns deles não são conscientemente associados ao tempo:

[2]Refiro-me aqui à musica ocidental de mercado, que é baseada em um pulso isócrono bastante regular e explícito, tornando flutuações temporais facilmente perceptíveis. A escolha pela expressão "música pop" ao invés da mais comum "música popular" se deve ao fato de que esta última pode ser entendida como englobando as músicas étnica e folclórica.

- Sistemas de amostragem de áudio tipicamente utilizam uma taxa de amostragem de 44.1KHz; o período de amostragem, portanto, é de cerca de $22.7\mu s$. No entanto, diferenças temporais entre os dois ouvidos tão pequenas quanto 10–$20\mu s$ podem ser utilizadas para a definição da posição espacial de uma fonte sonora; portanto, a existência de agitação na taxa de amostragem de áudio pode ter efeitos negativos, por exemplo, em sistemas de áudio onde há simulação de posicionamento espacial de fontes sonoras. A origem dessa agitação está na imprecisão do hardware, e não se pode fazer muito para reduzi-la a não ser aumentar a precisão do hardware ou a taxa de amostragem (como de fato tem ocorrido em sistemas sofisticados, que têm utilizado taxas de amostragem de 96KHz e 192KHz). Esse tipo de agitação, no entanto, não tem nenhuma relação direta com o aspecto interativo de um sistema; a interatividade não acrescenta dificuldades ao controle desse tipo de agitação. Assim, ela não é relevante para esta discussão.

- Eventos sonoros similares separados por intervalos de tempo pequenos e constantes podem ter seus timbres alterados pelo fenômeno de *comb filtering*, onde freqüências cujos períodos são submúltiplos do tempo de separação entre os eventos podem ser realçadas ou atenuadas. A latência de processamento de um sistema pode dar origem a esse fenômeno durante o processamento de um sinal sonoro se o som original e o processado forem reproduzidos simultaneamente. Esse problema, no entanto, não ocorre comumente: na maioria das aplicações, não é necessário que os dois sinais sejam reproduzidos, bastando que se utilize o sinal processado; em outros casos, pode-se introduzir um atraso artificial ao sinal original para eliminar a defasagem temporal entre os dois sinais. De fato, provavelmente a única situação em que esse fenômeno não pode ser eliminado é a situação em que um sinal sonoro gerado em tempo real é recebido diretamente pelo ouvinte (por exemplo, como ocorre no caso de instrumentos acústicos) e, ao mesmo tempo, é processado e reproduzido por equipamentos eletrônicos. Nessa situação, no entanto, o *comb filtering* resultante é similar ao que também ocorre naturalmente em diversos ambientes acústicos (D'Antonio, s.d.): graças à diferença no posicionamento espacial das fontes sonoras[3], o efeito varia conforme a posição do ouvinte e é mais significativo apenas durante o início de cada evento sonoro, pois a reverberação e demais características acústicas do ambiente amenizam o *comb filtering* nesses casos. Assim, a semelhança com o que ocorre normalmente em ambientes comuns sugere que, nesse tipo de cenário, o *comb filtering* pode ser geralmente ignorado. De qualquer modo, não há muito que possa ser feito com relação a ele a não ser atenuar um dos dois eventos sonoros, já que qualquer valor de latência, grande ou pequeno, irá dar origem ao *comb filtering*; diferentes latências apenas alteram as faixas de freqüências afetadas.

- Variações na separação temporal da ordem de 1–2ms entre dois eventos quase simultâneos, como em *flams* percussivos, também podem alterar a sensação timbrística desses eventos (Wessel e Wright, 2002). O *comb filtering* certamente tem papel decisivo nesse fenômeno mas, nesse caso, seu efeito é esperado e está, supostamente, sob o controle do instrumentista. Assim, o controle

[3]No caso do efeito provocado pelas características acústicas do ambiente, cada reflexão do som atua como uma fonte sonora secundária.

de um sistema digital sobre eventos sonoros deve oferecer níveis de agitação menores que $1ms$ se eventos desse tipo precisam ser representados com fidelidade. Como veremos mais adiante, agitações da ordem de 4–$6ms$ podem ter outros efeitos perceptivos; assim, minimizar a agitação (possivelmente através do acréscimo de latência) é geralmente interessante.

À parte esses casos excepcionais, os problemas causados pela agitação e pela latência em um sistema interativo estão vinculados à percepção da sincronia: a latência e a agitação podem impedir que percebamos eventos que deveriam parecer ser simultâneos como tais. Isso, por sua vez, pode afetar a interação com o sistema.

2.3 Latência, agitação e sincronização

A idéia de sincronização de eventos sonoros está intimamente ligada à noção do ataque desses eventos. É importante lembrar, no entanto, que é difícil definir o momento exato do ataque de um som: o que é comumente chamado de ataque de um som percussivo, por exemplo, costuma ter cerca de $10ms$ de duração, e outros tipos de som podem ter ataques muito mais longos. Essa é uma dificuldade adicional para a pesquisa nessa área (Bilmes, 1993); ainda assim, podemos assumir que nosso sistema perceptivo associa um momento no tempo ao início de um som e é capaz de avaliar com razoável precisão e decidir se dois eventos são simultâneos ou não. Nosso interesse aqui é inferir as características desse mecanismo.

Pares de eventos que podem precisar ser percebidos como simultâneos em um sistema musical podem ser divididos em três grandes categorias em relação ao usuário do sistema:

- Um pulso isócrono externo e um pulso isócrono interno (ou seja, a relação entre uma estrutura musical baseada em uma pulsação constante e a pulsação correspondente induzida no usuário);
- Um par de eventos externos (como pares de notas executadas por diferentes instrumentistas, ou ataques sonoros que ocorrem simultaneamente a piscadas de luzes);
- Uma ação do usuário e o efeito correspondente (como ocorre no caso da execução de um instrumento musical).

Cada uma dessas categorias tem características próprias perante o sistema perceptivo.

2.3.1 Percepção rítmica

Uma parte importante da percepção humana é a percepção rítmica, e o ritmo é evidentemente um aspecto importante em diversas aplicações musicais. Essa é uma área em que a percepção e a ação humanas se mostram extremamente precisas, embora nem sempre de maneira consciente. Experimentos demonstraram que podemos executar um pulso isócrono com variações típicas no tempo entre as batidas tão baixas quanto $4ms$ (Rubine e McAvinney, 1990). De maneira similar, ao executar um pulso em

sincronia com um estímulo isócrono, podemos ajustar nosso ritmo para compensar variações no estímulo da ordem de $4ms$ (Repp, 2000) e somos capazes de detectar conscientemente imprecisões temporais de cerca de $6ms$ em um estímulo isócrono (Friberg e Sundberg, 1995) em algumas condições. Se essas variações são cíclicas e um pouco maiores, da ordem de $10ms$, nós espontaneamente interiorizamos essas variações e as executamos em conjunto com o estímulo (e não apenas corrigimos as batidas subseqüentes a uma variação detectada) (Thaut, Tian e Azimi-Sadjadi, 1998). Esses tipos de ajuste são geralmente realizados de maneira subconsciente; no entanto, é bastante provável que essas variações temporais sejam percebidas conscientemente como algum tipo de qualidade musical, como a chamada "pegada". De fato, há fortes indícios de que músicos introduzem tais variações de acordo com o contexto musical (Bilmes, 1993; Clynes, 1994, 1995).

Experimentos sugerem que essa percepção rítmica é baseada na comparação entre os tempos esperado e real para cada ataque sonoro (Schulze, 1978); essa hipótese é reforçada pelo fato de que a precisão em acompanhar variações rítmicas não é significativamente afetada se o ritmo é executado fora de fase (ou seja, no contratempo) (Repp, 2001). Isso, por sua vez, significa que a percepção de variações rítmicas da ordem de 10–$20ms$ não é baseada na percepção de fenômenos acústicos resultantes de pequenas diferenças no momento do ataque de sons próximos; ao contrário, essa alta precisão no tocante ao ritmo significa que somos capazes de avaliar intervalos de tempo e momentos de ataque sonoro com precisão da ordem de $4ms$ em nível subconsciente e que irregularidades temporais dessa magnitude muito possivelmente afetam a sensação de "pegada" de alguns tipos de música (primariamente músicas baseadas em pulsação isócrona bastante regular e explícita, como diversos tipos de música "pop"). Assim, fica claro que é importante minimizar a agitação de um sistema para computação musical, especialmente se esse sistema precisar dar suporte a esse tipo de música.

Um caso especial da sincronização entre um pulso interno e um externo existe quando o pulso externo, ao invés de se manter constante, procura se adaptar ao pulso do usuário (expresso por algum tipo de ação do usuário). Um exemplo simples é o que ocorre quando da sincronização rítmica entre diferentes músicos executando seus instrumentos: o pulso de todos os instrumentistas deve estar sincronizado e, para cada participante, o pulso dos outros instrumentistas é externo. Existem experimentos preliminares a esse respeito (Schuett, 2002), mas os resultados foram um tanto inconsistentes; novas pesquisas precisam ser realizadas. Ainda assim, aparentemente atrasos da ordem de até $20ms$ entre dois executantes parecem não interferir significativamente no seu desempenho.

2.3.2 Percepção da simultaneidade entre sons

Seria fácil imaginar que a precisão de nossa percepção temporal descrita anteriormente implique na necessidade de garantir que eventos que devem ser percebidos como simultâneos devam de fato ocorrer com assincronias entre si menores que $4ms$. No entanto, assincronias de até $50ms$ em notas supostamente simultâneas não são, de forma alguma, incomuns durante a execução de peças musicais ordinárias. Em conjuntos de câmara, onde os efeitos causados pela distância entre os instrumentistas são mínimos, assincronias de até $50ms$ são comuns (Rasch, 1979). No caso de uma orquestra, as seções de percussão

e metais podem se encontrar mais de $10m$ mais distantes da platéia que a seção dos violinos, o que resulta em assincronias adicionais de cerca de $30ms$ entre elas para o público, sem contar as assincronias reais entre os instrumentos. De forma similar, as diferenças de dinâmica entre as vozes em peças para piano são responsáveis pelo que é normalmente chamado *melody lead*, ou *adiantamento da melodia*: as notas da melodia tipicamente soam cerca de $30ms$ antes das outras notas supostamente simultâneas a elas[4]. A despeito da característica percussiva do som do piano (que resulta em tempos de ataque pequenos e, portanto, em ataques facilmente discerníveis), essas assincronias não são percebidas como tal pelos instrumentistas ou pelo público. Finalmente, ao executar um ritmo isócrono em sincronia com um estímulo metronômico, sujeitos praticamente sempre executam o ritmo cerca de 10–$80ms$ adiantado (tipicamente, $30ms$) sem perceber (Aschersleben, 2002). Esses valores são surpreendentes, dado que o limiar para a percepção da relação de ordem entre dois sons distintos é de cerca de $20ms$ (Pierce, 1999, p. 95).

Esses fatos sugerem que latências responsáveis por assincronias de até pelo menos $30ms$ em eventos externos podem ser consideradas normais e aceitáveis na maioria dos casos; o desempenho de instrumentistas com instrumentos tradicionais não é afetado por elas. De fato, tais assincronias são usadas pelo sistema auditivo no processo de identificação de sons simultâneos; sua ausência reduz grandemente nossa capacidade de distinguir notas simultâneas (Rasch, 1978).

Pode-se argumentar que, mesmo que essas assincronias não sejam conscientemente percebidas, elas podem ter um papel em um contexto musical e podem estar, ao menos parcialmente, sob o controle do intérprete; afinal, como acabamos de dizer, sabemos que elas no mínimo são responsáveis por auxiliar na identificação de sons simultâneos. Aparentemente, no entanto, se esse papel musical existe, ele é mínimo: não apenas experimentos mostraram pouco impacto perceptivo em variações artificialmente criadas nessas assincronias (Goebl e Parncutt et al., 2003), mas também intérpretes aparentemente não têm tamanha precisão no controle das assincronias entre notas. Essa falta de precisão vem da influência das sensações táteis e cinestésicas[5] que acompanham a ação.

2.3.3 Percepção da relação temporal entre ação e reação

A relação entre a execução instrumental e as sensações táteis e cinestésicas correspondentes nos traz ao ponto central do problema da latência e da agitação em sistemas multimídia e musicais: a percepção da latência entre uma ação do usuário e a reação correspondente. Nesse contexto, nossa percepção mais uma vez apresenta um grande grau de precisão: experimentos mostraram que variações de $20ms$ no atraso de uma resposta a uma ação, embora não conscientemente percebidas, são corrigidas da mesma maneira que as batidas que acompanham um estímulo que sofre variações (Wing, 1977). É razoável imaginar que mecanismos similares estejam envolvidos nos dois casos; de fato, é provável que se trate

[4]Houve diversas especulações sobre se esse efeito é realmente apenas um reflexo das diferenças de dinâmica entre as vozes ou se ele é introduzido subconscientemente pelos instrumentistas com vistas a sublinhar a linha melódica. Experimentos recentes (Goebl, 2001), no entanto, deixam poucas dúvidas de que o efeito é de fato uma conseqüência da dinâmica; além disso, o efeito perceptivo do adiantamento da melodia parece ser mínimo (Goebl e Parncutt et al., 2003).

[5]Na literatura internacional, as sensações táteis e cinestésicas em conjunto são comumente chamadas de *haptic* (Gillespie, 1999a; Gillespie, 1999b).

do mesmo mecanismo: há uma expectativa sobre o momento para a reação; quando há uma diferença entre o momento esperado e o momento da reação, essa diferença é compensada.

A despeito dessa precisão, em situações em que há ações e reações, há três elementos em jogo que aumentam a complexidade do problema: os comandos motores do usuário, as sensações táteis e cinestésicas correspondentes e a reação externa percebida (*feedback*). Esses elementos são importantes porque há fortes evidências sugerindo que o momento em que percebemos a reação externa pode ser altamente influenciado por diversos fatores, inclusive as sensações táteis e cinestésicas resultantes da ação (que são, elas mesmas, uma forma de *feedback*) (Aschersleben, 2002). Isso significa que eventos que efetivamente acontecem simultaneamente podem ser percebidos como sendo assíncronos, mesmo que apenas em um nível subconsciente.

Como dito anteriormente, sujeitos tipicamente executam um ritmo em conjunto com um metrônomo de maneira regularmente adiantada. O montante da antecipação, no entanto, depende das características dos *feedbacks* auditivo e tátil-cinestésico. Em experimentos com pacientes privados de sensações táteis e cinestésicas, o *feedback* apenas auditivo levou a sincronizações perfeitas; por outro lado, em sujeitos sem esse tipo de distúrbio, o *feedback* apenas tátil-cinestésico, sem *feedback* auditivo, levou a antecipações relativamente grandes; finalmente, o *feedback* tátil-cinestésico normal combinado com o *feedback* auditivo atrasado levou a antecipações crescentes de acordo com o montante desse atraso (Stenneken et al., 2003; Aschersleben e Prinz, 1997; Mates e Aschersleben, 2000). A antecipação, por outro lado, tende a diminuir em contextos em que há outros sons entre cada batida do estímulo (Large, Fink e Kelso, 2002), e há indícios de que ela pode ser influenciada pela intensidade e pela parte do corpo responsável pela ação (mão ou pé, por exemplo).

Essas variações sugerem que o controle de um indivíduo sobre o momento exato em que uma ação deve ser realizada é significativamente influenciado por diversos fatores, o que reforça a hipótese de que, embora assincronias entre notas sejam usadas na discriminação entre notas, seu papel na expressividade musical é provavelmente pouco significativo. A conclusão mais importante que podemos extrair desses fatos, no entanto, é a de que podemos subconscientemente ajustar nossas ações para compensar as diferenças nas condições de atraso do *feedback*. De fato, como nosso sistema motor não é capaz de reagir a comandos instantaneamente, é bastante razoável assumir que precisamos realizar os comandos motores para a execução de qualquer tarefa com precisão temporal *antes* do momento esperado para a ação. Não é difícil de acreditar que diversas formas de percepção dos *feedbacks* resultantes dessa ação sejam usados para calibrar o quanto os comandos devem ser adiantados, o que explicaria os resultados expostos acima. E, realmente, em experimentos onde sujeitos executavam um ritmo isócrono concomitante a um estímulo metronômico, latências de até cerca de $30ms$ entre a ação dos sujeitos e o *feedback* auditivo correspondente foram compensadas, resultando em diferenças nas assincronias de cerca de $10ms$ entre o estímulo e o som provocado pela reação dos sujeitos (Mates e Aschersleben, 2000), que nos parecem ser irrelevantes[6]. As variações medidas na antecipação da ação dos sujeitos foram de cerca de $15ms$ para variações no atraso do *feedback* de pouco menos de $30ms$ no caso de sujeitos que demonstraram ser muito pouco suscetíveis ao fenômeno devido a treinamento anterior. Além disso,

[6]Como discutido anteriormente, assincronias muito maiores são comuns durante a execução musical.

durante experimentos com diferentes atrasos, os sujeitos claramente alteraram seu comportamento de acordo com cada iteração do experimento, forçando os pesquisadores a introduzir iterações de controle entre cada par de iterações para eliminar esse efeito (Aschersleben e Prinz, 1997; Mates e Aschersleben, 2000).

De forma similar, durante a execução de um piano, o tempo entre o pressionar de uma tecla e o início do som da nota é de cerca de $100ms$ para notas *piano* e cerca de $30ms$ para notas *staccato* e *forte* (Askenfelt e Jansson, 1990). Ainda que consideremos que o pianista espera que o ataque ocorra em algum ponto intermediário do percurso da tecla, é muito provável que diferentes níveis dinâmicos sejam responsáveis por diferentes latências do ponto de vista do pianista. Ainda assim, pianistas lidam com esse tipo de dificuldade continuamente, provavelmente de forma subconsciente. Como as vozes em peças para piano geralmente têm curvas dinâmicas que caminham gradualmente, o intérprete tem a oportunidade de se adequar às variações correspondentes na latência. Quando há variações abruptas na dinâmica, elas são geralmente relacionadas a aspectos estruturais da peça, que trazem consigo grandes flutuações temporais interpretativas. Finalmente, a música moderna pode fazer uso de grandes mudanças dinâmicas que não se vinculam a variações temporais interpretativas; no entanto, esse gênero musical geralmente não é baseado em um pulso evidente e constante, tornando os efeitos de variações abruptas na latência muito menos perceptíveis.

É possível que, no caso da execução do piano ou outros instrumentos, não ocorra apenas a adequação gradativa entre os diferentes níveis de latência de acordo com a percepção das variações temporais; talvez pianistas sejam capazes de prever o efeito de variações na dinâmica sobre a latência e aplicar as correções correspondentes sobre a execução. No entanto, embora essa hipótese deva ser investigada, o que importa para nós nessa discussão é que somos capazes de ajustar nosso sistema motor para compensar latências nos efeitos de nossas ações.

2.4 Desenvolvimento e baixa latência

Como vimos, é geralmente interessante garantir que a agitação de um sistema interativo seja mantida relativamente baixa; por outro lado, níveis de latência um pouco mais altos, da ordem de 20–$30ms$ aparentemente são aceitáveis. Em muitas situações, portanto, pode ser interessante utilizar procedimentos que promovam um aumento na latência mas, ao mesmo tempo, reduzam a agitação do sistema. Isso, no entanto, não significa que latências mais baixas não sejam interessantes; muito pelo contrário, já que as latências de diferentes partes de um sistema se somam. Por exemplo, o mero posicionamento de um alto-falante a cerca de 3–$4m$ de distância de um usuário (o que não seria incomum em uma instalação interativa em um ambiente qualquer) adiciona $10ms$ à latência total percebida do sistema.

De fato, diversos algoritmos DSP adicionam latências relativamente altas. Dentre os módulos LADSPA mais comumente usados (parte da coleção de módulos desenvolvida por Steve Harris — [PLUGINS]), alguns apresentam acréscimos desprezíveis à latência, como o módulo DJ EQ, que é responsável por latências da ordem de três amostras; outros, como Bode frequency shifter, GSM simulator, Impulse convolver e Hilbert transformer, são responsáveis por pequenos acréscimos na latência, da ordem

de 2–4ms; e ainda outros têm efeito mais pronunciado, como o módulo Multiband EQ, responsável por latências da ordem de 17ms, e o módulo Pitch scaler, por latências de até 87ms.

Portanto, o acúmulo das latências que têm origem nas diversas partes de um sistema justifica que se procure reduzir a latência de cada uma das partes para evitar latências totais excessivas. Outra razão para procurarmos minimizar a latência de um sistema é que, apesar de que a execução musical envolve comandos motores adiantados no tempo em relação ao momento em que eles devem ser executados, em diversas situações a percepção do *feedback* é usada para corrigir a ação em tempo real; por exemplo, um cantor se utiliza da audição para corrigir a freqüência de uma nota longa enquanto ela soa com um atraso de cerca de 100ms (Burnett et al., 1998; Leydon, Bauer e Larson, 2003). Portanto, quanto menor a latência, mais eficiente será a correção da ação. Ainda assim, esperamos ter demonstrado que soluções intermediárias para a latência são plenamente aceitáveis.

Capítulo 3

Processamento baseado em funções *callback*

O processamento de E/S por um computador envolve uma interação complexa entre o dispositivo de E/S, o sistema operacional e a aplicação. A necessidade de garantias de baixa latência aumenta consideravelmente essa complexidade. As características dessa interação acabaram por conduzir ao desenvolvimento de soluções para o tratamento de E/S de áudio (tais como ASIO, CoreAudio e JACK) que, embora diferentes, baseiam-se em um mecanismo similar: o uso de funções *callback*.

Neste capítulo, apresentamos o princípio de funcionamento de mecanismos de E/S tradicionais e abordamos suas deficiências para aplicações de tempo real. A seguir, discutimos os mecanismos baseados em funções *callback* e descrevemos os sistemas JACK e LADSPA, ambos baseados nesse paradigma. É importante observar que, embora tenhamos dado ênfase a sistemas para o processamento de áudio, os mecanismos aqui descritos podem ser aplicados também a outras mídias.

3.1 Interação entre o computador e o hardware de E/S[1]

Nos computadores de uso geral, o processamento de E/S quase sempre segue um mesmo tipo de funcionamento, baseado em interrupções de hardware. Os sistemas operacionais de uso geral, por sua vez, utilizam *buffers* de dados internos para a comunicação com o dispositivo de E/S ao processar essas interrupções de hardware. O uso desses *buffers*, embora excelente em termos de utilização dos recursos da máquina, não é adequado para aplicações em que baixa latência é desejável; outros mecanismos precisam ser implementados nesses casos.

3.1.1 Sistemas de E/S baseados em interrupções

A maior parte dos sistemas de E/S em microcomputadores é baseada no tratamento de interrupções de hardware. Em sistemas que usam esse mecanismo, o dispositivo de E/S gera uma requisição de

[1] Os mecanismos de tratamento de interrupções são abordados em textos introdutórios sobre arquiteturas de computadores ou sistemas operacionais; por exemplo, Rhyde (1996, Cap. 17); Silberschatz e Galvin (2000, Cap. 12)

interrupção quando precisa transferir dados para o computador ou está pronto para receber dados do computador. Ao receber essa requisição de interrupção, o processador abandona o processamento corrente, realiza as operações necessárias para a transferência de dados entre o computador e o dispositivo de E/S e retoma o processamento anterior; esse modo de operação permite que o processador esteja livre para o processamento mesmo que haja operações de E/S em andamento.

Embora o procedimento básico seja sempre igual, os diversos dispositivos de E/S de um computador precisam de diferentes rotinas para que a transferência de dados ocorra corretamente: o endereço de acesso a cada dispositivo é diferente, o tamanho do bloco de dados a ser transferido a cada interrupção pode variar etc. Dado que um computador pode ter um sem-número de diferentes dispositivos de E/S, fica evidente que as rotinas que respondem às requisições de interrupções dos dispositivos de E/S devem ser programáveis, e não implementadas diretamente no hardware ou firmware do computador[2]. E, de fato, essas rotinas são normalmente implementadas como parte do sistema operacional.

Para possibilitar que essas rotinas sejam programáveis, a maioria dos computadores armazena uma tabela (chamada *vetor de interrupções*) em que há uma correspondência entre uma linha de interrupção (gerida por um dispositivo de E/S) e o endereço na memória da rotina responsável por tratar as interrupções daquele dispositivo. O sistema operacional, durante a sua inicialização, preenche essa tabela com os endereços de suas rotinas de tratamento de interrupção. Quando o dispositivo gera uma requisição de interrupção, o hardware do computador executa a rotina correspondente registrada pelo sistema operacional. Essa rotina é uma função *callback*: após ser registrada, ela é executada se e somente se ocorre um evento relevante (uma interrupção), ou seja, nenhuma outra parte do sistema operacional executa chamadas a ela, e sua única razão de ser é tratar as interrupções. Essa rotina é executada sincronamente às interrupções geradas pelo dispositivo de E/S, ou seja, há unidade temporal entre a geração de uma interrupção e a execução de sua rotina de tratamento.

3.1.2 Escalonamento de processos e E/S

Embora em alguns sistemas embutidos de tempo real seja razoável realizar todo o processamento dos dados necessário dentro da própria rotina de tratamento de interrupção, essa não é a solução comumente usada. As operações de E/S muitas vezes têm restrições de tempo real; portanto, as rotinas de tratamento de interrupção devem ser executadas rapidamente para não interferirem com outras rotinas de tratamento de interrupções. Além disso, é de se esperar que diferentes processamentos possam ser dinamicamente associados a cada dispositivo de E/S, o que não seria possível caso o processamento fosse totalmente executado dentro da rotina de tratamento de E/S. Ao invés disso, o que se faz é executar um programa responsável por processar os dados que se comunica com o sistema operacional; este, por sua vez, realiza a comunicação com o dispositivo de E/S.

Como vimos, o mecanismo de interrupções permite que o processador possa trabalhar enquanto há operações de E/S em andamento. Para que isso seja possível, no entanto, é preciso que haja tarefas a ser executadas pelo processador enquanto a operação de E/S não é completada. Em sistemas operacionais

[2]Em sistemas embutidos, o uso de firmware para essas rotinas é possível porque os dispositivos de E/S do sistema são pré-determinados, diferentemente do que acontece em sistemas de uso geral.

multitarefa de uso geral, como o Linux, Windows, MacOS e outros, isso é possível porque o sistema operacional gerencia o uso do processador por diferentes processos (por exemplo, o processo que gerencia a disposição das janelas numa interface gráfica, o processo responsável por ler dados do disco rígido, o processo responsável por enviar dados para serem reproduzidos pela placa de som etc.), alternando periodicamente qual processo está sendo executado pelo processador. Quando um processo depende de uma operação de E/S para continuar seu processamento, o sistema operacional passa a executar outro processo entre aqueles que não estão esperando por nenhuma operação de E/S. O sistema também pode suspender a execução de um processo e executar um outro após um limite de tempo para garantir que todos os processos sejam executados alternadamente ao longo do tempo.

Num sistema desse tipo, pode haver diversos processos impedidos de continuar sua execução por estarem esperando por operações de E/S e diversos processos prontos para continuar a ser executados, esperando apenas a sua vez. É evidente, portanto, que o término de uma operação de E/S normalmente *não* faz com que a execução do processo correspondente seja retomada imediatamente; novas interrupções, correspondentes a novas operações de E/S, podem ocorrer antes de o processo voltar a ser executado.

Para que não haja perda de dados por causa disso, o sistema operacional tradicionalmente mantém *buffers* de dados vinculados aos dispositivos de E/S; as rotinas de tratamento de interrupções do sistema operacional apenas transferem os dados entre esses *buffers* e o dispositivo correspondente. Quando há espaço disponível nos *buffers* de saída ou dados disponíveis nos *buffers* de entrada, o sistema operacional coloca o processo correspondente na lista dos processos que estão prontos para ser executados e, em algum momento, retoma sua execução. O processo continuamente processa os dados e se comunica com o dispositivo de E/S usando chamadas de sistema como `read()` e `write()`; quando os *buffers* do sistema estão cheios ou vazios, o sistema operacional suspende o processo até que os *buffers* sejam ao menos parcialmente processados pela rotina de tratamento de interrupções correspondente. O processo, portanto, não tem nenhum controle e nem é informado sobre quando é suspenso e quando é retomado; o processamento realizado pelo processo é assíncrono em relação às interrupções geradas pelo dispositivo de E/S, ou seja, não há unidade temporal entre as interrupções e o processamento. Esse modelo é bastante adequado para diversas aplicações, e possibilita um excelente aproveitamento dos recursos do sistema; no entanto, ele acrescenta uma grande latência ao processamento do sinal, o que não é aceitável em diversas aplicações interativas.

3.2 Latência em dispositivos de áudio

As placas de som (tais como muitos outros dispositivos de E/S) normalmente possuem *buffers* de dados internos, usados tanto para o armazenamento dos dados referentes ao áudio capturado quanto para o armazenamento dos dados em espera para serem reproduzidos. Esses *buffers* são periodicamente esvaziados ou preenchidos, conforme o caso, através da comunicação, via barramento de E/S, com o computador; essa comunicação é regida pela geração de requisições de interrupção por parte da placa de som quando um de seus *buffers* está cheio ou vazio. A placa normalmente precisa alocar ao menos

dois *buffers* para cada sentido da comunicação: no caso da saída de dados, por exemplo, enquanto os dados contidos em um *buffer* são convertidos para o domínio analógico e reproduzidos, o outro *buffer* é preenchido com dados pelo computador; um mecanismo semelhante ocorre na comunicação no sentido inverso.

Assim, supondo que todos os *buffers* de entrada e saída têm tamanho igual e que, no caso de operação *full-duplex*, os *buffers* de entrada estão alinhados no tempo com os de saída (e supondo, portanto, que uma única interrupção corresponde a uma solicitação para que o processador preencha um *buffer* de saída e esvazie um *buffer* de entrada), a menor latência total possível de um computador que simplesmente copia os dados da entrada para a saída (ou seja, o menor tempo possível para que um sinal injetado na entrada seja reproduzido na saída) corresponde exatamente à soma das durações correspondentes ao tamanho de um *buffer* de entrada e um de saída. Como o tamanho dos *buffers* usados pela placa de som normalmente é configurável via software (dentro de alguns limites), é possível definir com clareza a latência de um sistema de áudio computadorizado. Para que isso seja verdade, no entanto, o computador não pode introduzir nenhum outro tipo de atraso ao sinal: ele necessariamente deve processar tanto os dados fornecidos pela placa de som quanto os dados que devem ser fornecidos para a placa de som no momento em que eles são fornecidos ou solicitados, ou seja, no momento da geração da requisição de interrupção por parte da placa. Isso significa que o processamento tem que ser realizado sincronamente às interrupções.

Como vimos, no entanto, essa não é uma tarefa simples em um sistema operacional de uso geral: o tratamento das interrupções geradas pelo hardware é papel do núcleo do sistema operacional, e é ele quem realiza a transferência de dados entre a placa de som e a aplicação, utilizando *buffers* adicionais para compensar o atraso não-determinístico do agendamento de tarefas.

3.3 *Callbacks*

Com uma abordagem diferente, os sistemas ASIO ([ASIO]), CoreAudio ([COREAUDIO]) e JACK ([JACK]) procuram justamente possibilitar que as aplicações possam processar os dados referentes à comunicação com o hardware de áudio no momento em que o hardware solicita uma interrupção, ou seja, procuram dar garantias de tempo real rígido a sistemas operacionais de uso geral. Esses sistemas procuram simplesmente imitar o mecanismo das funções *callback* do núcleo responsáveis pelo tratamento de interrupções, transpondo esse mecanismo para o nível das aplicações de usuário e permitindo que o tratamento das interrupções seja feito por aplicativos complexos quaisquer, independentemente de alteração no código do núcleo. A cada interrupção, uma função *callback* da aplicação (ou aplicações, como veremos mais adiante) é executada, tendo acesso aos *buffers* de dados da placa de som correspondentes à interrupção corrente, o que garante a menor latência possível para o sistema.

Esse tipo de operação corresponde a um modelo de processamento síncrono: os dois eventos (a requisição de interrupção e a execução da função *callback*) estão sincronizados entre si, ou seja, os dois sempre, necessariamente, ocorrem simultaneamente (ou quase). No caso aqui descrito, a sincronização é garantida porque há uma relação de causalidade entre a requisição de interrupção e a execução da

função *callback*; no entanto, a sincronização poderia ser baseada em uma relação de causalidade entre um outro evento e os dois eventos em questão, por exemplo.

É importante observar que, para o funcionamento correto desse mecanismo, a função *callback* não pode executar nenhuma chamada ao sistema que possa bloquear o processo. Chamadas que bloqueiam um processo (como requisições de acesso a disco, alocação de memória etc.) podem demorar tempos imprevisíveis para serem executadas. Além disso, tornam possível o agendamento de outra tarefa durante a função *callback*, aumentando ainda mais essa imprevisibilidade. Portanto, chamadas que bloqueiam o processo podem fazer a função *callback* não ser capaz de terminar seu processamento antes da próxima interrupção do dispositivo de E/S.

Fica claro, portanto, que aplicações compatíveis com esses sistemas precisam ser organizadas com base em funções *callback* e precisam poder operar de maneira sincronizada com as requisições de interrupção do hardware de E/S. Mas, como vimos, aplicações para o processamento de áudio desse tipo se beneficiam do desenvolvimento modular; isso significa que o funcionamento desses módulos também deve ser compatível com um sistema baseado em *callbacks*. As especificações de desenvolvimento de módulos de processamento DSP mais comumente usadas (LADSPA — [LADSPA] e VST — [VST]) solucionam esse problema implementando o mecanismo de comunicação entre as aplicações e os módulos também sob a forma de *callbacks*. Após carregar a biblioteca que implementa um determinado módulo e inicializá-lo, a aplicação registra *buffers* de entrada e saída de dados junto ao módulo. A partir desse momento, executa chamadas periódicas a uma função desse módulo; a cada chamada, o módulo processa os dados dos *buffers* de entrada e coloca os resultados do processamento nos *buffers* de saída. No caso de operação em tempo real, esses *buffers* podem ser os próprios *buffers* internos da placa de som, aos quais a aplicação tem acesso através do sistemas JACK, ASIO ou CoreAudio.

3.3.1 JACK

O mecanismo para o gerenciamento de E/S e processamento de áudio baseado em funções *callback* mais difundido é o ASIO (***A****udio* ***S****tream* ***I****nput/* ***O****utput* — [ASIO]), que é, na prática, o padrão da indústria de sistemas para áudio profissional e é compatível com os sistemas Windows e MacOS. Outro sistema importante é o CoreAudio ([COREAUDIO]), que é o mecanismo padrão para a comunicação com o hardware de áudio em sistemas baseados no MacOS X. Em ambiente Linux, o sistema mais difundido e sofisticado para o processamento de áudio usando funções *callback* é o sistema JACK (***J****ack* ***A****udio* ***C****onnection* ***K****it* — [JACK]). O JACK é responsável por intermediar a comunicação entre a aplicação e o dispositivo de E/S de áudio; para isso, ele normalmente utiliza o sistema ALSA.

O ALSA (***A****dvanced* ***L****inux* ***S****ound* ***A****rchitecture* — [ALSA]) consiste em um conjunto de controladores de dispositivos de baixo nível (operando como parte do núcleo do sistema operacional) e uma biblioteca para a comunicação com esses controladores, oferecendo um excelente nível de abstração do dispositivo de E/S de áudio. O JACK, por sua vez, oferece ao programador uma interface de mais alto nível que o padrão ALSA para a comunicação com a placa de som com baixa latência, utilizando uma representação única para o áudio independente das características e limitações do dispositivo. Ao

mesmo tempo, o JACK estabelece um mecanismo transparente para a comunicação entre as aplicações de áudio. O programador precisa apenas desenvolver a função *callback* que deve ser executada periodicamente para o processamento do áudio; o JACK é responsável por interligar as diversas aplicações entre si e com a placa de som.

O sistema JACK se baseia em 3 características importantes dos sistemas GNU/Linux modernos:

Escalonamento de tempo real: O crescimento de aplicações de tempo real flexível e firme trouxe à tona a necessidade de expandir os sistemas semelhantes ao UNIX para acomodá-las, o que impulsionou a definição do padrão POSIX para sistemas de tempo real (Gallmeister, 1995), implementado no Linux. De acordo com esse padrão, uma aplicação (desde que tenha privilégios para isso) pode solicitar ao sistema operacional que seja enquadrada na categoria de agendamento SCHED_FIFO: ao invés de serem agendadas para execução como as outras aplicações, aplicações na categoria SCHED_FIFO são agendadas para execução assim que estão prontas, e só são suspensas se forem bloqueadas (estiverem aguardando uma operação de E/S) ou se uma outra aplicação SCHED_FIFO de maior prioridade se tornar pronta para execução.

***Patches* para baixa latência:** Embora aplicações SCHED_FIFO devam ser agendadas para execução assim que estão prontas, o sistema operacional demora algum tempo antes de poder executar a aplicação agendada; há um período de latência entre o momento em que a aplicação está pronta e o momento em que ela efetivamente é executada. Essa latência no Linux até a versão 2.4.x pode chegar a mais de $500ms$ no pior caso. Para que o expediente baseado em *callbacks* funcione adequadamente para *buffers* de tamanho pequeno, é necessário que, após a detecção de uma interrupção, o núcleo do sistema suspenda a execução de quaisquer outros processos da máquina praticamente sem nenhum atraso e agende o programa de nível de usuário responsável pelo tratamento da interrupção. Diversos *patches* de baixa latência para o núcleo do linux, desenvolvidos por Ingo Molnar, Andrew Morton e Robert Love, são capazes de trazer o tempo de resposta do sistema para valores entre um e dois milissegundos[3]. A versão mais recente do Linux, 2.6, reduz significativamente as latências típicas de agendamento devido a diversas melhorias e à incorporação do patch que permite mudanças de contexto dentro do núcleo do Linux, mas não atinge latências tão baixas quanto os melhores *patches* citados acima.

Sistema de som ALSA: O sistema JACK foi desenvolvido de maneira modular, o que pode permitir que diferentes mecanismos de comunicação com a placa de som sejam utilizados por ele; no entanto, o único mecanismo efetivamente implementado é o que utiliza o padrão ALSA. O padrão ALSA oferece ao programador a possibilidade de solicitar ao sistema que o programa seja escalonado para execução de acordo com as interrupções do hardware de áudio. A aplicação, assim, acessa diretamente os *buffers* de dados da placa de som ao invés de utilizar *buffers* de dados

[3]Graças ao programa latencytest ([LATENCYTEST]), desenvolvido por Benno Senoner, é bastante simples realizar testes de desempenho para verificar o tempo de resposta do Linux. Por exemplo, os resultados do *patch* para preempção do núcleo aplicado ao Linux versão 2.4.6 estão disponíveis em <http://kpreempt.sourceforge.net/benno/linux+kp-2.4.6/3x256.html>.

intermediários do sistema operacional, o que permite o desenvolvimento de aplicações baseadas em funções *callback* com latência dependente apenas do tamanho desses *buffers*.

Graças às características acima, é possível desenvolver aplicações de áudio para o sistema Linux capazes de oferecer excelentes características de baixa latência de processamento: uma aplicação na categoria SCHED_FIFO sendo executada em um sistema que utilize os *patches* de baixa latência pode configurar a placa de som para utilizar *buffers* de dados pequenos, garantindo latências totais da ordem de 3 a $6ms$ (MacMillan, Droettboom e Fujinaga, 2001)[4].

O JACK é baseado em um servidor, jackd, que é executado com prioridade SCHED_FIFO e que interage com o controlador da placa de áudio através do sistema ALSA. Ao ser agendado para execução, ele passa a agir como um escalonador de tarefas secundário, no nível das aplicações de usuário, gerenciando a execução de outros aplicativos SCHED_FIFO (os clientes do servidor jackd) que, assim como ele, permanecem bloqueados entre uma interrupção e outra do hardware de áudio. Os clientes então realizam o processamento necessário e enviam os resultados para o servidor jackd, que preenche os *buffers* da placa de som.

Os clientes do servidor jackd são, na verdade, *threads* de programas de áudio quaisquer que têm suporte ao sistema JACK por terem sido ligados dinamicamente à biblioteca libjack. Essas *threads* são criadas pela libjack e executam o processamento referente a cada aplicativo conectado ao servidor através da leitura e escrita de dados em segmentos de memória compartilhada criados pelo servidor jackd, o que dá ao servidor jackd a possibilidade de realizar a conexão desses aplicativos entre si além de com a placa de som: basta fazer dois aplicativos compartilharem um mesmo segmento de memória para que um escreva os dados que serão lidos pelo outro.

Para que isso funcione adequadamente, os programas clientes precisam definir exatamente qual código deve ser executado pela *thread* criada pela libjack. Para isso, o programa define uma função (que não pode realizar chamadas que bloqueiem, já que será executada com necessidades estritas de tempo) e a registra junto à libjack como uma função *callback*, ou seja, uma função que deverá ser executada quando do agendamento para execução pelo servidor jackd. Muito provavelmente, essa função precisa ter acesso a dados compartilhados entre as *threads*; é muito importante que essa comunicação seja feita de forma independente de sistemas de bloqueios (*locks*) ou semáforos para que não haja atrasos na execução das chamadas *callback*.

Assim, o jackd funciona como um escalonador de tarefas que faz com que as sucessivas chamadas *callback* definidas pelos aplicativos sejam executadas em ordem. Para que isso seja possível, quando os clientes se registram junto ao jackd, o servidor estabelece uma ordenação topológica entre as chamadas *callback* de forma que qualquer aplicativo cuja saída seja processada por um outro seja executado antes deste. Para realizar o agendamento, o jackd simplesmente força o escalonador do sistema operacional a agendar o processo adequado para execução: todas as *threads* criadas pela libjack nos clientes estão bloqueadas aguardando a leitura de um byte a partir de um *named pipe* criado pelo servidor jackd;

[4]MacMillan, Droettboom e Fujinaga (2001) mencionam o sistema JACK sob o nome LAAGA, já que em seus estágios preliminares esse era o nome do projeto.

depois que o jackd é ativado para execução pelo núcleo, ele coloca os dados lidos da placa de som no segmento de memória de entrada do primeiro cliente a ser agendado, escreve um byte no *named pipe* sobre o qual essa aplicação está bloqueada em espera e bloqueia na leitura de um outro *named pipe*. Como a aplicação tem prioridade SCHED_FIFO e o servidor jackd bloqueia nesse momento, a aplicação é agendada para execução, processa os seus dados de entrada e os escreve na saída; a seguir, a aplicação escreve um byte em um outro *named pipe* e bloqueia novamente na leitura do *named pipe* anterior. É sobre esse segundo *named pipe* que a próxima aplicação a ser executada está bloqueada, o que faz com que essa aplicação desbloqueie e inicie sua execução. Esse processamento continua até que a última aplicação escreva um byte no *named pipe* sobre o qual o servidor jackd está bloqueado. Nesse momento, o servidor é desbloqueado, envia os dados para a placa de som e bloqueia novamente, agora aguardando pela próxima interrupção da placa de som.

Vale observar que, embora a implementação atual seja baseada no sistema ALSA, é possível implementar o JACK com base em outros sistemas; de fato, existe uma versão em desenvolvimento para operação sobre o sistema CoreAudio. Essa característica pode, inclusive, permitir que o JACK seja controlado por um dispositivo que não seja de áudio. Portanto, deve ser possível desenvolver um sistema semelhante ao que foi desenvolvido por nós com base no sistema JACK ao invés do padrão LADSPA.

3.3.2 LADSPA

O padrão de mercado para módulos de DSP para o processamento de áudio em aplicações musicais é o VST (***V****irtual* ***S****tudio* ***T****echnology* — [VST]). De maneira similar ao ASIO, ele é compatível com os sistemas Windows e MacOS, e existe uma grande quantidade de aplicações e módulos compatíveis com ele. Seria interessante fazer da especificação VST o mecanismo padrão para o uso de módulos DSP também em Linux: isso permitiria a utilização da enorme quantidade de módulos VST já existentes. No entanto, incompatibilidades de licença e problemas com a maneira com que alguns módulos utilizam interfaces gráficas com o usuário impossibilitaram esses planos[5]. Assim, uma nova especificação foi desenvolvida, a LADSPA (***L****inux* ***A****udio* ***D****eveloper's* ***S****imple* ***P****lugin* ***A****PI* — [LADSPA], Figura 3.1). Aplicações desenvolvidas de acordo com a especificação são capazes de carregar e utilizar quaisquer módulos LADSPA sem nenhum código adicional.

A especificação define que um conjunto de módulos relacionados são compilados em uma biblioteca dinâmica que pode ser carregada pela aplicação. Essa biblioteca expõe para a aplicação uma função (ladspa_descriptor ()) que devolve para a aplicação estruturas do tipo LADSPA_Descriptor, uma para cada módulo disponível na biblioteca. Um LADSPA_Descriptor, por sua vez, possui diversas informações sobre um módulo (nome, número e nome das portas de entrada e saída etc.) e apontadores para funções (presentes na biblioteca) que implementam a funcionalidade desse módulo.

Uma dessas funções é a função que cria uma instância do módulo, com suas estruturas de dados internas, e devolve uma referência opaca (*handle*) para essa instância; essa instância deve ser sempre fornecida como parâmetro para as chamadas de função do módulo. Após criar uma instância do

[5]Ainda assim, é possível utilizar alguns módulos VST em Linux com o programa VSTServer — [VSTSERVER].

```
typedef float LADSPA_Data;
typedef int LADSPA_Properties;
typedef int LADSPA_PortDescriptor;
typedef void * LADSPA_Handle;
typedef struct _LADSPA_Descriptor {
  unsigned long UniqueID;
  const char * Label;
  LADSPA_Properties Properties;
  const char * Name;
  const char * Maker;
  const char * Copyright;
  unsigned long PortCount;
  const LADSPA_PortDescriptor * PortDescriptors;
  const char * const * PortNames;
  void * ImplementationData;
  LADSPA_Handle (*instantiate)
        (const struct _LADSPA_Descriptor * Descriptor,
         unsigned long                     SampleRate);
   void (*connect_port)(LADSPA_Handle Instance,
                        unsigned long Port,
                        LADSPA_Data * DataLocation);
  void (*activate)(LADSPA_Handle Instance);
  void (*run)(LADSPA_Handle Instance,
              unsigned long SampleCount);
  void (*run_adding)(LADSPA_Handle Instance,
                     unsigned long SampleCount);
  void (*set_run_adding_gain)(LADSPA_Handle Instance,
                              LADSPA_Data   Gain);
  void (*deactivate)(LADSPA_Handle Instance);
  void (*cleanup)(LADSPA_Handle Instance);
} LADSPA_Descriptor;
const LADSPA_Descriptor * ladspa_descriptor(unsigned long Index);
typedef const LADSPA_Descriptor *
(*LADSPA_Descriptor_Function)(unsigned long Index);
```

Figura 3.1: A parte central da interface definida pela especificação LADSPA

módulo, a aplicação prepara a comunicação entre si e o módulo definindo, para cada uma das portas do módulo, um *buffer* de dados que será lido ou escrito pelo módulo a cada iteração. Quando todos os *buffers* foram definidos e registrados, a aplicação executa chamadas periódicas à função `run()`, que faz o módulo processar os dados presentes em suas portas de entrada e colocar os resultados em suas portas de saída; essa função é a função *callback* do módulo.

As portas de um módulo LADSPA podem ser entradas ou saídas, e podem ser portas de áudio ou de controle. Portas de áudio correspondem a *buffers* de tamanhos variáveis de acordo com o volume de dados a ser processados a cada iteração; portas de controle correspondem a um único valor, que normalmente é um parâmetro para o algoritmo implementado pelo módulo. Portas de controle que são saídas podem ser, por exemplo, medidores de nível de sinal.

Essa arquitetura permite que aplicações possam interagir com módulos sem possuir nenhum código específico para cada módulo; a aplicação pode, por exemplo, apresentar todas as portas de controle do módulo que são entradas para o usuário, que deve conhecer o módulo para poder definir os seus parâmetros. Já as portas de áudio podem ser conectadas automaticamente à entrada e à saída de áudio da aplicação ou a outros módulos, de acordo com a necessidade do usuário.

Capítulo 4

Sistemas distribuídos de tempo real

O sistema aqui descrito envolve o processamento distribuído de áudio em tempo real com vistas ao aumento na capacidade de processamento disponível; esse uso da distribuição de carga no sistema o aproxima de um sistema para processamento paralelo. Assim, antes de discutirmos os mecanismos de funcionamento do sistema, é preciso discutir as características relevantes de sistemas de tempo real, paralelos, distribuídos e de sistemas paralelos de tempo real.

Para simplificar essa discussão, começaremos apresentando um modelo simples de processamento computacional. A seguir, apresentaremos as semelhanças e diferenças entre sistemas paralelos e distribuídos e discutiremos diversas abordagens comumente usadas para o processamento paralelo. Em seguida, exporemos alguns conceitos básicos sobre sistemas de tempo real com vistas a determinar os diversos aspectos que definem suas características principais. Finalmente, levantaremos as características desejáveis em um sistema para o processamento paralelo voltado para aplicações de tempo real.

4.1 Um modelo para o processamento computacional[1]

Todo processamento em um sistema computacional é composto por operações primárias implementadas no nível do hardware, como leitura ou escrita de dados em posições de memória, comparações etc. Linguagens de programação, bibliotecas, chamadas de função e outros mecanismos permitem que o usuário do sistema encare um conjunto de operações primárias desse tipo como uma única operação; por exemplo, a soma de dois números pode ser encarada como uma única operação quando, na verdade, envolve vários acessos à memória e algumas instruções do processador.

Assim como um conjunto de operações pode ser encarado como uma única tarefa abstrata, também é possível tomar uma tarefa abstrata e dividi-la em subtarefas menores. Ou seja, dependendo do nível de abstração desejado, uma tarefa a ser realizada por um sistema computacional pode ser dividida em um número maior ou menor de subtarefas. Por exemplo, um conjunto de operações similares aplicadas sobre os elementos de um conjunto de dados pode ser encarado como várias operações independentes

[1] Existem diversas maneiras de modelar um sistema computacional; a que usamos aqui é baseada em uma das formas descritas em Norman e Thanisch (1993) e em Foster (1995, Seção 1.3).

ou como uma única operação sobre todos os elementos do conjunto; nesse segundo caso, essa operação é normalmente abstraída sob a forma de um laço ou recursão.

Por outro lado, todo processamento útil realizado por um sistema computacional, desde cálculos matemáticos até a atualização dos elementos em uma interface gráfica com o usuário, envolve a leitura de dados de entrada e a produção de dados de saída correspondentes. De fato, se não houver a produção de dados de saída, o processamento não é útil; e, já que computadores são sistemas determinísticos, se não houver dados de entrada não é possível haver variação nos dados de saída, tornando o processamento redundante.

Assim, podemos dizer que todo processamento completo realizado por um sistema computacional recebe dados de entrada, produz dados de saída e pode ser dividido em subprocessamentos menores que também recebem dados de entrada e produzem dados de saída. Ou seja, podemos caracterizar um sistema computacional destinado a realizar uma tarefa completa T como um sistema onde diversas subtarefas $t_1, t_2, t_3 \ldots t_n$ são executadas sobre um conjunto de dados de entrada para produzir um conjunto de dados como resultado. Cada uma dessas subtarefas recebe dados de entrada e produz dados de saída; muito provavelmente, algumas delas utilizam dados produzidos por outras como dados de entrada. Assim, é possível representar uma tarefa a ser realizada por um sistema computacional por um grafo direcionado acíclico (DAG) onde os vértices são subtarefas a serem executadas e as arestas são o fluxo de dados entre uma tarefa que os produz e outra que os utiliza. Podemos ainda acrescentar a este grafo as tarefas t_0 e t_f tais que t_0 corresponde ao mecanismo pelo qual os dados de entrada são fornecidos ao sistema e t_f ao mecanismo pelo qual os dados de saída são fornecidos pelo sistema; assim, existe pelo menos uma aresta que parte de t_0 e atinge alguma das tarefas t_1–t_n, pelo menos uma aresta que parte de alguma das tarefas t_1–t_n e atinge t_f, não existe nenhuma aresta que atinge t_0 e não existe nenhuma aresta que parte de t_f. Finalmente, podemos considerar que algumas das tarefas t_1–t_n são opcionais, ou seja, podem ou não ser executadas em função de testes condicionais realizados durante a execução da tarefa (Figura 4.1). Esse modelo será útil na discussão sobre sistemas paralelos e distribuídos a seguir.

Dados a, b, c, d (t_0):

```
a←a+1; (t1)
a←2*a; (t2)
b←b/2; (t3)
k←a+b; (t4)
l←a+c+k; (t5)
for i←1 to 5
    do print l*i; (t6–9)
print d*5; (t10)
```

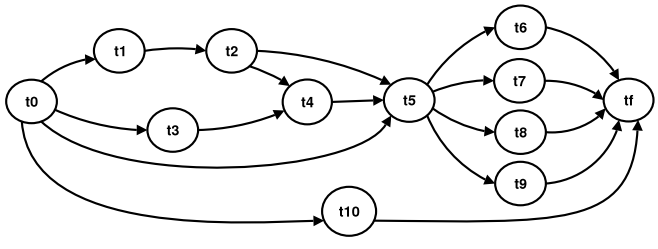


Figura 4.1: As tarefas a ser executadas pelo pseudocódigo à esquerda podem ser representadas pelo grafo à direita.

4.2 Processamento paralelo e distribuído

O número continuamente crescente de computadores em uso trouxe à tona a necessidade de comunicação entre esses computadores, popularizando cada vez mais as redes de computadores para uso geral. A complexidade em lidar com um grande número de computadores interligados simultaneamente, por sua vez, deu origem ao desenvolvimento de diversos mecanismos para promover a integração o mais transparente possível entre os computadores de uma rede. Sistemas de computadores interligados dessa maneira são comumente chamados sistemas distribuídos (Tanenbaum e Steen, 2002, p. 2), e são particularmente úteis em ambientes onde o uso de um único sistema computacional seria dificultado por limitações impostas pelo espaço físico.

Por outro lado, embora a capacidade de processamento dos sistemas computacionais venha crescendo exponencialmente, existem diversos problemas que exigem muito mais capacidade de processamento que um único processador é capaz de oferecer (como o processamento de linguagem natural, alguns tipos de reconhecimento de padrões, cálculos astronômicos, simulações atmosféricas para a previsão do tempo etc.). Como resultado, diversos mecanismos foram e têm sido desenvolvidos com vistas a combinar a capacidade de processamento de vários processadores para solucionar esses problemas. Sistemas onde diversos processadores são usados para solucionar uma tarefa mais rapidamente são chamados sistemas para computação paralela (Kumar et al., 1994, p. 3).

Embora sistemas para computação paralela e distribuída tenham sido tradicionalmente desenvolvidos em ambientes e com objetivos diferentes, o aumento na velocidade das redes de computadores de uso geral tem tornado possível o uso de sistemas distribuídos para o processamento paralelo de diversos problemas (Kung et al., 1991). Por outro lado, técnicas para o processamento paralelo podem muitas vezes oferecer benefícios para sistemas distribuídos. Neste trabalho, implementamos um sistema distribuído voltado para o aumento de desempenho no processamento de áudio em tempo real; assim, nosso interesse é no uso de sistemas distribuídos para o processamento paralelo.

4.2.1 Sistemas distribuídos

Sistemas distribuídos são baseados em um conjunto de sistemas computacionais interligados por uma rede de computadores. Um dos cenários mais intertessantes para o uso de sistemas distribuídos são aplicações que dependem, de alguma maneira, de separação espacial entre partes do processamento; essas aplicações podem envolver comunicação de dados, compartilhamento de recursos etc. Por exemplo, um servidor de arquivos central e clientes desse servidor dotados de capacidade de processamento local formam um sistema distribuído que permite o uso simultâneo por parte de um grande número de usuários, garante um bom nível de desempenho (já que cada máquina cliente está inteiramente disponível para o usuário) e, ao mesmo tempo, oferece ao usuário a possibilidade de usar qualquer máquina cliente indiscriminadamente e ao administrador da rede a facilidade de realizar cópias de segurança regulares em apenas uma máquina da rede. Outro exemplo é um sistema em que, numa fábrica robotizada, cada robô é controlado localmente por um computador que, por sua vez, se comunica com os outros computadores para coordenar o andamento do processo de fabricação.

Diversos outros cenários podem se beneficiar de sistemas distribuídos, inclusive com vistas a melhorias de desempenho. De fato, com o aumento de velocidade das redes locais de computadores, tornou-se possível o uso de sistemas interconectados por esse tipo de rede em aplicações que envolvem processamento paralelo (Kung et al., 1991), aproximando ainda mais sistemas paralelos e distribuídos. Um sistema distribuído pode ser visto como um sistema MIMD (*Multiple Instruction, Multiple Data*) fracamente acoplado (Chaudhuri, 1992, p. 19–36; Tanenbaum e Steen, 2002, p. 16–21), e pode ser utilizado para o processamento de tarefas que possam ser solucionadas efetivamente por arquiteturas desse tipo.

Uma das vantagens do uso de sistemas distribuídos para a computação paralela é que, de maneira geral, esses sistemas utilizam tanto computadores quanto redes de uso geral, evitando os custos relativamente elevados de sistemas para computação paralela baseados em hardware específico. Por outro lado, o atraso na comunicação de dados entre as diversas máquinas de um sistema distribuído, provocado pela velocidade ainda relativamente baixa do sistema de rede, pode ser excessivo dependendo da aplicação.

Middleware

O desenvolvimento de aplicações em sistemas distribuídos é relativamente complexo: além de ser necessário implementar os mecanismos para a solução do problema a que o sistema se destina, é preciso lidar com os diversos aspectos envolvidos na comunicação entre as máquinas do sistema. Exemplos desses aspectos são a sincronização entre processos, os mecanismos para o gerenciamento de um número variável de máquinas no sistema, as técnicas para garantir a segurança e a tolerância a falhas etc. Finalmente, é comum que as máquinas em um sistema distribuído sejam heterogêneas, o que aumenta ainda mais a complexidade para a implementação de um sistema capaz de ser executado em diversas máquinas simultaneamente.

Para simplificar o desenvolvimento de aplicações distribuídas, tem se tornado cada vez mais comum o uso de uma camada de abstração intermediária entre a aplicação e as diversas partes dos sistemas distribuídos. Essa camada, normalmente chamada middleware (Tanenbaum e Steen, 2002, p. 36), é responsável por gerenciar os diversos aspectos do sistema não diretamente ligados ao funcionamento da aplicação, como mecanismos para a troca de mensagens de alto nível entre processos, sistemas para a localização automática de processos ou dados no sistema e para o acesso a esses objetos como se eles estivessem disponíveis na máquina local etc. Um exemplo de middleware bastante difundido é a especificação CORBA (Tanenbaum e Steen, 2002, p. 494–525; Henning e Vinoski, 2001; Siegel, 2000), desenvolvida por um consórcio de empresas, universidades e institutos de pesquisa, que permite que diferentes fornecedores implementem middlewares de acordo com a especificação que são compatíveis entre si e se apresentam para o usuário com uma interface bem definida.

4.2.2 Processamento paralelo

Com base no modelo descrito anteriormente, fica claro que pode haver relações de precedência (diretas ou indiretas) entre algumas subtarefas componentes de uma tarefa complexa: existe uma relação de precedência entre duas subtarefas se e somente se existe um caminho no grafo entre elas. Subtarefas entre as quais há uma relação de precedência precisam necessariamente ser executadas em ordem; subtarefas entre as quais não há relação de precedência, no entanto, podem ser realizadas em qualquer ordem, inclusive simultaneamente. O objetivo do desenvolvimento de mecanismos para o processamento paralelo é identificar maneiras de dividir uma tarefa em subtarefas de forma a explorar a possibilidade de processamento simultâneo e, assim, minimizar o tempo necessário para finalizar a tarefa. No entanto, embora seja possível dividir o processamento em um grande número de subtarefas, é preciso levar em conta o tempo que o sistema demora para realizar a comunicação de dados entre subtarefas distintas. Em alguns sistemas para o processamento paralelo, esse tempo pode ser significativamente grande e, portanto, nem todas as possibilidades de processamento paralelo podem ser utilizadas eficientemente.

Abordagens para o processamento paralelo

O desenvolvimento de programas para o processamento paralelo segue diversos paradigmas (Chaudhuri, 1992, p. 52–56; Gibbons e Rytter, 1988, p. 6–19; Carriero e Gelernter, 1990, p. 14–20); no entanto, podemos dizer que há duas abordagens básicas (Kumar et al., 1994, p. 527): a paralelização sobre os dados e a paralelização sobre o controle. Se o sistema promove a paralelização da carga através da realização paralela de seqüências de instruções similares sobre diferentes conjuntos de dados, ele realiza paralelização sobre os dados. Se, por outro lado, o sistema promove a paralelização da carga através da realização paralela de seqüências de instruções diferentes sobre os dados (sejam dados diferentes ou não), ele realiza paralelização sobre o controle.

Um exemplo de paralelismo sobre os dados é uma aplicação que aplica uma transformação sobre os *pixels* de uma imagem em paralelo para gerar uma nova imagem onde o processamento de cada novo *pixel* depende apenas dos *pixels* vizinhos na imagem original. Numa aplicação desse tipo, pode-se associar uma subtarefa independente a cada *pixel* da imagem a ser processada; cada uma das subtarefas é idêntica à outra, mas opera sobre dados diferentes. Por outro lado, um exemplo de paralelismo sobre o controle é uma aplicação para a detecção de padrões em imagens que utilize diversos algoritmos diferentes para identificar os elementos da imagem e, com base nos resultados desses diferentes algoritmos, identifique os objetos nela representados. Nesse caso, pode-se associar uma subtarefa independente a cada um dos algoritmos a ser executados; cada uma das subtarefas é diferente das outras, mas todas operam sobre os mesmos dados (a imagem completa). Ainda outro exemplo de paralelismo sobre o controle é um sistema semelhante ao anterior mas que também procure reconhecer padrões sonoros. Nesse sistema, algumas subtarefas são voltadas para o processamento de imagens enquanto outras são voltadas para o processamento de áudio; as diversas subtarefas do sistema são diferentes entre si e operam sobre conjuntos de dados distintos.

Essas formas de paralelismo pressupõem que é possível encontrar subtarefas de uma tarefa tais que cada subtarefa não tem relações de precedência com outras subtarefas a serem executadas simultaneamente. Uma versão interessante do paralelismo sobre o controle, adequada para aplicações que precisam realizar um mesmo processamento sobre um grande volume de dados, é o uso de uma linha de produção (*pipeline*). Num sistema desse tipo, uma tarefa é dividida em subtarefas ordenadas linearmente tais que cada subtarefa (exceto a primeira) é dependente da subtarefa anterior. Quando uma subtarefa termina o processamento sobre um subconjunto dos dados, a próxima subtarefa pode ser executada e, além disso, a mesma subtarefa também pode ser executada novamente, agora sobre um novo subconjunto dos dados. Numa linha de produção, o tempo necessário para o processamento de um único subconjunto dos dados é constante, independentemente do número de subtarefas em execução no sistema; no entanto, o número de subconjuntos dos dados que podem ser processados ao longo de um intervalo de tempo relativamente longo cresce proporcionalmente ao número de subtarefas realizadas simultaneamente.

É interessante observar que o paralelismo sobre os dados oferece a oportunidade de aumentar o número de tarefas simultâneas em um sistema em função do volume dos dados a ser processados; por outro lado, o paralelismo sobre o controle está limitado pela possibilidade de subdivisão do algoritmo que implementa a solução do problema em partes distintas (Kumar et al., 1994, p. 528). Portanto, o paralelismo sobre os dados permite que o sistema cresça para abordar problemas de grande escala (ou seja, o sistema é *scalable*); o paralelismo sobre o controle, por sua vez, não possui essa característica. Por outro lado, em alguns casos pode não ser possível encontrar um mecanismo eficiente para o paralelismo sobre os dados, tornando o paralelismo sobre o controle a única opção viável. Finalmente, as duas formas de paralelismo não são incompatíveis entre si: diversas aplicações podem se beneficiar de uma combinação das duas técnicas.

4.3 Sistemas de tempo real

De maneira geral, um sistema de tempo real pode ser definido como um sistema cujo funcionamento correto depende não apenas da correção das operações realizadas, mas também do momento em que elas são finalizadas (Stankovic, 1988). Normalmente, várias operações podem ser executadas simultaneamente em um sistema de tempo real, e cada uma delas pode ter diferentes características em relação ao tempo. Por exemplo, algumas podem ser periódicas e outras não; algumas podem precisar ser executadas em um curto espaço de tempo, outras podem ser processadas ao longo de um grande período etc. No entanto, independentemente dessas diversas características, o que define o sistema como um sistema de tempo real é a presença de prazos para o término da execução de pelo menos uma tarefa.

Podemos dizer que um sistema de tempo real, por definição, interage com o "mundo real" (Gallmeister, 1995, p. 41). De fato, com base no modelo acima, se o momento em que uma operação é terminada determina a correção no funcionamento do sistema, necessariamente isso ocorre porque os dados produzidos por essa operação serão utilizados por alguma entidade fora do sistema computacional; se esse não fosse o caso, apenas a correção dos dados fornecidos seria relevante (pois os computadores são

sistemas determinísticos). Uma outra opção é considerar que o momento em que o processamento é finalizado é importante para que o sistema esteja pronto para processar mais dados antes que novos dados estejam disponíveis na entrada do sistema; pelo mesmo motivo, necessariamente esses dados são produzidos por uma entidade que não faz parte do sistema computacional. Grande parte dos sistemas de tempo real, no entanto, interagem com o "mundo real" tanto em sua entrada quanto em sua saída; em geral, esses sistemas podem ser chamados de sistemas reativos, ou seja, que reagem a condições do "mundo real". Um exemplo simples de aplicação que produz dados que são utilizados em tempo real por um sistema não-computacional é um reprodutor de multimídia; já um exemplo de aplicação que processa dados de entrada fornecidos em tempo real por um sistema não-computacional é um sistema que registra periodicamente diversas condições atmosféricas. Finalmente, um processador de áudio ou um sistema de controle de vôo em um avião são exemplos de sistemas reativos.

4.3.1 Prazos rígidos e flexíveis

Sistemas de tempo real são aqueles em que existem tarefas com prazos para serem executadas; portanto, as características dessas tarefas determinam as características dos sistemas de tempo real. Tradicionalmente, tarefas em sistemas de tempo real são divididas entre tarefas com prazo rígido (*hard deadline*) ou com prazo brando ou flexível (*soft deadline*). No entanto, as definições sobre o que são prazos rígidos e prazos flexíveis variam na literatura (Liu, 2000, p. 27–29; Farines, Fraga e Oliveira, 2000, p. 7).

- Uma definição comumente usada é a de que um prazo é rígido se uma falha em cumprir o prazo for uma falha fatal. De acordo com essa definição, uma falha é fatal se suas conseqüências excedem em muito os benefícios normais oferecidos pelo sistema (como em sistemas de controle de vôo ou de gerenciamento de uma usina nuclear), caso em que a falha é dita catastrófica. Se as conseqüências de uma falha em cumprir o prazo são da mesma ordem de grandeza que os benefícios normais do sistema (como em sistemas de telefonia ou de processamento bancário), o prazo é considerado flexível. De acordo com essa definição, uma falha em cumprir um prazo flexível consiste em um erro aceitável de processamento. Em geral, o que ocorre é que um sistema que lida com prazos flexíveis sofre algum tipo de degradação na *qualidade do serviço* oferecido de acordo com o aumento no número de prazos não cumpridos.

- Outra definição comum é a que distingue prazos rígidos e flexíveis de acordo com a possibilidade de utilização dos resultados do processamento mesmo após o prazo esperado. De acordo com essa definição, se uma falha em cumprir o prazo de uma tarefa torna o resultado do processamento inútil, o prazo é rígido; se o resultado do processamento pode ainda ser utilizado mesmo após uma falha em cumprir o prazo (em geral, com utilidade progressivamente reduzida), o prazo é flexível. Essa definição supõe, implicitamente, que uma aplicação com prazos flexíveis implementa mecanismos para adaptação a essas falhas; nesses casos, pode-se inclusive definir limites estatísticos para o número de falhas temporais aceitáveis para garantir o funcionamento adequado da aplicação.

- Ainda outra definição é aquela que propõe que prazos rígidos são aqueles em que falhas não são aceitáveis; já prazos flexíveis são aqueles em que há uma probabilidade, relativamente alta, de que os prazos serão cumpridos. Normalmente, essa definição está vinculada à divisão de sistemas de tempo real entre sistemas de resposta garantida (pode-se demonstrar que todos os prazos rígidos serão cumpridos) e de resposta de melhor esforço (através de experimentos, simulações ou demonstrações, pode-se mostrar que o sistema provavelmente vai cumprir a grande maioria dos prazos flexíveis). Essa definição, no entanto, não deixa claro qual o comportamento esperado de um sistema de melhor esforço em caso de falha.

De maneira geral, as definições acima para prazos rígidos, embora diferentes, são bastante consistentes entre si: um sistema com prazos rígidos deve, de maneira bastante confiável, garantir que esses prazos serão cumpridos. Por outro lado, as definições elencadas para prazos flexíveis são muito menos coerentes entre si: uma falha em cumprir um prazo flexível pode ser vista como um erro excepcional ou como uma condição relativamente comum e esperada; e pode provocar a finalização do processamento, a degradação no desempenho do sistema ou ser administrada por um mecanismo adaptativo.

4.3.2 Sistemas de tempo real rígido e flexível

Seguindo a divisão entre tarefas de tempo real rígido e flexível, os sistemas de tempo real também são divididos em sistemas de tempo real rígido e flexível de acordo com o tipo de tarefas desses sistemas. A despeito das variações nas definições de tarefas de tempo real rígido e flexível, pode-se dizer que sistemas de tempo real rígido e flexível tradicionalmente existem em campos de aplicações distintos. Essa divisão, por sua vez, faz com que diversas características sejam associadas tanto aos sistemas de tempo real rígido quanto aos de tempo real flexível:

- O projeto de sistemas de tempo real rígido deve garantir a previsibilidade dos modos de operação para garantir o cumprimento dos prazos. Isso em geral implica numa razoável complexidade no desenvolvimento, além da necessidade de reserva de recursos para o tratamento do pior caso de operação possível, o que acarreta a sub-utilização dos recursos computacionais na maior parte do tempo. Para garantir o cumprimento de todos os requisitos temporais, tais sistemas não aceitam o agendamento de tarefas que poderiam implicar no não-cumprimento de algum requisito temporal (seja em tempo de projeto do sistema, quando todas as tarefas a ser agendadas são conhecidas, seja em tempo de execução, quando o sistema analisa cada nova requisição para verificar a viabilidade de sua inclusão). Além disso, sistemas operacionais especificamente projetados para dar suporte a aplicações de tempo real[2] são preferidos, pois são capazes de oferecer garantias de tempo explícitas para o agendamento e execução de tarefas etc. em detrimento da oferta de serviços de mais alto nível, como memória virtual, interfaces gráficas etc.

- Por outro lado, em sistemas com prazos flexíveis, é comum que a previsibilidade absoluta, a reserva de recursos e o agendamento de tarefas capaz de dar garantias de tempo real sejam deixados

[2]Por exemplo, RT-Linux (Barabanov e Yodaiken, 1996).

de lado. Em seu lugar são utilizadas análises probabilísticas em tempo de projeto e execução que apenas ofereçam a garantia de que o sistema funcionará corretamente na maior parte do tempo; graças a isso, esses sistemas acabam por utilizar melhor os recursos computacionais disponíveis (Farines, Fraga e Oliveira, 2000, p. 17–19). Esses sistemas comumente se utilizam de sistemas operacionais de uso geral, que não oferecem garantias de tempo real mas que, em compensação, oferecem diversos serviços de alto nível, como interfaces gráficas, sistemas de arquivos etc. Também é comum que os prazos em um sistema desse tipo sejam menos estritos, já que os sistemas operacionais de uso geral normalmente possuem latências de interrupção, agendamento etc. relativamente grandes. Ainda assim, dado que a possibilidade de falhas temporais desses sistemas não é desprezível, vários sistemas voltados para a gerência de prazos flexíveis apresentam algum tipo de mecanismo adaptativo para manter a operação do sistema com degradação na qualidade do serviço no caso de falhas temporais. Por exemplo, uma aplicação para a reprodução de arquivos de vídeo pode descartar alguns quadros durante a execução no caso de desempenho insuficiente do computador[3].

4.3.3 Sistemas de tempo real firme

Embora diversas aplicações de tempo real, em particular as voltadas para o processamento de multimídia (como reprodutores de vídeo), possam ser facilmente caracterizadas como aplicações de tempo real flexível, outras (como ferramentas interativas para a edição e processamento de multimídia) não correspondem tão bem a essa categoria: elas não podem utilizar mecanismos adaptativos para contornar falhas temporais (no caso do exemplo, isso causaria ruídos ou "pulos" no áudio, quadros de vídeo faltando ou repetidos etc. que seriam aceitáveis em outros ambientes, mas não em uma ferramenta profissional de edição) e, portanto, dependem de garantias temporais as mais próximas possíveis daquelas dos sistemas de tempo real rígido. Por outro lado, falhas eventuais não são catastróficas, o que significa que garantias probabilísticas de correção temporal são suficientes para que um sistema desse tipo possa ser considerado "correto" (ainda usando o mesmo exemplo, a funcionalidade da aplicação não é significativamente afetada se falhas esporádicas acontecem, digamos, uma vez a cada hora; se o usuário inicia a aplicação e ela executa corretamente por alguns segundos, é provável que ela vá funcionar corretamente também por períodos mais longos, e mesmo uma evidência empírica como essa sobre a "correção temporal" do sistema pode ser suficiente para o usuário nesse caso.).

Como não é possível lidar com erros temporais de forma automática, esses sistemas não precisam implementar mecanismos sofisticados para correção de erros ou adaptação como a maioria dos sistemas de tempo real flexível; eles precisam apenas ser capazes de detectar erros, que podem ser tratados como outros erros ordinários do sistema. Por exemplo, durante a gravação de um evento musical ao vivo, uma falha pode ser registrada para correção posterior. A redução na qualidade da gravação, por outro lado, não é uma solução aceitável nesse cenário. Durante o trabalho posterior de edição, uma falha temporal pode simplesmente levar o sistema a parar o processamento e apresentar uma mensagem de

[3]Diversos trabalhos têm sido dedicados a este problema, em particular no caso de sistemas distribuídos; c.f., por exemplo, Chen et al. (1995); Vieira (1999).

erro para o usuário, que pode reiniciar a operação. Nenhum desses mecanismos pode ser caracterizado como "adaptação" por parte do sistema: as falhas temporais são, de fato, tratadas como erros de processamento e não como condições às quais o sistema pode se adaptar.

Além disso, tais sistemas são preferencialmente baseados em computadores e sistemas operacionais de uso geral, pois estes estão disponíveis por baixo custo e oferecem interfaces de usuário sofisticadas e um amplo repertório de serviços por parte do sistema operacional. O uso de sistemas operacionais de uso geral também garante a compatibilidade do sistema com uma gama muito maior de dispositivos de E/S, particularmente para multimídia, de uso geral; esses sistemas normalmente dispõem dos controladores para esses dispositivos, diferentemente do que normalmente ocorre em sistemas operacionais projetados especificamente para operação em tempo real. Por outro lado, em alguns casos (como no sistema objeto deste trabalho) os prazos envolvidos no processamento são relativamente pequenos, diferentemente do que normalmente ocorre em sistemas de tempo real flexível baseados em sistemas operacionais de uso geral.

Sistemas com as características "híbridas" descritas assim têm sido recentemente classificados como sistemas de tempo real firme (*firm* — Srinivasan et al., 1998). Tais sistemas são caracterizados por serem baseados em computadores e sistemas operacionais de uso geral, necessitarem apenas de garantias probabilísticas sobre a correção temporal e por tratarem erros temporais como erros ordinários do sistema, sem a necessidade ou possibilidade de adaptação. Graças ao crescente número de aplicações de tempo real firme, sistemas operacionais de uso geral como Linux e MacOS X têm sofrido melhorias no sentido de oferecer boas características de desempenho para esse tipo de aplicação. Com combinações adequadas de hardware e software, Linux, Windows e MacOS são capazes de oferecer níveis típicos e máximos de latência adequados para o processamento de multimídia: de 3 a $6ms$ (MacMillan, Droettboom e Fujinaga, 2001).

4.3.4 Caracterização de sistemas de tempo real

Recentemente, algumas abordagens têm procurado simplificar o desenvolvimento de aplicações de tempo real, oferecer melhores garantias temporais e, ao mesmo tempo, permitir uma melhor utilização dos recursos computacionais disponíveis. Essas abordagens consistem na combinação de tarefas com prazos rígidos e flexíveis de maneira controlada (Hamdaoui e Ramanathan, 1995; Bernat e Burns, 1997) ou em permitir a realização de apenas uma parte do processamento quando há sobrecargas no sistema (Liu et al., 1994).

Nesse contexto, foram propostas maneiras de caracterizar as necessidades de tempo real de um sistema de acordo com o número máximo de falhas temporais aceitáveis em um conjunto de prazos (Hamdaoui e Ramanathan, 1995; Bernat, Burns e Llamosi, 2001; Hu et al., s.d.). Pode-se expressar as necessidades de tempo real de um sistema tomando-se dois números k e m tais que a cada k tarefas consecutivas, pelo menos m precisam ter seus prazos cumpridos para a operação adequada do sistema. Um sistema de tempo real rígido pode ser definido como um sistema onde $m = 1$ e $k = 1$; um sistema de tempo real flexível pode ser definido como um sistema onde $m \ll k$. Se as necessidades de tempo

real de um sistema são expressas em termos dos números m e k, diz-se que esse é um sistema que tem (m, k) prazos *firmes*; sistemas desenvolvidos para oferecer garantias temporais expressas dessa forma são geralmente chamados de *sistemas de tempo real firme* (tornando também essa designação variável na literatura).

Graças a essas novas caracterizações, a divisão entre sistemas de tempo real rígido, flexível e firme é cada vez mais imprecisa; como vimos, mesmo as definições dessas categorias variam grandemente. Assim, essa divisão é útil apenas enquanto uma categorização genérica e aproximativa; uma categorização mais precisa deve levar em conta os diversos aspectos que caracterizam os sistemas de tempo real. Assim, numa tentativa de especificar mais claramente o sistema aqui descrito, vamos procurar caracterizá-lo de acordo com um conjunto de aspectos básicos (Stankovic, 1992; Farines, Fraga e Oliveira, 2000, p. 7; Liu, 2000, p. 49–50; Kopetz e Veríssimo, 1993)[4]:

- O sistema oferece resposta garantida ou de melhor esforço? Ou seja, é possível demonstrar ou, pelo menos, estimar com grande confiabilidade que o sistema vai cumprir estritamente todos os prazos ou simplesmente espera-se, com base em análises estatísticas, que o sistema cumpra a grande maioria dos prazos?

- No caso de sistemas de resposta garantida, a resposta é garantida por um conhecimento *a priori* do sistema que permite que o agendamento das tarefas seja pré-calculado (agendamento *offline*) ou é garantida por um algoritmo que determina o agendamento em tempo de execução (agendamento *online*)?

- No caso de sistemas de melhor esforço, o sistema possui mecanismos para adaptar-se às falhas temporais, falhas temporais causam degradação no serviço ou são tratadas como erros? Uma falha temporal pode ser tratada como um evento relativamente comum, caso em que o sistema realizará algum procedimento para minimizar ou eliminar os efeitos da falha (podemos citar novamente o exemplo da aplicação de reprodução de vídeo que pode simplesmente reproduzir duas vezes um mesmo quadro de imagem caso não tenha sido possível processar um quadro dentro do prazo). Uma falha temporal pode também não ser tratada de nenhuma forma especial, provocando degradação na qualidade do serviço (no caso de uma aplicação de vídeo, falhas temporais poderiam ser responsáveis pela perda de sincronização entre o áudio e o vídeo no sistema). Finalmente, uma falha temporal pode ser tratada como um erro e provocar a finalização da aplicação, possivelmente com procedimentos para que a finalização seja correta (por exemplo, uma falha temporal no sistema de freio de um trem pode iniciar um procedimento para parar o trem); ou pode ser tratada como um erro que deve ser contornado por um procedimento específico (como discutido anteriormente, uma falha num sistema de gravação de áudio pode fazer o sistema registrar a falha para posterior edição e continuar a gravação normalmente).

[4]Esses aspectos são razoavelmente independentes entre si, ou seja, uma característica não implica nessessariamente em outra; no entanto, como vimos, alguns conjuntos de características são comumente associados a alguns tipos de aplicações (justamente os sistemas de tempo real rígido e flexível). Assim, por exemplo, sistemas onde falhas temporais têm graves conseqüências sobre o mundo real são normalmente associados a sistemas de resposta garantida.

- As tarefas têm requisitos temporais estritos? Ou seja, o tempo entre o agendamento para a execução de uma tarefa e o prazo para o término de sua execução é pequeno? Se os requisitos temporais são estritos, é preciso que haja uma preocupação grande com a velocidade e a previsibilidade das diversas partes do sistema; se os requisitos temporais são pouco estritos, algoritmos mais complexos e menos previsíveis podem ser utilizados pelo sistema e flutuações no tempo de processamento de diversas tarefas podem ser consideradas desprezíveis.

- O resultado de um processamento realizado após o seu prazo tem algum valor para o sistema? Em alguns casos, o resultado atrasado de um cálculo pode servir como uma estimativa do valor real e, portanto, ser útil. Por exemplo, no caso de arquivos de vídeo compactados de acordo com o padrão MPEG, um quadro de imagem pode ser baseado em um quadro anterior; assim, a falha em decodificar um quadro a tempo não permite que esse quadro possa ser apresentado dentro do prazo para o usuário, mas o resultado atrasado da decodificação desse quadro pode ser usado para o processamento dos próximos quadros.

- O sistema pode fazer uso de computação imprecisa? Em casos em que o sistema detecta que não cumprirá um prazo, pode ser possível realizar um processamento parcial capaz de oferecer uma estimativa dos resultados reais que teriam sido obtidos caso não houvesse escassez de tempo para o processamento. Por exemplo, um robô pode ajustar sua trajetória de acordo com uma estimativa desse tipo mesmo que a sua posição exata não possa ter sido calculada dentro do prazo.

- As tarefas de tempo real do sistema são periódicas ou aperiódicas? Um sistema de tempo real pode reagir a eventos aperiódicos e que, portanto, podem ocorrer a qualquer momento, ou pode realizar diversas operações repetidamente a intervalos de tempo regulares.

- Qual o ambiente de execução (hardware e software) previsto para o sistema? Alguns sistemas dependem de combinações específicas de hardware e software para funcionar; outros visam a portabilidade.

- O sistema pode ser modificado dinamicamente, com adição ou remoção de tarefas durante a execução? Sistemas capazes de oferecer a possibilidade do acréscimo ou remoção de tarefas são evidentemente mais flexíveis, mas é muito mais difícil garantir sua previsibilidade.

- Qual a gravidade de uma falha do sistema em relação ao mundo real? Uma falha em um sistema multimídia para a transmissão de um evento artístico pode ser considerada grave no sentido que pode destruir o interesse pelo evento; já uma falha em um sistema de controle de vôo de um avião pode ser responsável pela queda do avião.

- Como caracterizar o sistema em termos de prazos (m, k)-firmes? Essa caracterização oferece uma visão mais clara das necessidades de tempo real de um sistema que a mera divisão em sistemas de tempo real rígido ou flexível.

4.4 Processamento paralelo em tempo real

Num sistema em que há apenas uma tarefa a ser executada, o desempenho é expressado diretamente pelo tempo que o sistema demora para executar essa tarefa. No entanto, se há mais de uma tarefa a ser executada, podemos ver o desempenho do sistema por dois pontos de vista distintos: o tempo que o sistema demora para executar cada uma das tarefas individualmente e o número de tarefas que o sistema é capaz de executar durante um intervalo de tempo relativamente longo. Embora esses dois aspectos estejam relacionados, essa relação não é direta: por exemplo, se o sistema demorar um tempo significativamente grande para realizar uma tarefa mas for capaz de processar um grande número de tarefas simultaneamente, o sistema será capaz de concluir um grande número de tarefas durante um intervalo de tempo razoavelmente longo apesar de o tempo para o processamento de cada tarefa ser grande.

O volume de tarefas que o sistema é capaz de processar durante um intervalo de tempo relativamente longo é normalmente chamado de vazão de processamento do sistema (*throughput*); o tempo que o sistema demora para terminar o processamento de uma única tarefa é chamado de latência de processamento do sistema. Evidentemente, é desejável que um sistema computacional ofereça baixa latência e alta vazão de processamento; no entanto, sistemas paralelos muitas vezes sacrificam uma dessas características em prol da outra (Subhlok e Vondran, 1996; Choudhary et al., 1994).

Sistemas de tempo real onde há processamento periódico, em particular, podem definir limites rígidos tanto para a latência quanto para a vazão de processamento. Aplicações onde há o processamento em tempo real de mídias contínuas, como sistemas interativos para multimídia, se enquadram nessa categoria. Por exemplo, um sistema para o processamento de áudio em tempo real deve ser capaz de processar um grande número de canais de áudio simultaneamente com altas taxas de amostragem (entre 44.1KHz e 192KHz no caso de sistemas profissionais) e, ao mesmo tempo, garantir níveis de latência baixos o suficiente para o usuário.

Não são apenas sistemas interativos que dependem de baixas latências e grandes volumes de processamento: um sistema de visão computacional para o controle de um veículo em movimento, por exemplo, envolve limitações semelhantes. Um sistema desse tipo deve ser capaz de estimar a posição, velocidade e trajetória do veículo controlado e dos objetos à sua volta; para que isso seja possível, o sistema deve ser capaz de processar imagens de diversas câmeras de vídeo periodicamente e atuar sobre a direção e velocidade do veículo. Por um lado, o sistema precisa ser capaz de reagir a uma ameaça de colisão em um espaço de tempo relativamente curto; por outro, uma maior freqüência de amostragem de imagens pode permitir ao sistema estimar trajetórias com mais precisão. Se o sistema capturar imagens de um veículo que segue uma trajetória em zigue-zague com uma freqüência muito baixa, a trajetória será erroneamente estimada como sendo linear, o que sugere que as imagens devem ser capturadas com pelo menos alguns Hertz de freqüência. Já um tempo de resposta entre $500ms$ e $1500ms$ parece ser adequado, já que esse é o tempo de reação de melhor caso em se tratando de um motorista humano (Triggs e Harris, 1982). Observemos que, nesses dois exemplos, o período de amostragem dos dados não está diretamente relacionado à latência esperada do sistema.

Considerando que apenas um tipo de tarefa completa existe no sistema[5], se a latência de processamento l necessária para o funcionamento do sistema for menor ou igual ao menor intervalo de tempo δ que pode haver entre o início de duas tarefas consecutivas, cada tarefa necessariamente precisa ter sido terminada antes de a próxima ser iniciada; portanto, um mecanismo que respeite a latência máxima requerida pelo sistema também garante que o sistema será capaz de respeitar a vazão de processamento mínima do sistema. Como vimos, é possível dividir tal tarefa em subtarefas e procurar promover a paralelização do processamento através do processamento simultâneo de subtarefas entre as quais não há relação de precedência. Nesse cenário, no entanto, não é possível fazer uso do mecanismo de linha de produção descrito anteriormente: a vazão e a latência de processamento estão diretamente ligadas nesse caso.

Por outro lado, se $\delta < l$, é possível satisfazer os requisitos de latência e vazão de processamento se cada tarefa sempre for terminada em tempo menor que δ; nesse caso, no entanto, essa garantia não é necessária. Se $\delta < l$, uma maneira de garantir que os requisitos de latência e de vazão de processamento sejam satisfeitos é garantir que o sistema seja capaz de realizar uma tarefa em tempo menor que l e implementar um mecanismo de linha de produção que viabilize o processamento paralelo de várias tarefas (King, Chou e Ni, 1988). Para isso, a tarefa é dividida em tantas subtarefas em seqüência quantas forem necessárias para que cada uma delas possa ser executada por um processador em tempo não superior a δ. Se um conjunto de tarefas $t_1, t_2 \ldots t_n$ começa a ser processado pelo sistema em seqüência com um intervalo de tempo δ entre cada tarefa, o que ocorre é que o primeiro processador começa a executar a primeira subtarefa de t_1; após transcorrido o tempo δ, os dados produzidos pelo primeiro processador são enviados para o segundo, que continua a execução de t_1 enquanto o primeiro processador começa a execução de t_2; após transcorrido outro intervalo de tempo δ, o terceiro processador recebe os dados produzidos pelo segundo e continua a execução de t_1, o segundo processador recebe os dados produzidos pelo primeiro e continua a execução de t_2 e o primeiro processador inicia a execução de t_3. Esse processo continua até que $t_1 \ldots tn$ sejam finalizadas em seqüência pelo último processador. Esse sistema garante que novas tarefas podem ser iniciadas pelo sistema a cada intervalo de tempo δ e que cada tarefa será completada em tempo menor que l.

Observemos que, nesse caso, é fundamental que cada uma das subtarefas do sistema seja finalizada em tempo menor que δ; caso contrário, a próxima subtarefa, parte da tarefa seguinte, não poderá ser executada a tempo. De maneira semelhante, o tempo gasto na comunicação de dados entre duas subtarefas também precisa ser menor que δ. Além disso, como levantamos anteriormente, o tempo para a execução de todas as subtarefas de uma tarefa somado ao tempo total gasto na comunicação de dados entre as subtarefas precisa ser menor ou igual à latência desejada para o sistema. Podemos, de fato, considerar a comunicação entre cada par de processadores como uma tarefa similar às demais, o que simplifica ainda mais o modelo de paralelização. Finalmente, observemos que, nesse cenário, cada processador executa sempre a mesma subtarefa sobre os dados que recebe.

[5]Podemos discutir uma única tarefa sem perda de generalidade se considerarmos que o raciocínio pode ser expandido para um conjunto de tarefas se todas as tarefas do sistema não possuem relações de precedência entre si. Por outro lado, se este não for o caso, podemos tomar tarefas que possuem relações de precedência entre si e encará-las como uma única tarefa.

Na discussão acima, assumimos que existe apenas um tipo de tarefa a ser executada pelo sistema. Vale notar que podemos considerar um sistema como o descrito aqui como uma parte de um sistema mais complexo, onde os dados fornecidos pela linha de produção são utilizados por outras tarefas em execução em outros processadores sem que haja alterações no sistema descrito. Também é possível implementar mecanismos para utilizar linhas de produção de forma similar à descrita com mais de uma tarefa; esses mecanismos, no entanto, fogem ao escopo deste trabalho[6].

[6]Há vários trabalhos a esse respeito; c.f., por exemplo, Sevcik (1989) e os já citados Subhlok e Vondran (1996) e Choudhary et al. (1994).

Parte II

O sistema

Capítulo 5

O middleware

O processamento de mídias contínuas em sistemas distribuídos em tempo real com garantias de baixa latência oferece dificuldades específicas. Embora este trabalho enfoque o processamento de áudio, as soluções genéricas referentes ao processamento distribuído de mídias contínuas foram implementadas em um pequeno middleware especificamente criado para abordar essa classe de problemas. Com base nesse middleware foi desenvolvida uma aplicação (DLADSPA — Distributed LADSPA) que permite o processamento distribuído de áudio por parte de aplicações legadas compatíveis com a especificação LADSPA através do uso de módulos DSP também compatíveis com essa especificação. Neste capítulo, apresentaremos como diversos aspectos foram encarados no desenvolvimento desse middleware e descreveremos sua implementação e funcionamento.

5.1 Mecanismos de funcionamento

Como vimos, embora alguns algoritmos para o processamento de multimídia exijam altas capacidades de processamento por parte do computador, a paralelização desses algoritmos envolve a reimplementação de cada um deles; não há muito espaço para a generalização desse processo. No entanto, sabemos que o processamento de multimídia consiste na manipulação de mídias contínuas (Steinmetz e Nahrstedt, 1995, p. 13, 17, 571). De maneira geral, o acúmulo de transformações a serem aplicadas a diferentes fluxos de dados também pode provocar a exaustão da capacidade do sistema, mesmo que cada uma delas individualmente represente pouca carga de utilização do equipamento. De fato, os algoritmos de áudio de alto custo computacional correspondem a uma parcela pequena dos algoritmos normalmente utilizados; por outro lado, em um estúdio de gravação não é incomum que dezenas de canais de áudio sejam processados simultaneamente.

Como vimos na Seção 1.3.6, um mecanismo que permita o processamento de diferentes fluxos de dados por diferentes máquinas ou processadores pode oferecer uma capacidade de processamento combinada muito maior que um sistema monoprocessado isolado na maior parte das situações. Um sistema desse tipo é simples de ser implementado porque realiza o processamento paralelo de tarefas que não têm relação de precedência entre si. Por outro lado, múltiplas transformações aplicadas sobre um único

fluxo de dados (que, portanto, possuem relações de precedência entre si) também podem obter ganhos de desempenho através do uso de linhas de produção (Seção 4.4). No entanto, é mais comum que o acúmulo de canais de áudio a serem processados seja o principal responsável pela exaustão na capacidade de processamento de um sistema. Além disso, o uso de linhas de produção provoca um aumento na latência de processamento. Por essas razões, o middleware aqui descrito é baseado na alocação de fluxos de dados independentes para serem processados em diferentes máquinas, mas sem a possibilidade do uso de linhas de produção[1]. Essa possibilidade, no entanto, pode ser explorada em trabalhos futuros.

5.1.1 Tempo real

Tendo em mente os aspectos discutidos anteriormente sobre sistemas de tempo real e as características desejáveis do nosso sistema, podemos levantar as suas características de tempo real:

- As tarefas de tempo real do sistema são periódicas (tarefas relacionadas a E/S de disco, interface com o usuário etc. não são periódicas, mas também não são tarefas de tempo real);
- O sistema deve ser capaz de ser executado em sistemas operacionais e arquiteturas de hardware de uso geral, e não em sistemas especiais para tempo real. Como discutido anteriormente, nosso principal objetivo é garantir a compatibilidade com o sistema Linux;
- O sistema não precisa implementar mecanismos complexos para oferecer resposta garantida de funcionamento; as variações temporais devidas à rede e ao próprio sistema operacional sugerem que o sistema necessariamente é um sistema de melhor esforço;
- Por outro lado, o comportamento esperado do sistema é que todos os prazos sejam cumpridos; ou seja, o sistema tem $(1, 1)$ prazos firmes;
- As necessidades de tempo do sistema são medianamente estritas; latências da ordem de $1ms$ estão próximas do limite que o Linux é capaz de oferecer, mas são relativamente grandes para sistemas operacionais especificamente desenvolvidos para o processamento em tempo real;
- Uma falha temporal do sistema não representa uma falha catastrófica; embora falhas regulares não sejam aceitáveis, o pior efeito que uma falha desse tipo pode oferecer é um ruído indesejável no resultado do processamento;
- Por outro lado, uma falha temporal é um erro ao qual o sistema não tem como se adaptar; diversos mecanismos podem ser implementados para o tratamento desses erros, mas não cabe nenhum tipo de adaptação;
- Justamente porque falhas temporais são erros, o sistema não pode fazer uso de computação imprecisa;

[1] Na verdade, como veremos adiante, o middleware faz uso de uma linha de produção; no entanto, ela é utilizada apenas para otimizar a comunicação via rede, e não para aumentar o número de processadores a serem usados na execução de uma tarefa.

- Como um dos objetivos do sistema é garantir a baixa latência, não há utilidade no sistema para resultados de cálculos obtidos após seu prazo;
- O sistema deve permitir ao usuário definir facilmente quais os algoritmos serão aplicados sobre quais fluxos de dados. Deve ser possível acrescentar ou remover tarefas do sistema facilmente; essas alterações, no entanto, podem ser realizadas sem restrições temporais.

O sistema descrito aqui foi desenvolvido de acordo com essas características. De maneira geral, esse conjunto de características aproxima o sistema da definição genérica de sistemas de tempo real firme; ainda assim, é útil oferecer definições mais claras dos pressupostos e objetivos utilizados durante o desenvolvimento.

5.1.2 Transmissão de blocos de dados

A comunicação de dados entre aplicações via rede pode envolver muitos tipos de dados; especificações como a CORBA (Henning e Vinoski, 2001; Tanenbaum e Steen, 2002, p. 493–525; Siegel, 2000) procuram garantir um alto nível de abstração nesse tipo de comunicação, permitindo que uma ampla gama de tipos de dados sejam enviados e recebidos transparentemente via rede. O custo de sistemas desse tipo, no entanto, é a possível perda de desempenho ou o aumento na latência da comunicação.

O mecanismo de distribuição aqui apresentado, por sua vez, não pretende competir com sistemas como CORBA em termos de recursos, nível de abstração ou flexibilidade; pelo contrário, sua intenção é apenas estabelecer um método simples, porém eficiente, para a transmissão de dados sincronizadamente em tempo real entre máquinas em uma rede local. Por esta razão, ele lida apenas com *buffers* de dados, ou seja, não trata os dados de maneira orientada a objetos e, pela mesma razão, é limitado aos tipos de dados nativos da linguagem C: inteiros, números de ponto flutuante ou caracteres, que têm correspondência mais ou menos direta com as representações primitivas dos dados no nível do hardware.

5.1.3 Operação síncrona

Uma das dificuldades de um sistema distribuído é garantir a coerência temporal entre as diversas máquinas do sistema; no caso do processamento de mídias contínuas em tempo real, esse problema se torna ainda mais premente. E, como vimos no Capítulo 3, um sistema que procure oferecer processamento com baixas latências deve ser capaz de operar com base em funções *callback* executadas em sincronia com as requisições de interrupção geradas pelo dispositivo de E/S de mídia. No caso de um sistema distribuído para o processamento em tempo real com baixas latências, portanto, a coerência temporal deve consistir em um mecanismo que garanta que o processamento distribuído ocorra de forma síncrona às interrupções de hardware do dispositivo de E/S de mídia.

Esse problema pode ser resolvido com a ajuda de hardware específico: se todas as diversas máquinas de um sistema distribuído possuírem placas de som com suporte a sincronização unificada (*word clock*), é possível garantir o processamento síncrono em todas as máquinas através das interrupções geradas pelo hardware de áudio. Nesse caso, cada nó teria condições de operar como um sistema independente,

mas a sincronização entre as diferentes máquinas estaria garantida pelo sincronismo entre as requisições de interrupções do hardware.

Essa configuração, no entanto, envolve um aumento de custo significativo em cada nó (pois as placas de som com suporte a sincronização unificada são placas de uso profissional, custando metade do preço de um computador mediano nos EUA). Além disso, o funcionamento independente de cada uma das máquinas traz dificuldades em termos da interface com o usuário. Para uma melhor utilização de um sistema desse tipo, seria fundamental o desenvolvimento de aplicações especificamente projetadas para tirar proveito da arquitetura distribuída do sistema, com o conseqüente aumento de complexidade em cada aplicação; e, nesse caso, o acoplamento entre as máquinas, certamente baseado no tráfego de dados via rede, justificaria a implementação de um mecanismo para a sincronização baseado nesse próprio tráfego, minimizando o interesse do uso do equipamento com suporte a sincronização unificada.

Num sistema que não se utilize de equipamentos desse tipo, é necessário que, a cada interrupção do dispositivo de E/S de mídia, seja executada uma função *callback* que provoque o envio e recebimento dos dados via rede e que provoque a execução das funções *callback* responsáveis pelo processamento dos dados em todas as máquinas remotas; é preciso, portanto, que haja um mecanismo para a troca de dados entre as máquinas e um mecanismo para a sincronização entre as interrupções do dispositivo de E/S e a execução das funções *callback* remotas. Neste cenário, portanto, uma das máquinas tem um papel diferenciado, interagindo com o dispositivo de E/S de mídia e funcionando como coordenador das outras máquinas.

No sistema descrito aqui, optamos por utilizar as próprias interrupções geradas pelo dispositivo de rede no tráfego dos dados como gerenciadoras do sincronismo entre as diversas máquinas do sistema, o que não incorre em custo adicional e garante que o sistema possa funcionar independentemente do tipo de mídia a ser processada e de hardware específico.

Assim, um dispositivo de E/S de mídia em uma das máquinas gera as interrupções que coordenam o tempo do sistema. A interação com esse dispositivo é gerida pela aplicação, que se comunica com ele e executa uma chamada de função ao nosso middleware para que ele processe os dados recebidos e a serem reproduzidos; o middleware, por sua vez, envia os dados via rede para as máquinas remotas. A parte do middleware em execução nas máquinas remotas, originalmente bloqueada em espera por dados da rede, é desbloqueada, lê os dados recebidos e executa uma chamada à função responsável pelo processamento dos dados. Quando essa função é finalizada, os dados são enviados de volta para a máquina com o dispositivo de E/S de mídia, onde são devolvidos para a aplicação que, finalmente, os envia para o dispositivo de E/S. Após o envio dos dados, o middleware remoto volta a bloquear-se em espera por novos dados vindos da rede.

O middleware, portanto, tem estruturas quase simétricas nos dois lados da comunicação: na máquina responsável pela E/S de mídia, o gerenciamento temporal fica a cargo da aplicação (que, por sua vez, é gerida pelas interrupções do dispositivo de E/S) e o middleware é executado como uma função *callback*; nas máquinas responsáveis pelo processamento remoto, o gerenciamento temporal fica a cargo do middleware (gerido pelas interrupções do dispositivo de rede, em consonância com o dispositivo de E/S remoto) e o código da aplicação remota responsável pelo processamento é executado como uma

função *callback*. Essa organização segue o que foi descrito no Capítulo 3 no tocante à necessidade do uso de funções *callback* para o processamento em tempo real com baixa latência, mas separando o recebimento das requisições de interrupção do hardware em uma máquina do processamento das funções *callback* associadas em outras máquinas.

Protocolo de rede orientado a mensagens

Para que isso funcione corretamente, o sistema operacional deve agendar o middleware para execução assim que um bloco de dados completo é recebido em cada uma das máquinas remotas responsáveis pelo processamento dos dados. Esse mecanismo exige que o protocolo de rede utilizado não seja baseado na idéia de fluxos de dados (como o protocolo TCP), mas sim em fluxos de mensagens:

- Quando do envio de dados, o sistema operacional deve enviar cada mensagem imediatamente, sem procurar, por exemplo, concatenar mensagens pequenas em uma mensagem maior, dividir uma mensagem grande em pequenas mensagens[2] ou inserir esperas no envio de dados em função do tráfego estimado da rede;
- De maneira similar, quando da recepção de dados, o sistema operacional deve devolver um bloco de dados completo a cada operação de leitura dos dados, sem concatenar mensagens ou devolver partes de uma mensagem.

Dentre os protocolos da família TCP/IP, o protocolo UDP (Stevens, 1994, p. 143–146) satisfaz esses dois requisitos: cada chamada à função `write()` resulta na transmissão de exatamente um pacote UDP (fragmentado ou não) e, de maneira similar, cada chamada à função `read()` resulta na leitura de exatamente um pacote UDP. Por isso, o sistema aqui descrito baseia-se no protocolo UDP; é possível, no entanto, adaptar o sistema facilmente para o uso de outros protocolos de rede, desde que compatíveis com as restrições acima.

Agendamento de tempo real

Nas máquinas responsáveis pelo processamento remoto, o sistema operacional deve agendar o middleware para ser executado assim que um novo bloco de dados for recebido; o middleware deve ser visto como uma função *callback* responsável por tratar os dados recebidos da rede a cada mensagem completa[3].

Para que isso seja possível, o midleware (e a função *callback* chamada por ele) precisa ser executado com prioridades de tempo real. Como discutimos na Seção 3.3.1, aplicações com prioridade de agendamento de tempo real são agendadas para execução assim que são desbloqueadas, ou seja, assim que há

[2] Isso não significa que não pode haver fragmentação de mensagens; uma mensagem pode ser fragmentada em mais de um pacote Ethernet ou UDP, por exemplo, mas os fragmentos devem serem reagrupados de maneira transparente ao ser recebidos, garantindo a integridade e a unicidade da mensagem.

[3] Não se trata exatamente de tratar todas as interrupções do dispositivo de rede, já que pode haver fragmentação; no entanto, se as mensagens forem pequenas o suficiente para serem enviadas sem fragmentação, então o middleware será efetivamente agendado para execução a cada interrupção do dispositivo de rede.

trabalho a ser realizado por elas. Esse mecanismo garante que, a cada nova mensagem recebida da rede, o middleware é agendado para execução. Como as mensagens completas chegam da rede em consonância com as interrupções geradas pelo dispositivo de E/S de mídia remoto, o agendamento da execução do processamento de todas as máquinas do sistema é gerenciado pelas interrupções do dispositivo de E/S de mídia remoto.

Observemos, portanto, que esse sistema é efetivamente síncrono: todo o processamento em todas as máquinas é disparado em função do momento em que o dispositivo de E/S de mídia requisita uma nova interrupção; a unidade temporal entre todas as máquinas é garantida. No entanto, como veremos a seguir, os dados processados a cada interrupção não são obrigatoriamente os dados correspondentes no tempo àquela interrupção, o que pode soar um tanto inesperado nesse contexto. Assim, embora o sistema seja efetivamente síncrono, talvez o ideal seja chamá-lo de "pseudo-síncrono" ou "quase síncrono".

Vale notar que esse mecanismo pressupõe que não haja perda ou atraso significativo de pacotes na comunicação. Embora essa restrição possa parecer problemática, ela se justifica por duas razões:

1. A necessidade de baixa latência sugere que o sistema deverá ser usado em redes locais; como não há roteadores TCP/IP entre as máquinas (que podem ignorar pacotes em situações de sobrecarga), não há descarte de pacotes nesse nível.

2. Tomando como exemplo redes baseadas no padrão Fast Ethernet, embora colisões possam ser responsáveis por atrasos na transmissão de um pacote, esses atrasos são normalmente da ordem de poucos microssegundos. Tais atrasos são desprezíveis no contexto de nosso sistema. Além disso, embora haja a possibilidade de descarte de pacotes no caso de sobrecarga da rede, essa situação na prática é praticamente inexistente, em particular se assumirmos que podemos contar com a rede exclusivamente para o sistema. Finalmente, o uso de *switches* ao invés de concentradores simples reduz ainda mais as chances de problemas nesse nível. De fato, nos experimentos realizados não houve problemas causados por perdas de pacotes.

5.1.4 Janelas deslizantes

Se, por um lado, nosso objetivo é distribuir o processamento de mídias contínuas em tempo real através de uma rede local, por outro, os sistemas de rede comumente usados têm algumas limitações que dificultam seu uso para essas aplicações. As taxas de transferência de dados das redes, embora crescentes, ainda são relativamente pequenas para o tráfego de vários canais de áudio e vídeo, e essas velocidades relativamente baixas podem introduzir latências na comunicação que precisam ser eliminadas ou, pelo menos, levadas em conta. Além disso, a maioria dos sistemas de rede de baixo custo não oferece mecanismos para a transmissão de dados com garantias de tempo real.

Redes e tempo real

Para processar dados de mídias contínuas remotamente com baixa latência, uma aplicação em execução na máquina que gerencia o dispositivo de E/S de mídia deve:

1. receber os dados capturados pelo dispositivo de E/S de mídia quando da requisição de interrupção gerada pelo dispositivo;

2. enviar os dados capturados para serem processados remotamente via rede;

3. esperar o processamento remoto;

4. receber os dados de volta através da rede; e, finalmente,

5. transferir os dados recebidos para o dispositivo de E/S de mídia antes que ele gere uma nova requisição de interrupção (Figura 5.1).

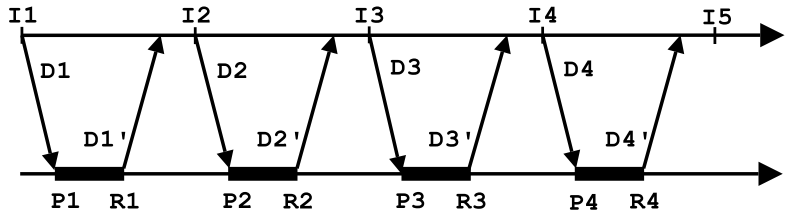


Figura 5.1: A cada interrupção I_n, os dados capturados (D_n) são enviados para a máquina remota, processados (P_n), devolvidos (R_n) à máquina de origem já processados (D'_n), e enviados para o dispositivo de E/S de mídia antes da próxima interrupção (I_{n+1}).

Esse modo de operação utiliza a rede em modo *half-duplex*, além de utilizar tanto a rede quanto o processador remoto por apenas uma fração do período entre cada interrupção. Dadas as atuais tecnologias de rede de baixo custo (primariamente Fast Ethernet), é fácil perceber que um sistema desse tipo oferece muito pouca capacidade de processamento remoto.

De fato, tomando como exemplo o caso do áudio, dependendo da velocidade nominal da rede, da taxa de amostragem e do número de bits de cada amostra de áudio, podemos estimar o número máximo de canais que pode ser enviado e recebido pela rede em tempo real em modo *half-duplex* em função da porcentagem do período gasto pelo processamento remoto:

$$n^o\,de\,canais = \frac{velocidade\,da\,rede\,(em\,bits/s)}{2{*}taxa\,de\,amostragem(em\,Hertz){*}resolucao} * (1 - \frac{\%\,do\,periodo\,gasto\,no\,processamento}{100})$$

Se tomarmos como exemplo uma rede de 100Mbits de velocidade nominal e uma aplicação que processe o áudio como amostras de 32 bits à freqüência de 96KHz[4] utilizando 80% do período no processamento remoto, veremos que o número máximo de canais que podem ser enviados e recebidos pela rede em tempo real é $\frac{100000000}{2*96000*32} * 0,2 \cong 3$ (no caso de taxa de amostragem de 44100Hz, o limite

[4]Os sistemas de gravação de áudio modernos têm trabalhado com resolução de 24 bits, e as amostras são processadas como números de ponto flutuante de 32 bits, o que praticamente elimina problemas de distorções causados por ajustes de ganho muito grandes ou pequenos que, no caso de inteiros, resultariam em perda de definição no sinal. Por outro lado, taxas de amostragem acima de 48KHz ainda não são tão comumente usadas, mas também oferecem ganhos de qualidade para o sinal, especialmente em relação à percepção espacial. O sistema *Pro Tools HD*, atualmente topo-de-linha da família *Pro Tools*, opera com taxas de amostragem de até 192KHz.

sobe para cerca de 7). Esse número, no entanto, é um limite superior do qual só seria possível nos aproximarmos caso o fluxo de dados a transmitir fosse contínuo, o que garantiria a utilização quase total da capacidade da rede. No entanto, como queremos garantir a operação com baixa latência, a transmissão dos dados está vinculada à recepção dos dados vindos do dispositivo de E/S de mídia e, portanto, não podemos usar *buffers* intermediários. Por causa disso, a transmissão e recepção de cada grupo de dados recebidos ou a ser enviados para o dispositivo de E/S de mídia está sujeita a todos os efeitos que podem reduzir a eficiência da rede, como:

- o tempo gasto na comunicação entre o computador e a placa de rede, tanto para a transmissão quanto para a recepção de dados (a velocidade dessa comunicação depende, entre outros fatores, da qualidade da placa de rede);
- o tempo gasto pela própria placa de rede tanto entre o recebimento dos dados vindos do computador e o início do envio dos dados quanto entre o recebimento dos dados e a geração da requisição de interrupção para que o computador leia os dados recebidos;
- o tempo gasto pelo sistema operacional para processar as interrupções provocadas pelo hardware de rede;
- a necessidade de sempre precisarmos levar em conta o tempo de pior caso para todos os possíveis atrasos envolvidos na comunicação para garantirmos uma operação confiável, já que não há *buffers* em uso na comunicação;
- de forma semelhante, no caso de redes como a Ethernet, que não oferece garantias de tempo real (diferentemente do padrão IEEE 1394, por exemplo), a inconstância no próprio tempo consumido pelo trânsito de dados pela rede; mesmo em uma rede dedicada para esse fim, pode haver colisões capazes de causar atrasos na transmissão de pacotes;
- a carga adicional gerada na rede correspondente aos protocolos de rede que viabilizam a comunicação, que reduz a largura de banda de fato utilizável.

Como é necessário enviar e receber os dados com uma grande freqüência (pois o objetivo é manter a baixa latência), todos os fatores elencados acima se tornam ainda mais significativos, já que cada um deles tem impacto sobre cada mensagem enviada ou recebida pela rede. Além disso, qualquer tentativa de aumentar o volume de dados trafegados pela rede se reverte numa redução no tempo disponível para o processamento remoto. É evidente que uma configuração como essa não é adequada.

Melhorias para a utilização da rede

Para procurarmos alternativas capazes de melhorar o sistema descrito acima, vale a pena encarar o processamento como sendo dividido em três subtarefas (sem perda de generalidade, vamos discutir o processamento com apenas duas máquinas):

1. a coleta dos dados vindos do dispositivo de E/S de mídia e a transferência dos dados da máquina de origem para a máquina remota (t_1);

2. o processamento remoto dos dados (t_2);

3. a transferência dos dados da máquina remota para a máquina de origem e o envio dos dados para o dispositivo de E/S de mídia (t_3).

Esse conjunto de três subtarefas corresponde a uma tarefa T completa que precisa ser executada a cada interrupção do dispositivo de E/S (ou seja, a cada chamada da função *callback*). Essa tarefa T pode ser processada eficientemente usando-se uma linha de produção, como discutido na Seção 4.4. Nessa linha de produção há apenas dois processadores em jogo: o processador responsável pela comunicação com o dispositivo de E/S e o processador responsável pelo processamento remoto. No entanto, das três subtarefas, duas (t_1 e t_3) consistem basicamente na transmissão dos dados via rede. Como vimos (também na Seção 4.4), cada uma das subtarefas t_{1-3} precisa ser executada em tempo inferior ao período da taxa de amostragem do sistema. Por outro lado, a soma do tempo de processamento dessas tarefas pode ser superior a esse período, e corresponde à latência de processamento do sistema. Assim, podemos sobrepor a comunicação de dados e o processamento remoto, oferecendo a possibilidade de utilizarmos melhor tanto a capacidade de processamento do processador remoto quanto a capacidade de transmissão de dados da rede. Um mecanismo semelhante, comumente chamado de sistema de janelas deslizantes, é utilizado em diversos protocolos de rede[5].

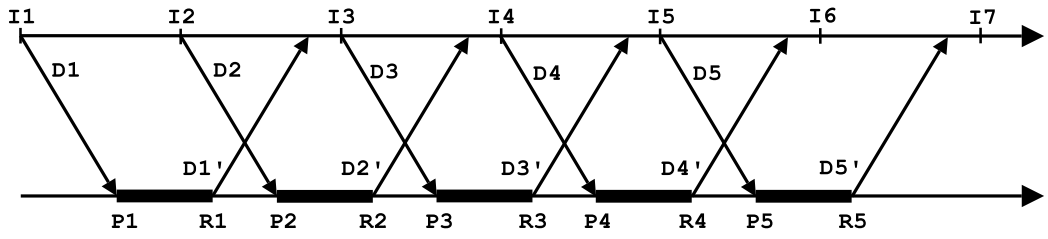
Chamemos de k o tempo para a execução da tarefa T (composta pelas subtarefas t_{1-3}) e δ o período da taxa de amostragem do dispositivo de E/S. Se $k \leq \delta$, T é executada por completo a cada interrupção do dispositivo de E/S (ou seja, a cada chamada *callback*), como visto acima. Se $k \leq 2\delta$, a cada chamada *callback*, a máquina onde essa chamada é executada envia dados para o processamento remoto e recebe dados processados pela máquina remota de volta; no entanto, os dados recebidos da máquina remota correspondem ao resultado do processamento dos dados enviados na chamada *callback* anterior. Ou seja, os dados são reproduzidos na saída com um atraso correspondente a exatamente um período δ a mais que o normal mas, por outro lado, o volume de dados que podem ser processados remotamente aumenta significativamente (Figura 5.2 – a), já que o tempo total para a execução de T é maior.

Se tomarmos novamente o exemplo do processamento de áudio usado acima, poderemos dispor de 80% do período para o processamento remoto dos dados e 120% do período para o tráfego de dados entre as máquinas, ou seja, 60% do período para a transferência de dados em cada direção. Nesse caso, o limite teórico de canais que podemos trafegar pela rede simultaneamente corresponde a $\frac{100000000}{1{,}666*96000*32} \cong 19$.

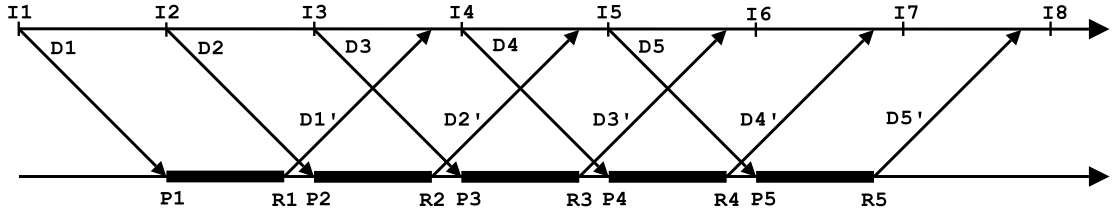
Se $k > 2\delta$, podemos utilizar um sistema onde a máquina de origem recebe, a cada chamada *callback*, os dados processados remotamente que foram enviados dois ciclos atrás. Com isso, podemos operar com o máximo de eficiência da rede e do processador remoto, mas por outro lado acrescentamos o equivalente a mais um período à latência total do processamento (Figura 5.2 – b).

[5]Por exemplo, o sistema de janelas deslizantes do protocolo TCP é descrito em Stevens, 1994, p. 280–284.

Mais uma vez retomando o exemplo acima, num cenário desse tipo podemos utilizar próximo a 100% do tempo de um período para o processamento remoto e ainda utilizar a rede em modo *full-duplex*, de forma a utilizar 200% do período para o tráfego de dados, 100% em cada direção. Assim, o limite teórico para o número de canais que podem ser enviados para processamento remoto é $\frac{100000000}{96000*32} \cong 32$.



(a) janela tamanho 1



(b) janela tamanho 2

Figura 5.2: A cada interrupção I_n, os dados capturados (D_n) são enviados para ser processados pela máquina remota (P_n); no final de uma iteração anterior (R'_{n-1}ou R'_{n-2}), os dados processados são devolvidos (D_{n-1}ou D_{n-2}) e enviados para o dispositivo de E/S de mídia antes da próxima interrupção (I_{n+1}).

É importante observar que não é possível estender esse raciocínio para além do tempo correspondente a dois períodos como descrito acima: com uma defasagem de dois períodos entre o envio e o recebimento dos dados pela máquina gerenciadora, a utilização do sistema é ótima, permitindo o uso de praticamente 100% da capacidade do processador remoto bem como de praticamente 100% da capacidade da rede em modo *full-duplex*. Só seria interessante implementar um mecanismo para inserir uma defasagem maior entre o envio e o recebimento de dados se houvesse mais máquinas na linha de produção. Como mencionado anteriormente, essa possibilidade foi deixada para pesquisas futuras.

5.1.5 Processamento isócrono

O mecanismo de janelas deslizantes descrito acima assume implicitamente que as requisições de interrupção do dispositivo de E/S de mídia são isócronas, ou seja, ocorrem em intervalos de tempo

regulares. De fato, só é possível reproduzir os dados correspondentes a um período em um outro período se os dois períodos tiverem a mesma duração. Esse mecanismo também não é capaz de lidar diretamente com sistemas que não sejam baseados em mídias contínuas, tais como MIDI.

Em sistemas de mídia contínua, o fluxo de dados a ser processado é, por definição, contínuo. A comunicação entre o dispositivo de E/S de mídia e o computador segmenta esse fluxo de dados com vistas à otimização no uso do processador, como já vimos (Seção 3.1.1). De maneira geral, essa segmentação ocorre, de fato, de maneira isócrona; no entanto, pode haver dispositivos que não seguem esse padrão. O sistema aqui descrito não é capaz de operar corretamente com dispositivos desse tipo. No entanto, se o tempo máximo possível entre duas interrupções for conhecido, o sistema pode manter um pequeno *buffer* de dados (correspondente a esse tempo máximo) e mantê-lo com mais ou menos dados de acordo com o tamanho do período utilizado a cada iteração, tendo como conseqüência o acréscimo na latência do sistema. Esse mecanismo, contudo, não foi implementado.

No entanto, mesmo em sistemas isócronos, pode surgir a necessidade de se enviar e receber *buffers* de tamanhos variáveis. Um exemplo tradicional consiste em fluxos de vídeo compactados com o padrão MPEG. Nesse formato, alguns quadros da imagem são representados por inteiro enquanto outros são representados por relações diferenciais em relação aos primeiros. Assim, diferentes quadros da imagem, correspondentes a quantidades de tempo iguais, consistem em volumes maiores ou menores de dados dependendo do tipo de quadro e também da taxa de compressão, que é variável no caso do padrão MPEG2. O mesmo se aplica a fluxos de áudio compactados com o padrão MPEG2, camada 3 (comumente chamados "MP3"). O sistema aqui descrito trata adequadamente fluxos de dados onde os *buffers* de dados correspondentes a segmentos temporais iguais têm tamanhos diferentes.

5.1.6 Integração com CORBA

Tarefas de pré-configuração do sistema (negociações para o estabelecimento de conexões entre as diversas máquinas, definição das funções a serem chamadas em cada período, alocação de memória etc.) não têm restrições de tempo real. Portanto, não é vantajoso procurar implementar mecanismos com características estritas de tempo para a pré-configuração do sistema, instanciação de objetos etc. Esses mecanismos podem, com grandes vantagens, ser delegados para uma camada baseada em CORBA (Tanenbaum e Steen, 2002, p. 494–525; Henning e Vinoski, 2001; Siegel 2000). De fato, o mecanismo descrito aqui pode também ser usado por objetos CORBA para a comunicação com baixa latência; aliado a sistemas como CORBA, um sistema desse tipo pode permitir a combinação das características mais interessantes dos sistemas de objetos distribuídos, como flexibilidade, heterogeneidade etc. com um bom desempenho para aplicações distribuídas com necessidades de processamento sincronizado em tempo real com baixa latência.

5.2 Implementação

De acordo com o que foi discutido até aqui, o middleware aqui descrito foi desenvolvido de maneira a permitir ao usuário registrar blocos de dados a serem enviados para processamento remoto, processados e recebidos do sistema remoto a cada iteração, possivelmente com o atraso correspondente ao mecanismo de janelas deslizantes descrito anteriormente. Para isso, o cliente:

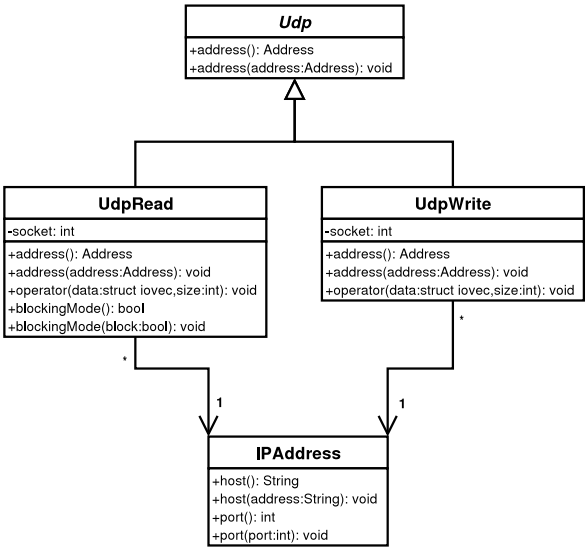
1. cria objetos da classe Master na máquina que possui o dispositivo de E/S de mídia e objetos da classe Slave nas outras máquinas do sistema;
2. registra *buffers* de dados que serão enviados e recebidos pelo sistema junto a esses objetos;
3. registra funções *callback* junto aos objetos da classe Slave que serão responsáveis pelo processamento dos *buffers* recebidos remotamente;
4. e, finalmente, executa chamadas periódicas à função process() dos objetos da classe Master, que promovem a troca de dados entre as máquinas e a conseqüente execução das funções *callback* remotas registradas.

O sistema utiliza o protocolo UDP para a comunicação; no entanto, é possível substituir esse protocolo por outros. Para isso, todas as funções especificamente relacionadas ao protocolo de rede foram delegadas a classes independentes que podem ser substituídas, de forma análoga ao padrão *Strategy* (Gamma et al., 1994, p. 315).

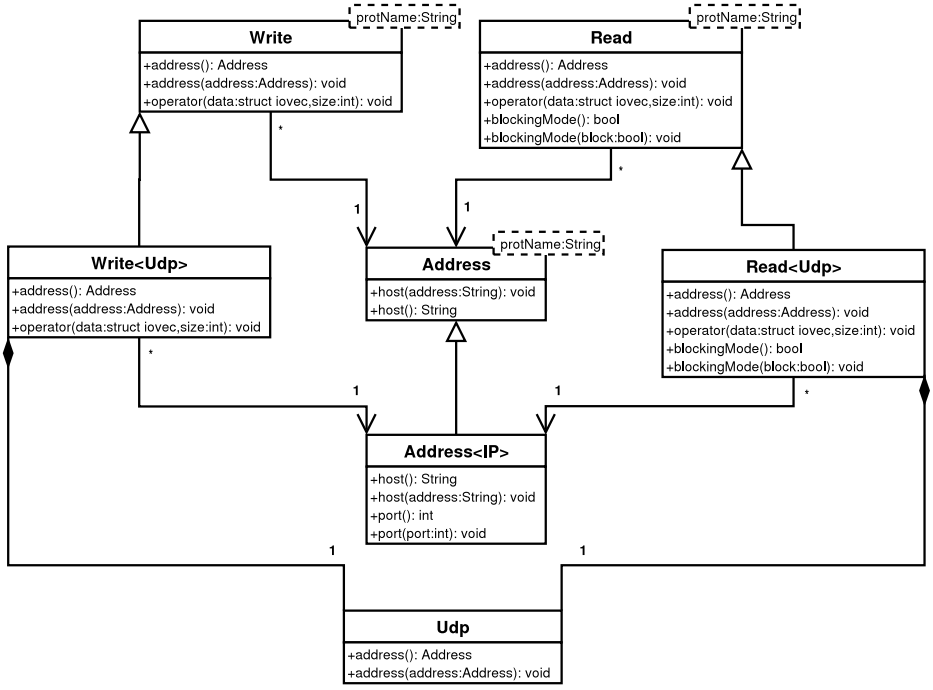
5.2.1 Abstração do protocolo de rede

Conforme discutido na Seção 5.1.3 (na página 60), o protocolo de comunicação que utilizamos na implementação do middleware aqui descrito é o protocolo UDP (Figura 5.3 – a). Como é nosso objetivo que esse protocolo possa ser facilmente substituído por outros (desde que atendam às características elencadas naquela seção), implementamos um conjunto de classes que abstraem o uso desse protocolo para uma interface básica, que pode ser reproduzida por outras classes voltadas para o funcionamento com outros protocolos. O uso de *templates* da linguagem C++ poderia tornar o uso de diferentes protocolos quase transparente para o usuário, embora a implementação atual ainda não faça uso de *templates* com esse objetivo (Figura 5.3 – b)[6].

[6] O uso de *templates* nessa situação é mais adequado que a mera utilização de mecanismos de herança. O uso de herança geralmente se justifica ou pela possibilidade de reutilização de código (quando, muitas vezes, pode-se usar com vantagens delegações ao invés de herança) ou pela possibilidade do uso de polimorfismo em tempo de execução. No entanto, nessa situação não há interesse em polimorfismo; de fato, embora a interface entre as classes que implementam o mecanismo de comunicação e a aplicação cliente deva ser fixa, as interfaces entre as diversas classes que implementam esse mecanismo de comunicação podem variar (por exemplo, as interfaces entre classes como EndereçoRemoto e Conexão podem variar em função do protocolo de rede a que se referem; não é possível definir *a priori* uma interface entre essas classes que sirva a todos eles). Como essas classes precisam ser manipuladas pela aplicação cliente, elas devem ser acessadas por referências às suas superclasses; portanto, internamente elas precisam realizar *downcasts* para poder acessar os métodos específicos a cada protocolo. Por outro lado, o uso de *templates* garante uma interface com a aplicação que é consistente, permite que haja reutilização de código, evita a necessidade de *downcasts* e oferece uma pequena vantagem em termos de desempenho em relação a funções virtuais.



(a) A implementação atual utiliza o protocolo UDP



(b) O código pode ser alterado para permitir a utilização de outros protocolos

Figura 5.3: Classes para a comunicação entre as máquinas

A primeira dessas classes é a classe IPAddress, que abstrai o conceito de um endereço de rede. De maneira geral, a aplicação cliente trata essa classe como um tipo opaco, apenas capaz de receber e fornecer representações textuais de endereços de rede; somente as outras classes que lidam diretamente com a comunicação de rede se utilizam dos métodos especificamente referentes ao protocolo TCP/IP, o que garante que todas essas classes podem ser substituídas conjuntamente sem que seja necessário promover outras alterações no sistema.

Lembremos que, conforme visto no Capítulo 3, a operação baseada em *callbacks* exige que um número qualquer de *buffers* de dados previamente alocados seja processado periodicamente. No caso do nosso sistema distribuído, é necessário que esses *buffers* sejam enviados e recebidos pela rede como uma mensagem única a cada iteração (Seção 5.1.3, página 60); isso significa que todos os *buffers* a serem enviados em uma iteração devem ser concatenados em um único pacote UDP. Felizmente, o Linux oferece as funções readv() e writev() (componentes opcionais de um sistema POSIX) que permitem, exatamente, que diversos *buffers* de dados sejam lidos ou escritos através de uma única chamada de sistema: o cliente cria um vetor de elementos do tipo struct iovec onde cada elemento descreve um *buffer* a ser enviado ou recebido por uma das funções readv() e writev(). Cada um desses elementos possui um apontador para um *buffer* de dados e um inteiro que armazena o número de bytes que compõem esse *buffer*, permitindo às funções readv() e writev() agrupar e separar os dados contidos em cada *buffer*.

Conseqüentemente, criamos as classes UdpRead e UdpWrite (derivadas de uma classe abstrata, Udp), que funcionam como *functors* (Josuttis, 1999, p. 124–130; Stroustrup, 1997, p. 514). Um *functor* de um desses tipos mantém um soquete aberto especificamente vinculado ao endereço recebido no construtor para o envio ou o recebimento de dados. A cada chamada ao método operator()(), um vetor de elementos do tipo struct iovec é recebido com os *buffers* de dados a ser enviados ou preenchidos. Internamente, operator()() chama readv() ou writev() sobre o *socket* correspondente. UdpRead pode ser utilizada para a leitura normal a partir da rede ou para a leitura sem bloqueio; nesse caso, UdpRead entra em um laço contínuo tentando realizar a leitura dos dados até um limite de tempo, o que é necessário para Master (como veremos abaixo).

5.2.2 Encapsulamento dos *buffers* de dados

O middleware aqui descrito permite a uma aplicação cliente fazer com que diversos *buffers* de dados sejam transportados para serem processados em sistemas remotos com garantias de tempo real e baixa latência. Portanto, antes de poder executar o processamento distribuído, o cliente do nosso middleware precisa registrar, junto ao middleware, quais os *buffers* de dados vão ser enviados e recebidos a cada iteração. Embora seja de se esperar que isso seja feito apenas durante a inicialização do sistema, já que essa tarefa não tem nenhum tipo de garantia de tempo real, é preciso que seja possível ao usuário modificar o conjunto de *buffers* registrados a qualquer momento. Assim, o middleware precisa ser capaz de manter, durante todo o tempo em que está sendo utilizado, uma associação entre cada *buffer* registrado e os seguintes atributos:

- endereço do *buffer*;

- tamanho do *buffer* em número de elementos (o que permite ao cliente tanto registrar um *buffer* sem convertê-lo em um apontador genérico quanto saber quantos elementos foram recebidos do sistema remoto);
- tamanho do *buffer* em número de bytes (necessário pela camada mais baixa de comunicação);
- tamanho de cada elemento do *buffer* (usado para converter entre os dois valores anteriores);
- direção dos dados do *buffer* (leitura, escrita ou ambos);
- necessidade ou não de correção de ordenação de bytes (dependente do tipo dos elementos do *buffer*).

O mecanismo para a manutenção dessa associação precisa levar em conta diversos aspectos, particularmente a necessidade de lidar corretamente com *buffers* de tamanho variável e com possíveis diferenças de ordenação de bytes (*endianness*) entre as máquinas.

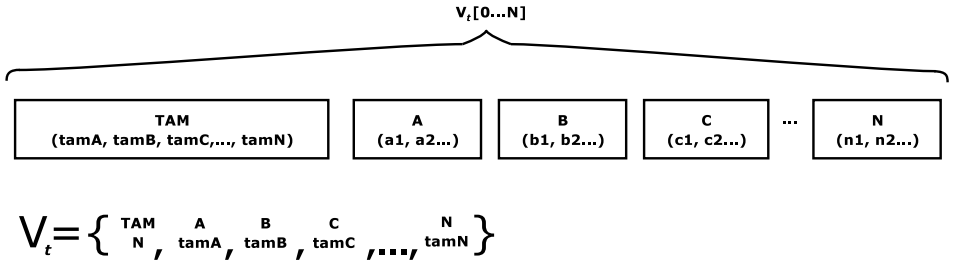
Buffers de tamanho variável

Como discutido na Seção 5.1.5, mesmo em sistemas isócronos pode ser necessário o envio e recebimento de *buffers* de dados de tamanhos variáveis. De fato, como veremos mais à frente (Seção 6.2.2), a aplicação desenvolvida neste trabalho para o processamento distribuído de módulos LADSPA necessita dessa funcionalidade.

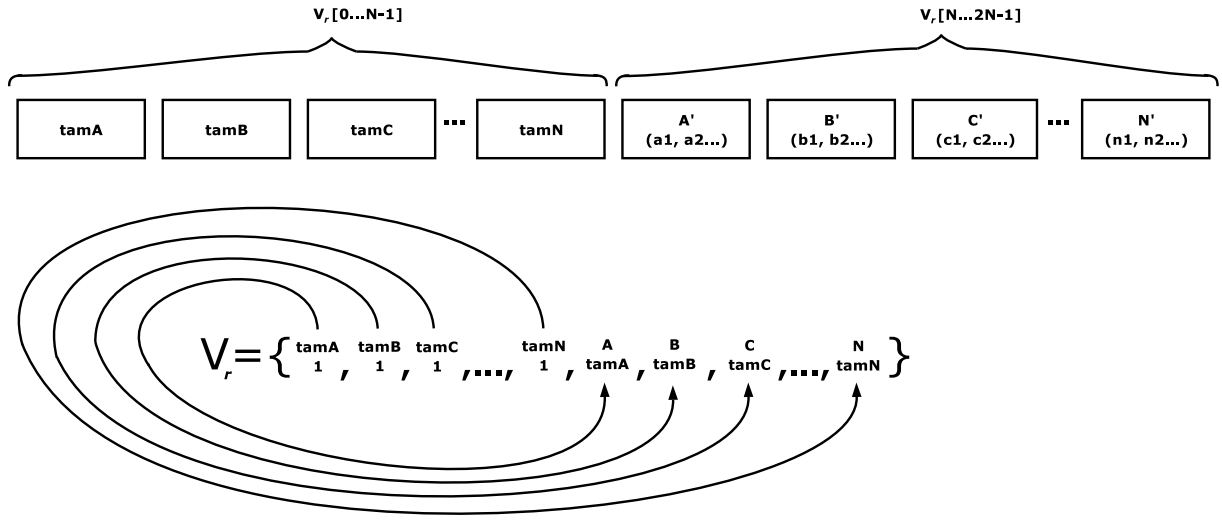
Normalmente, as funções readv() e writev() podem funcionar corretamente porque há *buffers* de dados correspondentes nos dois lados da comunicação com tamanhos iguais. writev() envia todos os dados dos *buffers* representados por um vetor com elementos do tipo struct iovec em seqüência para o sistema remoto; readv(), por sua vez, lê esses dados e preenche os *buffers* registrados em um vetor similar de acordo com os tamanhos definidos nesse vetor. Para isso, readv() simplesmente passa a escrever no *buffer* seguinte ao esgotar o tamanho de um *buffer*. No entanto, como nosso sistema precisa ser capaz de lidar com tamanhos de *buffers* variáveis, readv() não pode utilizar um vetor com elementos do tipo struct iovec construído de antemão para a leitura dos dados.

A solução para esse problema consiste em enviar para o sistema remoto, juntamente com o conteúdo dos *buffers*, seus tamanhos (Figura 5.4; como vimos, um struct iovec possui informações sobre o endereço de um *buffer* e seu tamanho; os tamanhos são representados na figura como os elementos inferiores dos vetores V_t e V_r). Em http://www.opengroup.org/onlinepubs/007904975/functions/readv.html, lê-se que "The readv() function shall always fill an area completely before proceeding to the next" (The Austin Group, 2003). De forma similar, a versão Linux da página de manual (*manpage*) da função readv() diz: "The readv() function reads [...] into the multiple buffers described by vector. [...] Buffers are processed in the order vector[0], vector[1],... vector[count]". Com base nisso, para enviar N *buffers*, criamos um vetor V_t com elementos do tipo struct iovec com $N + 1$ elementos (Figura 5.4 – a). O primeiro elemento desse vetor descreve um *buffer* que nada mais é que um outro vetor, este com elementos do tipo size_t, onde o elemento k é o tamanho do *buffer* k. No lado que recebe os dados

a serem distribuídos nos N *buffers* correspondentes, criamos um vetor V_r similar a V_t, mas com $2N$ elementos (Figura 5.4 – b). Os N primeiros elementos de V_r são tais que, para todo $k < N$, $V_r[k]$ descreve um *buffer* do tamanho de um único inteiro do tipo size_t que nada mais é que o membro iov_len de $V_r[k+N]$. Assim, ao ler o *buffer* correspondente a $V_r[k]$, readv() atualiza o membro iov_len de $V_r[k+N]$; conseqüentemente, para todo $k \geq N$, o elemento $V_r[k]$ (que corresponde a um dos *buffers* de dados que efetivamente nos interessam) já terá sido atualizado antes de ser processado, no momento do processamento do *buffer* correspondente ao elemento $V_r[k-N]$. Isso garante que tamanhos de *buffers* variáveis podem ser enviados e recebidos sem problemas.



(a) O tamanho de cada *buffer* é enviado juntamente com os *buffers*.



(b) Os tamanhos recebidos são inseridos no vetor usado por readv() durante a leitura dos dados da rede.

Figura 5.4: *Buffers* de dados de tamanhos variáveis são enviados através das funções readv() e writev().

Ordenação de bytes (*endianness*)

Para que o mecanismo descrito acima funcione adequadamente entre máquinas que utilizam ordenações de bytes diferentes, writev() precisa enviar o tamanho dos *buffers* já na ordenação de bytes nativa do sistema remoto. Isso significa que o middleware precisa ser capaz de saber qual é a ordenação de bytes do sistema remoto e, se necessário, corrigir a ordenação de bytes dos tamanhos dos *buffers* a ser enviados; não é possível simplesmente usar funções como htons() e ntohs().

Como esse mecanismo é necessário, optamos por não utilizar essas funções também no tratamento dos *buffers* de dados a ser enviados e recebidos; ao invés disso, o sistema sempre envia os dados de acordo com a sua ordenação de bytes nativa (exceto no caso do tamanho dos *buffers*, como mencionado acima). Ao receber dados de uma máquina remota, o sistema realiza a correção da ordenação de bytes se isso for necessário. Essa configuração pode minimizar o número de correções de ordenação em relação ao que seria necessário com o uso das funções htons() e ntohs() e similares.

Embora seria possível (e, talvez, mais coerente) realizar a mudança na ordenação de bytes antes do envio dos dados, e não após seu recebimento, isso teria o efeito de invalidar os dados contidos nos *buffers* locais. Como não podemos supor que a aplicação cliente do middleware não vá voltar a usar os dados desses *buffers*, preferimos garantir a consistência desses dados e promover a correção na ordenação de bytes quando da recepção dos dados.

5.2.3 Gerenciamento e agrupamento dos *buffers* de dados

As aplicações clientes do nosso middleware precisam ser capazes de registrar *buffers* de dados e seus respectivos tamanhos para serem processados remotamente a cada iteração. Embora seria possível exigir da aplicação o uso de vetores de elementos do tipo struct iovec para isso, diversos fatores sugerem uma abordagem diferente. Em particular, as dificuldades referentes à ordenação de bytes e a possibilidade de utilização de um mesmo *buffer* para o envio e o recebimento de dados aumentariam significamente a complexidade da aplicação cliente.

Para minimizar essas dificuldades, definimos duas classes responsáveis por gerir os *buffers* de dados registrados pela aplicação: DataBlock e DataBlockSet. DataBlock é um apontador inteligente para um desses *buffers*; DataBlockSet é simplesmente uma coleção de DataBlocks, e gerencia os vetores de elementos do tipo struct iovec usados para a comunicação remota. Essas duas classes são intimamente ligadas entre si, e cada uma delas mantém referências para a outra; como conseqüência, DataBlock só pode ser instanciada por uma instância de DataBlockSet. Apesar disso, o cliente não interage com DataBlockSet; ele solicita a criação de um DataBlock para uma instância de Master (descrita abaixo) e, a partir daí, interage apenas com esse DataBlock.

Essa organização permite que o cliente simplesmente solicite a Master a criação de DataBlocks e registre os *buffers* de dados a ser processados junto a esses DataBlocks. Se a aplicação mantiver referências a esses DataBlocks, ela pode também remover ou alterar esses *buffers* com a mesma facilidade, sem precisar realizar as diversas alterações necessárias nas estruturas de dados usadas para a comunicação remota. Apesar dessa simplicidade, é importante observar que essas classes apenas funcionam em uma

das extremidades da comunicação; é responsabilidade da aplicação cliente garantir que os *buffers* registrados em um dos extremos da comunicação em um DataBlockSet tenham seus correspondentes em um DataBlockSet no outro extremo.

Como os tamanhos dos *buffers* podem ser variáveis, DataBlockSet oferece os métodos prepareToSendData() e correctReceivedData(). prepareToSendData() atualiza o tamanho dos *buffers* de dados a serem enviados armazenados internamente de acordo com os dados fornecidos pela aplicação cliente; correctReceivedData() atualiza as variáveis do cliente correspondentes aos tamanhos dos *buffers* de dados de acordo com o que foi recebido. Além disso, esses métodos lidam com as possíveis diferenças na ordenação de bytes entre as máquinas da maneira descrita anteriormente. Esses métodos não são usados pelo cliente, sendo chamados pelas classes Master e Slave, como veremos adiante.

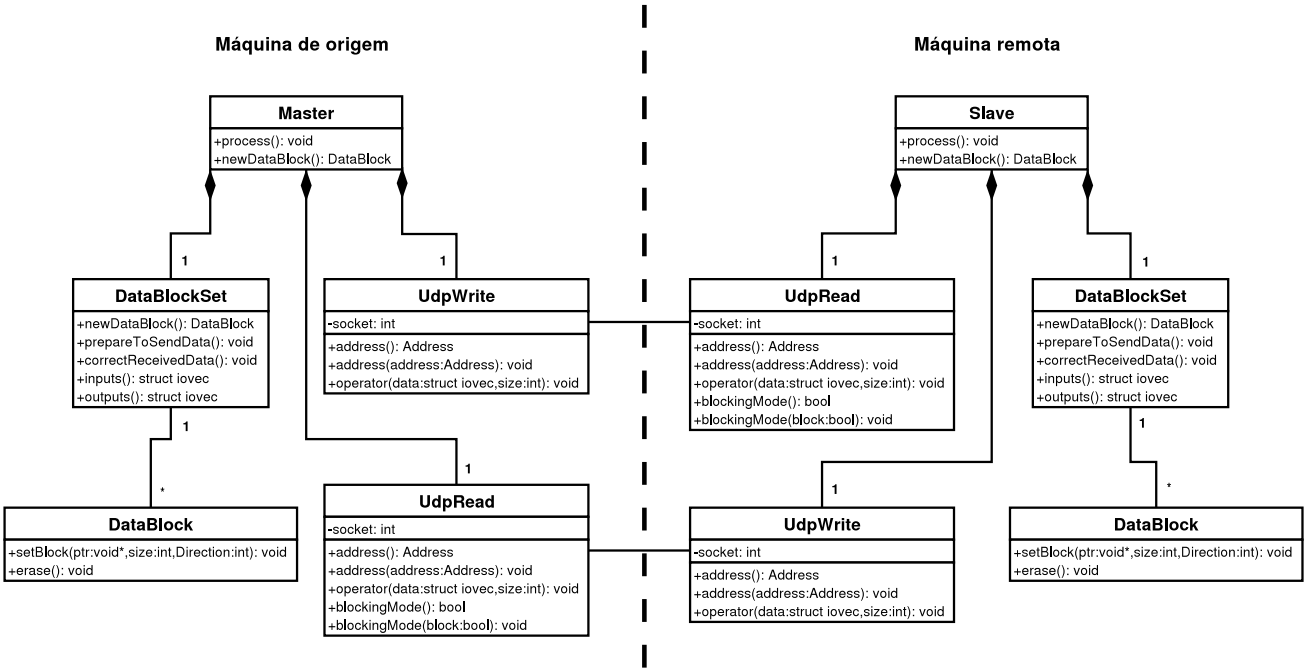
5.2.4 Processamento remoto

As classes descritas anteriormente realizam, de um lado, o envio e recebimento de dados via rede e, de outro, o gerenciamento dos blocos de dados a serem enviados e recebidos. No entanto, elas não contemplam operações como estabelecer a comunicação entre as máquinas, gerenciar o mecanismo de janelas deslizantes ou o registro e a execução das funções *callback*. Essas tarefas são levadas a cabo pelas classes Master (utilizada no sistema que interage com o dispositivo de E/S de mídia) e Slave (utilizada nas máquinas responsáveis pelo processamento remoto) (Figura 5.5).

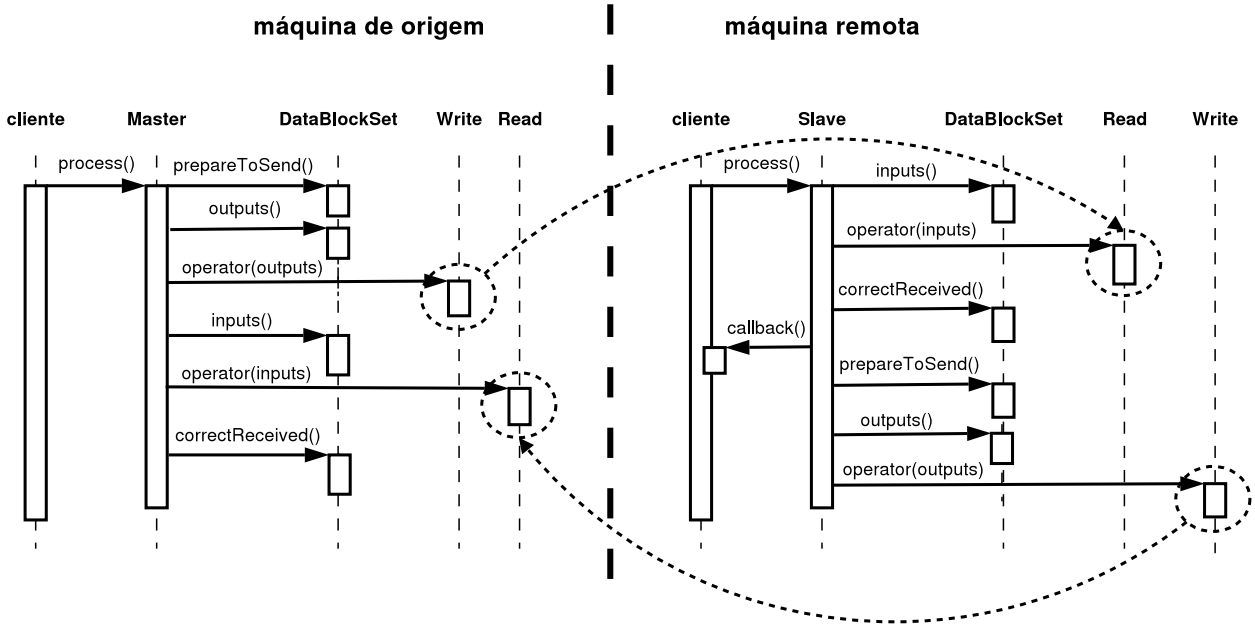
Master e Slave têm papéis simétricos: Master interage com a aplicação na máquina que possui o dispositivo de E/S de mídia; Slave, por sua vez, realiza um laço infinito esperando para processar dados enviados periodicamente por Master. Master oferece um método (process()) que é usado pela aplicação para solicitar o processamento remoto. Quando process() é chamada, Master envia os dados registrados pela aplicação para Slave. Cada vez que recebe um novo conjunto de dados enviados por Master, Slave chama uma função *callback* registrada pela aplicação para processá-los e envia os resultados de volta. Ao receber os dados processados, Master atualiza os *buffers* de dados registrados pela aplicação e finaliza o método process().

Master e Slave possuem cada uma um DataBlockSet como membro e recebem como parâmetros de seus construtores objetos das classes UdpRead e UdpWrite que devem ser utilizados para a comunicação. Além dos *buffers* de dados registrados pelo cliente, Master acrescenta um elemento interno ao DataBlockSet: um contador que é utilizado para garantir que pacotes UDP recebidos fora de ordem não coloquem o sistema em um estado incoerente e para permitir o gerenciamento do tamanho da janela do sistema de janelas deslizantes.

Como Master é executada dentro da função *callback* da aplicação responsável por interagir com o dispositivo de E/S, a comunicação via rede não pode provocar o bloqueio da aplicação (como discutido na Seção 3.3); assim, a instância de UdpRead usada por Master deve ser configurada para operação sem bloqueio. Por outro lado, a instância de UdpRead usada por Slave bloqueia esperando os dados da rede; como já vimos, essa característica garante a unidade temporal entre as interrupções do dispositivo de E/S de mídia e o processamento realizado remotamente pela função *callback*. Master e Slave também



(a) Diagrama de classes



(b) Diagrama de interação

Figura 5.5: As classes do middleware

são responsáveis por realizar as chamadas aos métodos prepareToSendData() e correctReceivedData() de DataBlockSet quando da comunicação de dados.

O sistema exige que a aplicação cliente seja capaz de identificar os endereços de rede das máquinas envolvidas no processamento. Seria possível procurar implementar um mecanismo para que Master e Slave fossem responsáveis por essa tarefa; por exemplo, Slave poderia registrar-se num servidor de nomes CORBA. No entanto, esse mecanismo dificilmente poderia ser generalizado o suficiente, pois o middleware pode ser usado em diversas aplicações incompatíveis entre si, dificultando a tarefa de identificar qual dos objetos do tipo Slave registrados no servidor de nomes seria adequado para uma determinada aplicação. Esse mecanismo poderia, inclusive, não ser adequado para algumas aplicações. Assim, como veremos mais adiante, optamos por implementar um mecanismo desse tipo como parte da aplicação para o processamento distribuído de áudio que pode ser a base para o desenvolvimento de outras aplicações similares sem, contudo, vincular o middleware diretamente a esse mecanismo.

Capítulo 6

A aplicação DLADSPA

O objetivo central deste trabalho é implementar um mecanismo para a distribuição do processamento de áudio em tempo real com baixa latência em uma rede de computadores com baixo custo. Para atingir esse objetivo, desenvolvemos o middleware descrito anteriormente, que apresenta uma interface para programas clientes baseada nas classes Master e Slave. No entanto, as classes Master e Slave são genéricas; é preciso escrever classes adaptadoras (Gamma et al., 1994, p. 139) para utilizá-las em contextos específicos, em particular em aplicações legadas. Neste capítulo, descrevemos os mecanismos utilizados no desenvolvimento da aplicação DLADSPA (Distributed LADSPA), baseada nesse middleware, para promover o processamento distribuído de módulos compatíveis com a especificação LADSPA ([LADSPA]), bem como sua implementação. Esse mecanismo tanto encapsula o middleware sob uma interface compatível com a especificação LADSPA quanto se utiliza de módulos LADSPA já existentes para realizar o processamento de áudio, o que permite que aplicações legadas possam se utilizar de módulos LADSPA quaisquer em um sistema distribuído sem alterações.

6.1 Distribuição de módulos LADSPA

Seria possível implementar diversos mecanismos para atingir o objetivo de possibilitar o processamento distribuído de áudio; no entanto, como discutido na Introdução (Seção 1.3.4), é interessante garantir a compatibilidade com aplicações legadas. Dentre os diversos mecanismos possíveis para isso, a utilização da interface definida pela especificação LADSPA e de módulos LADSPA já existentes nos pareceu a solução mais simples e, ao mesmo tempo, versátil.

6.1.1 Compatibilidade com aplicações legadas

Uma maneira de viabilizar o uso de sistemas distribuídos para o processamento de áudio é oferecer ao desenvolvedor bibliotecas ou componentes que o auxiliem na tarefa de criar aplicações capazes de se beneficiar de tais sistemas. De maneira geral, o middleware descrito anteriormente poderia ser usado para o desenvolvimento de novas aplicações com tais características. O desenvolvedor, porém, precisaria criar, para a sua aplicação, um padrão específico de comunicação com esse middleware. Essa tarefa

tornaria o desenvolvimento de cada aplicação com suporte a processamento distribuído uma tarefa relativamente trabalhosa.

No entanto, existe, em ambiente Linux, o sistema gstreamer ([GSTREAMER]), que é um arcabouço para o desenvolvimento de aplicações multimídia. Esse sistema consiste em diversos módulos capazes de realizar as mais variadas operações relacionadas a multimídia (ler e decodificar arquivos de áudio ou vídeo, promover transformações no sinal etc.) e em uma infra-estrutura para a interconexão desses módulos. De maneira geral, uma aplicação baseada no sistema gstreamer consiste em uma interface de usuário e uma série de módulos interconectados de maneira a executar uma tarefa específica. O desenvolvimento de um módulo gstreamer para o processamento distribuído (ou seja, um módulo capaz de delegar parte do processamento para uma máquina remota) abriria as portas para a criação de novas aplicações com suporte ao processamento distribuído com esforço mínimo por parte do desenvolvedor.

Por outro lado, já existe um grande e crescente número de aplicações para o processamento de áudio disponíveis, como vimos na Seção 1.3.4. Um mecanismo para o processamento distribuído de áudio compatível com tais aplicações legadas teria uma abrangência muito maior que um sistema dependente de modificações no seu código ou do desenvolvimento de novas aplicações, projetadas especificamente para se beneficiarem de sistemas distribuídos. Isso sugere que as interfaces entre essas aplicações e outras partes do sistema podem oferecer oportunidades para a implementação de mecanismos de distribuição compatíveis com todas elas.

Uma opção seria utilizar a interface da aplicação com o controlador do dispositivo de E/S de áudio como o meio para a distribuição: a aplicação se comunicaria com o sistema remoto através da interface normal de E/S para a qual foi projetada. Por exemplo, o sistema JACK (Seção 3.3.1) poderia permitir que uma aplicação completa fosse executada em uma máquina enquanto outra fosse executada em uma máquina diferente, mas com a possibilidade de troca de dados entre elas e com um único dispositivo de E/S de áudio ao qual as duas aplicações estariam vinculadas (a estrutura modular do servidor jackd possibilitaria essa alteração sem grandes dificuldades). De fato, o sistema VST System Link (veja a discussão à página 15), da empresa Steinberg, funciona de maneira mais ou menos similar[1].

Tal sistema, no entanto, ofereceria uma interface um tanto complexa com o usuário, já que aplicações inteiras, com suas interfaces de usuário, estariam em execução simultaneamente em máquinas diferentes (no ambiente Linux, o uso do sistema de janelas X — [XORG], que permite a visualização remota de programas, poderia minimizar os inconvenientes dessa organização). Mais importante que isso, no entanto, é o fato de que, em geral, cada aplicação não realiza apenas uma operação complexa sobre um sinal; pelo contrário, comumente cada aplicação processa diferentes fluxos de mídia com diferentes operações simultaneamente. A granularidade oferecida por um sistema capaz de interconectar aplicações remotas inteiras é muito grossa.

Uma outra interface comum entre partes do sistema é a interface dos módulos LADSPA. De fato, nas aplicações que utilizam módulos LADSPA, é comum que a maior parte do processamento seja efetivamente executada por uma grande quantidade desses módulos carregados simultaneamente pela aplicação; a aplicação apenas oferece um ambiente para a execução desses módulos (contexto de exe-

[1] Embora um pouco diferentemente, pois há mais de um dispositivo de E/S envolvido.

cução, gerenciamento de memória etc.), além da interface com o usuário e os mecanismos para E/S. Assim, possibilitar à aplicação carregar módulos LADSPA que serão executados em máquinas remotas pode permitir-lhe carregar um número muito maior de módulos e executá-los sobre um volume muito maior de fluxos de áudio do que se todos os módulos fossem executados em um único processador. Além disso, a grande maioria das aplicações compatíveis com o sistema JACK também é compatível com a especificação LADSPA, e o sistema gstreamer possui um módulo que permite o uso de módulos LADSPA, o que significa que o uso da especificação LADSPA para a implementação do sistema de distribuição garante a compatibilidade com um grande número de aplicações. Por essas razões, elegemos a interface do padrão LADSPA para implementar a ponte entre as aplicações que queremos distribuir e o nosso sistema. Para que isso funcione adequadamente, a aplicação carrega um módulo LADSPA que, na verdade, opera como um *proxy* (Gamma et al., 1994, p. 207) que delega o processamento real para uma máquina remota; nessa máquina remota, o processamento é efetivamente levado a cabo por módulos LADSPA ordinários. O nome desse sistema é DLADSPA: ***D****istributed* ***LADSPA***.

6.1.2 Conjuntos de módulos

Como discutido anteriormente, nosso objetivo é promover o processamento remoto de fluxos de dados de mídias contínuas. Também vimos que o problema de desempenho no processamento de multimídia em tempo real é, em geral, decorrente da necessidade de se processar múltiplas transformações em múltiplos fluxos de dados. Assim, para solucionar esse problema eficientemente, um sistema para o processamento de multimídia em tempo real precisa ser capaz de delegar todo o processamento referente a um determinado fluxo de dados para uma máquina remota. Isso equivale a dizer, no nosso caso, que precisamos de um sistema que permita que os processamentos de vários módulos LADSPA sejam aplicados remotamente sobre um determinado fluxo de áudio. Esse mecanismo deve permitir que a aplicação seja capaz de alterar os parâmetros de controle desses diversos módulos LADSPA remotos, o que significa que esses parâmetros devem ser enviados e recebidos como os outros *buffers* de dados pelo middleware.

Seria possível implementar um mecanismo para realizar o processamento remoto referente a um único módulo LADSPA e utilizar esse mecanismo em várias instâncias diferentes para levar a cabo o processamento de mais de um módulo. No entanto, como vimos na Seção 5.1.4, o processamento no nosso sistema envolve um aumento de latência no processamento, provocado pelo mecanismo de janelas deslizantes; o acúmulo de atrasos adicionais decorrente dessa solução, portanto, a torna inadequada.

No sistema DLADSPA, optamos por implementar um mecanismo para o processamento remoto de apenas um módulo LADSPA e, ao mesmo tempo, implementamos um outro mecanismo para que um conjunto de diversos módulos LADSPA possa ser encapsulado e acessado através da interface padrão de um módulo LADSPA comum. Isso nos permite, por um lado, apresentar o descritor desse módulo composto para a aplicação e, por outro, instanciar apenas um módulo LADSPA no sistema remoto. Toda a complexidade envolvida na instanciação e interconexão dos vários módulos remotos fica limitada dentro de um conjunto de classes bem definido, simplificando o mecanismo de comunicação remota.

6.1.3 Editores

A especificação LADSPA prevê que uma aplicação realize o levantamento dos módulos LADSPA disponíveis apenas uma vez (geralmente durante sua inicialização); nosso sistema, portanto, precisa ser capaz de fornecer à aplicação uma coleção de descritores de módulos quando consultada de acordo com a interface padrão LADSPA. Isso significa que não é possível definir um conjunto de módulos interconectados de maneira *ad hoc* durante a execução da aplicação: os conjuntos de módulos precisam ser definidos de antemão. É necessário, portanto, que o usuário defina quais os conjuntos de módulos devem ser passíveis de ser executados remotamente, o que significa que é necessário que exista algum tipo de editor que permita ao usuário definir esses conjuntos de módulos e gravar suas definições de maneira que elas estejam disponíveis nas próximas execuções das aplicações que vão se utilizar do sistema.

Ao invés de desenvolver um editor desse tipo, optamos por utilizar um editor já existente, jack-rack ([JACKRACK] — Figura 6.1), que permite ao usuário definir cadeias de módulos LADSPA a serem aplicadas sobre um ou mais fluxos de áudio. Esse aplicativo oferece praticamente toda a funcionalidade necessária e, por gravar as configurações definidas pelo usuário em um arquivo XML, permite facilmente a implementação de um mecanismo para a interpretação desse arquivo.

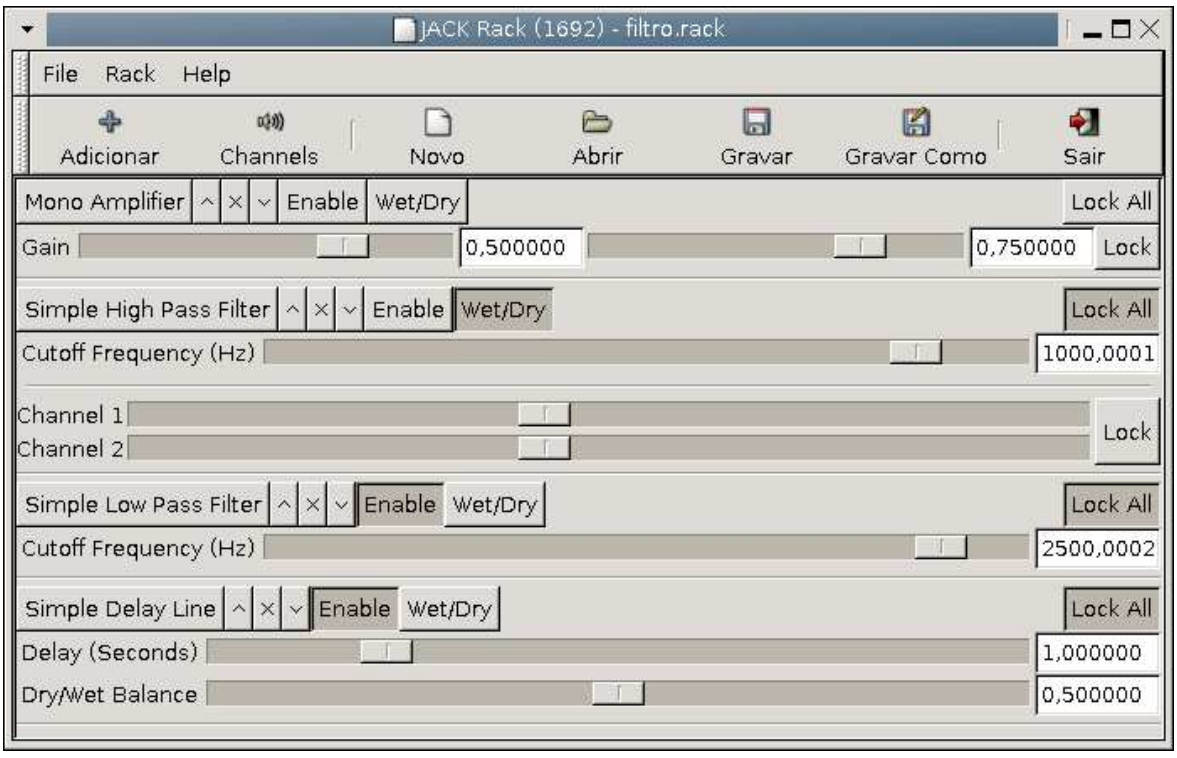


Figura 6.1: O processador de efeitos jack-rack.

6.1.4 Servidores de módulos LADSPA

Como mencionado anteriormente, o sistema DLADSPA precisa ser capaz de instanciar e remover módulos LADSPA remotos e estabelecer um mecanismo de comunicação que lhe permita tanto solicitar a instanciação e remoção desses módulos quanto configurar o middleware para a troca de dados entre as máquinas em tempo real. É necessário, portanto, que exista um programa constantemente em execução nas máquinas destinadas a abrigar os módulos remotos capaz de receber mensagens de outras máquinas e de criar e destruir os módulos de acordo com as solicitações recebidas. Esse programa é um servidor que funciona como uma fábrica de módulos acessíveis remotamente.

Uma maneira de viabilizar a comunicação entre as diversas máquinas do sistema é utilizar o sistema CORBA (Tanenbaum e Steen, 2002, p. 494–525; Henning e Vinoski, 2001; Siegel, 2000): através de chamadas remotas de métodos definidas por uma interface IDL simples, é possível transmitir todos os dados necessários ao sistema. Como pode haver um grande número de máquinas para o processamento remoto disponíveis em uma rede, é também necessário que haja um mecanismo para que a aplicação seja capaz de escolher qual máquina deverá abrigar o processamento de um determinado fluxo de áudio. O serviço de nomes CORBA (OMG, 1998) oferece uma maneira simples de realizar isso: cada servidor de módulos disponível se registra junto a um servidor de nomes CORBA, permitindo ao sistema encontrar facilmente todas as máquinas disponíveis para o processamento remoto e oferecer uma lista de opções para o usuário. Uma outra abordagem, mais sofisticada, seria usar o serviço de negociações (*trader*) CORBA (OMG, 1998) para permitir ao sistema escolher automaticamente um sistema remoto adequado para o processamento dentre todos os disponíveis. A utilização de CORBA, portanto, oferece vantagens para a implementação do sistema.

Como veremos mais adiante, implementamos um servidor de módulos com esse perfil. Embora esse servidor (bem como o gerenciador, a ser discutido mais adiante, e outras classes auxiliares) tenha sido criado especificamente para o sistema DLADSPA, é de se esperar que outras aplicações possivelmente sejam baseadas em paradigmas semelhantes ao da especificação LADSPA. Assim, procurou-se garantir que seu código seja genérico o suficiente para que possa ser facilmente adaptado para outros sistemas.

6.2 Implementação

Como dito anteriormente, foi preciso desenvolver um conjunto de classes adaptadoras para permitir que aplicações legadas interajam com nosso middleware de maneira a realizar o processamento distribuído de áudio. As duas classes mais importantes desse sistema, que realizam a conexão entre a especificação LADSPA e as classes Master e Slave, são Proxy e Container. Proxy é carregada pela aplicação como um módulo LADSPA qualquer, mas na realidade redireciona todo o processamento solicitado pela aplicação para uma máquina remota. Container instancia remotamente o módulo LADSPA real solicitado pela aplicação (que é, de fato, um conjunto de módulos encapsulados) e aloca a memória necessária para os *buffers* de dados que serão usados para a comunicação com o sistema remoto. Além disso, Container também solicita ao módulo LADSPA local que processe os dados quando são recebidos.

Para configurar o sistema para a operação em tempo real, Proxy envia mensagens através de um soquete TCP (Stevens, 1994, p. 223–227) para uma aplicação, o gerenciador; este, por sua vez, realiza a comunicação (via CORBA) com o servidor de módulos LADSPA remoto, que cria uma nova instância de Container.

6.2.1 Encapsulamento de módulos LADSPA

O processamento de fluxos de áudio por múltiplos módulos LADSPA depende de um mecanismo para a definição de conjuntos de módulos LADSPA a serem instanciados bem como de descritores LADSPA correspondentes ao conjunto. Ao invés de criar novas interfaces para lidar com essas tarefas, preferimos criar mecanismos para a criação de módulos LADSPA compostos por conjuntos de outros módulos; assim, a única interface utilizada entre as diversas partes do sistema é aquela definida pela especificação LADSPA. Como veremos, esse procedimento foi realizado em dois níveis, de maneira similar ao sugerido pelo padrão *Composite* (Gamma et al., 1994, p. 163).

Processamento de mais de um fluxo de áudio

Embora em geral tenhamos nos referido ao processamento remoto de fluxos de áudio independentes, não é incomum que dois ou mais fluxos de áudio estejam intimamente ligados e, portanto, sejam submetidos às mesmas transformações. Por exemplo, dois fluxos de áudio correspondentes a um sinal estereofônico geralmente sofrem processamento idêntico; há alguns algoritmos, como alguns tipos de reverberação, que efetivamente operam sobre sinais estereofônicos. O mesmo pode valer para um número maior de fluxos de áudio em outras situações.

Assim, é útil que seja possível ao usuário do nosso sistema definir um número arbitrário de canais de áudio que deverão ser processados remotamente em uma determinada máquina. No entanto, os módulos LADSPA já existentes definem exatamente quantos canais de áudio são capazes de processar; fica normalmente a cargo das aplicações compatíveis com LADSPA criar quantas instâncias forem necessárias de cada módulo para o processamento de um determinado número de canais de áudio. No nosso caso, no entanto, permitir à aplicação que instancie múltiplas cópias de um módulo a ser executado remotamente não é uma solução eficiente, pois isso envolveria um número maior de mensagens através da rede, com o correspondente custo em termos de mudanças de contexto nas máquinas remotas; seria também necessário exigir do usuário que configurasse todas essas instanciações remotas.

Para contornar esses problemas, criamos um par de classes (MultiChannelPlugin e MultiChannelPluginType) que permitem que um determinado módulo seja instanciado quantas vezes for necessário para que seja possível processar o número de canais desejado. Esse conjunto de módulos iguais, que operam paralelamente em diferentes fluxos de áudio, é ele próprio acessível através da interface LADSPA como um único módulo com vários canais de entrada e saída. MultiChannelPluginType é responsável por criar o LADSPA_Descriptor correspondente ao conjunto de módulos de acordo com o número de canais desejados e definir as relações entre as portas internas de cada módulo e as portas que são apresentadas para a aplicação cliente. Portanto, MultiChannelPluginType oferece uma representação de um módulo

composto. Uma instância de um desses módulos é uma instância de MultiChannelPlugin, que instancia os módulos internos e efetivamente realiza as chamadas de função desses módulos quando assim solicitado pela aplicação.

Para simplificar o uso desses módulos compostos, o sistema possibilita que portas de controle iguais de diferentes instâncias dos módulos internos apareçam para o cliente LADSPA como uma única porta de controle, permitindo que essa porta seja controlada pelo cliente simultaneamente em todas as instâncias. Por exemplo, um módulo mono capaz de alterar o volume do sinal que passa por ele pode ser encapsulado em um módulo composto por quatro instâncias para processar quatro canais de áudio simultaneamente ao invés de apenas um. O sistema permite que esse módulo composto possua quatro portas de controle, para o controle de volume de cada um dos canais de áudio, ou apenas uma, que corresponde internamente às quatro portas de controle de volume dos quatro módulos. O sistema também permite que algumas portas de controle dos módulos encapsulados simplesmente não apareçam para o cliente LADSPA; nesse caso, um valor *default*, definido quando da especificação das características do módulo composto, será usado. Infelizmente, o editor que usamos, jack-rack, não permite ao usuário definir se uma porta deve ser ou não habilitada; todas as portas de controle criadas por ele estão habilitadas.

O sistema também pode criar portas de controle adicionais ("*wetness ports*"), usadas para controlar a intensidade relativa entre o sinal original recebido pelo módulo e o sinal depois de processado. Essas portas são tratadas de forma similar às outras portas de controle e também podem ser travadas (fazendo com que uma única porta de controle afete o processamento de todos os módulos internos) e desabilitadas (fazendo com que um valor *default* seja utilizado). É importante observar que todo o código para o gerenciamento das "*wetness ports*" está implementado, exceto pelo código que efetivamente realiza o controle de equilíbrio entre os dois sinais; na implementação atual, o sinal de saída sempre é 100% processado.

Encadeamento de módulos

Nosso objetivo é poder realizar o processamento de diversos módulos LADSPA sobre um ou mais fluxos de áudio. Assim sendo, é necessário que possamos instanciar e interconectar um número arbitrário de módulos LADSPA, iguais ou diferentes. Seria interessante que essas interconexões pudessem formar grafos quaisquer definidos pelo usuário; no entanto, isso teria gerado uma grande complexidade adicional ao sistema, até porque não encontramos nenhum bom programa que permita ao usuário gerar um grafo desse tipo de forma simples e que pudesse ser reaproveitado por nós.

A grande maioria das aplicações de áudio, no entanto, segue um paradigma similar ao de uma mesa de mixagem, onde muito raramente há interconexões de processadores de efeito que não consistam simplesmente no encadeamento em série de processadores. Como existem aplicações capazes de editar cadeias desse tipo e essa organização é mais simples de ser implementada, optamos por apenas oferecer suporte a cadeias de módulos LADSPA. No entanto, graças (mais uma vez) ao uso da interface padronizada LADSPA para definir como é a interação entre a aplicação e o módulo composto de outros módulos, se um mecanismo para a criação de conjuntos de módulos compostos por grafos de módulos

for desenvolvido, ele poderá substituir facilmente o mecanismo atual, que permite apenas a criação de cadeias de módulos.

O mecanismo de encadeamento de módulos de processamento é implementado pelas classes PluginChain e PluginChainType, que dividem as tarefas entre si de forma similar a MultiChannelPlugin e MultiChannelPluginType. O encapsulamento dos módulos possibilita apresentar para a aplicação apenas as portas de entrada de áudio do primeiro módulo da cadeia e as portas de saída de áudio do último módulo da cadeia, além das portas de controle. Internamente, as saídas de áudio de cada módulo intermediário estão ligadas às entradas de áudio de próximo módulo da cadeia através de *buffers* temporários mantidos pela instância de PluginChain.

Uma característica desejável para os módulos LADSPA é que eles permitam o processamento com um único *buffer* de dados utilizado simultaneamente como entrada e saída, pois isso pode aumentar significativamente a taxa de sucessos no *cache* de memória e, portanto, garantir uma melhor utilização do processador. Criar *buffers* separados para cada interconexão entre módulos, portanto, poderia ter um impacto significativo no desempenho do sistema. Assim, ao invés de criar um *buffer* de dados para cada interconexão interna entre dois módulos, optamos por implementar um mecanismo que minimiza o número de *buffers* diferentes necessários para o processamento. Esse mecanismo precisa levar em conta o fato que alguns módulos LADSPA não são capazes de compartilhar um mesmo *buffer* de dados para a entrada e saída de dados, o que é indicado se seu descritor LADSPA contém a propriedade LADSPA_PROPERTY_INPLACE_BROKEN. Nesses casos, *buffers* adicionais devem ser usados.

6.2.2 Interação com aplicações LADSPA

Para que aplicações legadas sejam capazes de interagir com nosso sistema, optamos por utilizar a interface padrão definida na especificação LADSPA para apresentar para a aplicação os serviços do nosso middleware. As aplicações legadas, portanto, precisam ser capazes de carregar bibliotecas de módulos compatíveis com LADSPA que, efetivamente, se encarregam de promover o processamento remoto dos dados. Para viabilizar esse mecanismo, criamos uma pequena biblioteca (remote_plugins.so) de acordo com a especificação LADSPA, que deve ser carregada dinamicamente pela aplicação (através da função dlopen()). Ao ser carregada, essa biblioteca lê arquivos XML criados pelo usuário com o aplicativo jack-rack que definem módulos remotos que podem ser criados e cria descritores LADSPA correspondentes a cada um deles.

Quando a aplicação solicita a instanciação de um desses módulos, uma instância da classe Proxy é criada. Quando a aplicação executa chamadas ao método connect_port() de Proxy, Proxy simplesmente registra cada *buffer* recebido junto a uma instância de Master. Proxy também é responsável por solicitar a instanciação remota do módulo que vai efetivamente realizar o processamento, como veremos mais adiante. Finalmente, Proxy registra dois blocos adicionais junto a Master: functionToCall e sampleCount. functionToCall permite a Proxy solicitar a execução remota de mais de uma função, como activate()/deactivate(), run()/run_adding() etc. Cada vez que um método de Proxy que envolve proces-

samento remoto é chamado (tipicamente, o método run()), Proxy simplesmente define functionToCall e executa o método process() de Master.

sampleCount é necessário porque a especificação LADSPA permite que a aplicação que faz uso de um módulo LADSPA realize a chamada ao módulo com tamanhos de *buffers* de dados diferentes a cada iteração. Esse tipo de uso não é compatível com o sistema DLADSPA; como discutimos anteriormente, o sistema assume que as chamadas *callback* ocorrem a intervalos de tempo isócronos. No entanto, uma conseqüência dessa possibilidade é que a especificação LADSPA não define um mecanismo para que a aplicação informe ao módulo qual o tamanho dos *buffers* será utilizado antes da primeira iteração; ao invés disso, a função run() recebe sampleCount como parâmetro. Assim, é necessário ser possível fornecer essa informação ao módulo remoto durante a chamada à primeira iteração, o que é possível graças ao registro de sampleCount junto a Master. Essa necessidade abre a possibilidade de enviarmos essa informação em todas as iterações, o que pode vir a ser utilizado no futuro por um mecanismo que se adapte a essas variações no tamanho do *buffer*.

6.2.3 Módulos LADSPA remotos

Para cada instância de Proxy, é preciso que exista um módulo LADSPA real em execução em uma máquina remota. Assim, é preciso estabelecer mecanismos para a instanciação e gerenciamento desse módulo nessa máquina remota e para a criação dos *buffers* de dados que serão enviados e recebidos por ela. Esse mecanismo foi implementado pela classe Container, que tem funcionamento similar e complementar a Proxy. Essa classe instancia o módulo solicitado por Proxy (uma instância de PluginChain) e cria *buffers* de dados locais correspondentes a cada uma das portas do módulo instanciado. Esses *buffers* são conectados às portas do módulo e registrados junto a uma instância de Slave. Quando seu método process() é chamado, Container verifica o valor de functionToCall e executa a função correspondente do módulo LADSPA (a instância de PluginChain) pelo qual é responsável (tipicamente, run()).

Ao criar as instâncias de Slave e de Container, o servidor remoto registra a função process() de Container como a função *callback* a ser chamada por Slave quando novos dados são recebidos. A seguir, inicia um laço infinito em que executa o método process() de Slave. Com isso, sempre que a máquina que interage com o dispositivo de E/S de áudio envia uma mensagem, o servidor é agendado para execução, fazendo Slave preencher os *buffers* com os dados recebidos. A seguir, Slave executa a função *callback* registrada, que corresponde ao método process() de Container e que, por sua vez, executa a função desejada do módulo LADSPA encapsulado. Finalmente, Slave envia os dados de volta para a máquina de origem, garantindo o processamento síncrono e remoto do módulo LADSPA.

6.2.4 Servidores remotos e comunicação

Nosso sistema para o processamento distribuído de áudio é baseado no processamento remoto de módulos LADSPA quando uma aplicação legada solicita a instanciação local desse módulo. Os elementos descritos anteriormente implementam os mecanismos usados para a comunicação e operação do sistema,

mas não contemplam o mecanismo para o estabelecimento inicial da comunicação entre as máquinas local e remota ou para a instanciação do módulo LADSPA remoto. Esse mecanismo deve promover:

- a escolha da máquina a ser usada para o processamento remoto;
- a definição do tamanho da janela a ser usado pelo mecanismo de janelas deslizantes;
- o estabelecimento da comunicação entre as classes Master e Slave do sistema;
- o fornecimento da informação referente ao módulo a ser instanciado ao sistema remoto;
- a criação do contexto para execução desse módulo.

Na implementação atual, esse procedimento envolve a classe Proxy e dois programas, manager e plugin_factory.

Proxy, manager e plugin_factory

Como vimos, a aplicação carrega dinamicamente uma biblioteca que disponibiliza para a aplicação os vários descritores LADSPA correspondentes ao módulos remotos. Quando a aplicação instancia um desses módulos, ela efetivamente cria uma instância de Proxy, que recebe o descritor LADSPA desejado como parâmetro em seu construtor. Proxy, por sua vez, é quem precisa dar início ao processo de instanciação do módulo LADSPA remoto e de configuração inicial da comunicação entre si mesmo e a instância remota de Container (através de Master e Slave).

Para que esse mecanismo possa funcionar adequadamente, é preciso que seja possível criar módulos LADSPA para o processamento remoto quando assim solicitado por uma aplicação. Portanto, é preciso que haja um servidor em execução nas diversas máquinas remotas do sistema que permita a instanciação desses módulos. No sistema DLADSPA, optamos por implementar esse servidor de acordo com a especificação CORBA, o que oferece as vantagens já abordadas anteriormente. Esse servidor é plugin_factory, que oferece uma interface simples para a criação e destruição de módulos LADSPA e se registra, durante sua inicialização, no serviço de nomes CORBA (Figura 6.2).

Em última instância, Proxy precisa se comunicar de alguma maneira com o plugin_factory. No entanto, a configuração inicial do sistema envolve uma fase interativa com o usuário; mesmo que valores *default* possam ser usados, é importante que seja possível ao usuário escolher, dentre as máquinas disponíveis, qual deve ser usada para o processamento, além de definir o tamanho da janela a ser usada pelo mecanismo de janelas deslizantes. Isso traz uma dificuldade para o sistema, pois Proxy é executada dentro do contexto de execução de uma aplicação legada qualquer; se Proxy tentar interagir com o usuário através de uma interface gráfica, pode haver incompatibilidades entre essa interface e a interface de usuário da aplicação. Além disso, vimos anteriormente que o uso de CORBA pode oferecer vantagens para o sistema; mas a carga de um ORB CORBA dentro do contexto de execução de uma aplicação legada também pode oferecer problemas de compatibilidade.

```
module DLADSPA {
    typedef unsigned long SampleRate;
    struct IPAddressSpec{
        string host;
        unsigned long port;
    };
    interface PluginFactory {
        IPAddressSpec newPlugin (in IPAddressSpec clientAddress,
            in string pluginDescriptor, in SampleRate rate);
        void deletePlugin (in IPAddressSpec serverAddress);
    };
};
```

Figura 6.2: IDL do servidor plugin_factory.

Para contornar essas dificuldades, Proxy se comunica, através de soquetes TCP, com uma aplicação auxiliar, manager, responsável por intermediar a comunicação entre Proxy e as máquinas remotas. Por ser um programa independente, manager pode criar interfaces de usuário e utilizar serviços CORBA sem interferir no funcionamento da aplicação legada que interage com Proxy; manager é, efetivamente, uma aplicação do padrão *Proxy* (Gamma et al., 1994, p. 207) (tal qual a própria classe Proxy).

Assim, Proxy comunica-se com manager e fornece uma cadeia de caracteres correspondente à descrição XML do módulo solicitado. manager, por sua vez, identifica, através do serviço de nomes CORBA, as máquinas disponíveis para o processamento remoto e apresenta uma interface (em modo texto) com o usuário para que ele escolha a máquina remota a ser usada e o tamanho da janela desejado. A seguir, manager solicita ao servidor escolhido que instancie um novo Container e fornece a descrição XML recebida de Proxy. manager também é responsável por definir os endereços e portas de comunicação a ser usados por Master (de Proxy) e Slave (de Container). Assim, a partir desse momento, Proxy pode executar chamadas ao método process() de sua instância de Master para realizar remotamente o processamento solicitado pela aplicação.

A implementação atual de manager faz uso muito limitado dos mecanismos CORBA; no entanto, como dito anteriormente, a utilização do serviço de negociações CORBA pode oferecer a possibilidade de configuração semi-automática do sistema. De maneira similar, embora a interface com o usuário atualmente implementada seja baseada em modo texto, é muito fácil adaptar manager para utilizar diversas interfaces gráficas (ou mesmo outras) com o usuário.

6.2.5 Reutilização do código

É de se esperar que o modelo de interação entre as classes Proxy e Container possa ser estendido para outras aplicações do nosso middleware. Assim, procuramos escrever o código de Proxy, Container, manager e plugin_factory da maneira mais genérica possível, restringindo ao máximo a interação dessas classes com as especificidades da especificação LADSPA.

Por interagirem diretamente com o sistema de módulos LADSPA, Proxy e Container necessariamente dependem de alterações para funcionar com novos padrões. No entanto, o uso de *templates* da linguagem C++ pode permitir que boa parte de seu código seja reutilizado em outras aplicações. manager, por outro lado, é totalmente independente do padrão LADSPA, e pode ser reutilizado praticamente sem alterações em outros ambientes. manager apenas manipula a descrição XML que corresponde ao módulo desejado, mas não procura interpretá-la em nenhum momento. Já plugin_factory utiliza uma classe virtual pura, FactoryManager, responsável por criar um servente CORBA, registrá-lo e removê-lo do serviço de nomes CORBA. As definições exatas do servente a ser criado e do nome a ser usado por FactoryManager, no entanto, são delegadas para suas subclasses. Assim, plugin_factory pode lidar com qualquer tipo de módulo para o qual uma subclasse de FactoryManager seja definida (essa é uma aplicação do padrão *Factory Method* — Gamma et al., 1994, p. 107). Seguindo esse modelo, criamos a subclasse DLADSPAFactoryManager para o tratamento de módulos LADSPA, que possui um servente específico para lidar com esses módulos como membro. Esse servente cria e executa o módulo LADSPA solicitado (uma instância de PluginChain) em uma *thread* separada.

6.3 Resultados experimentais

Para verificar a viabilidade do middleware aqui descrito, foram feitos experimentos com o sistema DLADSPA. Criamos um módulo LADSPA (o *plugin* waste_time) que simplesmente copia seus dados de entrada para a saída e então realiza um *busy-wait* por um certo período de tempo. Ao iniciar sua operação, ele espera, a cada iteração, pelo tempo necessário para fazer o tempo total de processamento daquela iteração igual a $100\mu s$; após $10s$ de operação, o tempo de espera a cada iteração passa a ser o suficiente para que a iteração tome $200\mu s$; depois de mais $10s$, $300\mu s$; e assim por diante. Quando o servidor JACK, jackd, começa a detectar erros temporais, sabemos que o sistema não pode mais dar conta do tempo de processamento usado pelo módulo e, portanto, registramos o maior tempo que uma iteração pode demorar antes que o sistema deixe de funcionar corretamente. O módulo também armazena algumas informações do sistema de arquivos virtual /proc do Linux sobre a carga do sistema em termos de tempo de usuário e tempo de sistema e, ao final do experimento, grava esses dados em um arquivo. Com essa configuração, fomos capazes de determinar a porcentagem máxima do período que está disponível para o processamento de um módulo LADSPA e, ao mesmo tempo, a carga gerada nas máquinas por esse processamento. Com esses dados, fomos capazes de determinar a carga adicional gerada pelo nosso middleware. Embora $10s$ possam parecer pouco tempo para cada experimento, devemos nos lembrar que cada iteração toma menos que $10ms$; portanto, $10s$ são suficientes para executar mais de 2000 iterações de cada configuração testada.

6.3.1 Ambiente de software e hardware

Para executar os experimentos, utilizamos o servidor JACK, jackd, em sua versão 0.72.4; ele nos permitiu a comunicação com o dispositivo de áudio com baixa latência e experimentar com diversos

tamanhos de período; o controlador de dispositivo de áudio usado foi o ALSA versão 0.9.4. Sobre o jackd, utilizamos a aplicação jack-rack ([JACKRACK]), versão 1.4.1, para carregar e executar nosso módulo. Essa aplicação é um processador de efeitos que funciona como um cliente jack que carrega módulos LADSPA e os aplica sobre os fluxos de dados oferecidos pelo jackd. Primeiramente foram feitas medições com o módulo waste_time executando-o na máquina local, como qualquer outro módulo LADSPA. Depois, realizamos experimentos onde o jack-rack carregava instâncias do nosso módulo proxy, que enviava os dados para instâncias remotas do módulo waste_time. Ao mesmo tempo, medimos (através do sistema de arquivos /proc) a carga de uso do processador na máquina central. Os experimentos foram executados parte com taxas de amostragem de áudio de 44.1KHz, parte com 96KHz; LADSPA e JACK tratam todas as amostras como floats de 32 bits.

A máquina central utilizada foi um PC Athlon 1.4GHz com 256MB de RAM, usando Debian GNU/Linux com o kernel do Linux versão 2.4.20 e os *patches* de baixa latência aplicados; a placa de som foi uma Delta44 da M-Audio ([DELTA]). As outras máquinas executavam um sistema GNU/Linux system com o mínimo para a execução da aplicação escrava, com o mesmo kernel da máquina central. As máquinas remotas utilizadas foram: outro PC Athlon 1.4GHz com 256MB de RAM, dois PCs Athlon 1.1GHz com 256MB de RAM, um PC AMD K6-2 400MHz com 192 MB RAM, dois PCs AMD K6-2 450MHz com 192MB RAM e um PC AMD K6-2 350MHz com 192MB RAM. Os athlons tinham placas de rede embutidas modelo SiS900, enquanto que os K6s utilizaram placas de rede PCI de baixo custo baseadas no chip 8139. As máquinas foram interconectadas por um switching hub Encore modelo ENH908-NWY+[2].

6.3.2 Limitações dos experimentos

O módulo waste_time, por ser muito simples, permite que o cache de memória das máquinas onde são executados esteja sempre preenchido com o código do núcleo do sistema operacional necessário para realizar as tarefas de comunicação via rede e agendamento de tarefas; isso provavelmente não aconteceria com uma aplicação real. Assim, é de se esperar que aplicações reais sofram um impacto maior da carga adicional gerada pelo middleware do que o que foi medido. Por outro lado, como veremos, essa carga adicional foi quase inexistente.

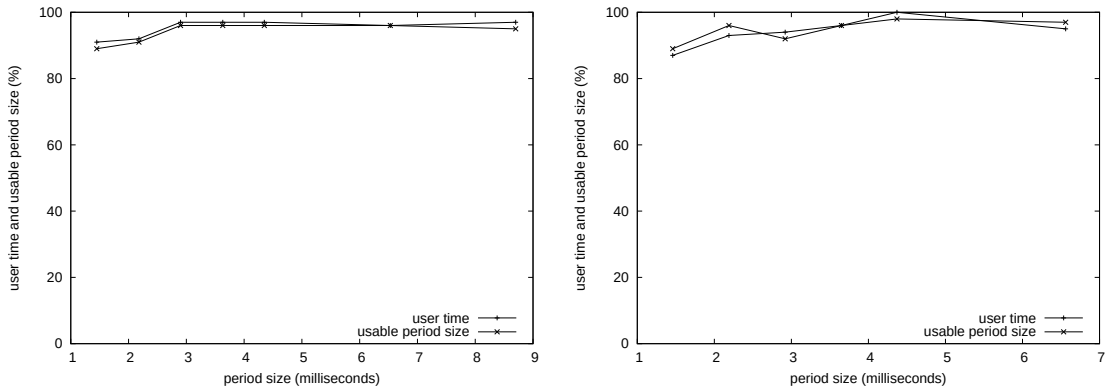
Embora o jackd rode extremamente bem no ambiente Linux, houve "xruns" ocasionais. "Xruns" são erros que ocorrem quando o sistema foi incapaz de ler ou gravar um *buffer* de dados de ou para a placa de som a tempo. Xruns entre 30–60μs ocorreram esporadicamente, quase sempre quando a carga gerada pelo processamento estava relativamente baixa; eles ocorreram, inclusive, quando o jackd estava em execução sem nenhum cliente e, portanto, a máquina não estava realizando nenhum processamento relacionado ao sistema. A razão provável é que, embora a latência de agendamento típica do Linux com os *patches* de baixa latência seja da ordem de 500μs, ela pode ser eventualmente maior, gerando atrasos também maiores. Quando a carga de processamento é maior, o sistema operacional muito provavelmente realiza menos mudanças de contexto entre cada interrupção, gerando menos atividade

[2]Agradecemos ao pessoal da *fonte design* por disponibilizarem o ambiente de testes.

de E/S e, portanto, mantendo-se mais próximo da latência típica de $500\mu s$. Esse problema impediu que realizássemos experimentos com períodos menores que $1.45ms$, quando esses xruns aleatórios se tornaram muito freqüentes. É provavelmente possível reduzir ou eliminar esses xruns configurando o jackd para utilizar três *buffers* de dados ao invés de dois para a comunicação com a placa de som (sob pena de aumento na latência do sistema), com uma máquina mais rápida ou com alguns ajustes de desempenho na máquina, mas não tentamos nada disso: nós simplesmente ignoramos esses xruns, já que eles não são relacionados com o nosso sistema e são facilmente discerníveis dos xruns causados pela exaustão na capacidade de processamento.

6.3.3 Resultados

Inicialmente, tentamos determinar o tempo máximo que o módulo waste_time podia gastar em cada período executando na máquina local; este experimento serve como um parâmetro de comparação para o processamento distribuído em que estamos interessados (Figura 6.3).



(a) taxa de amostragem 44.1KHz

(b) taxa de amostragem 96KHz

Figura 6.3: Em uma única máquina, períodos maiores permitem ao módulo utilizar mais tempo do processador, aumentando a eficiência.

Como esperado, períodos menores tornam o sistema menos eficiente, porque correspondem a uma maior taxa de interrupções e, portanto, a uma carga adicional maior. Além disso, a latência de agendamento é proporcionalmente maior com períodos menores: depois que a placa de som gera uma interrupção, o núcleo do sistema operacional leva aproximadamente $500\mu s$ para agendar a aplicação para ser executada e processar os dados, o que corresponde a aproximadamente 30% do tempo entre as interrupções se o período é $1.45ms$. Finalmente, irregularidades na latência de agendamento se tornam mais evidentes com períodos menores. A 44.1KHz e com um período de $1.45ms$, 89% do período pode ser usado pelo módulo; a carga sobre o sistema foi de 91% de tempo de usuário (*user time*: o tempo gasto pelo processador com processos comuns) e 1% de tempo de sistema (*system time*: o tempo gasto

pelo processador em tarefas gerais do núcleo). Portanto, não pudemos utilizar mais que 92% da capacidade do processador com períodos desse tamanho. A 96KHz e com um período do mesmo tamanho, o máximo tempo utilizável do período e a utilização máxima do processador foram um pouco menores. A 44.1KHz com períodos de 3.0ms ou mais, 96% do período puderam ser utilizados pelo módulo; a carga do sistema foi de cerca de 97% de tempo de usuário e 0% de tempo de sistema. Para períodos desses tamanhos, portanto, pudemos utilizar quase toda a capacidade de processamento do processador.

O experimento seguinte realizado foi executar o módulo waste_time em máquinas remotas utilizando o mecanismo de comunicação sem o uso do mecanismo de janelas deslizantes, i.e., a máquina central realizava espera em laço (*busy-wait*) até que os dados processados fossem recebidos de volta (Figura 6.4).

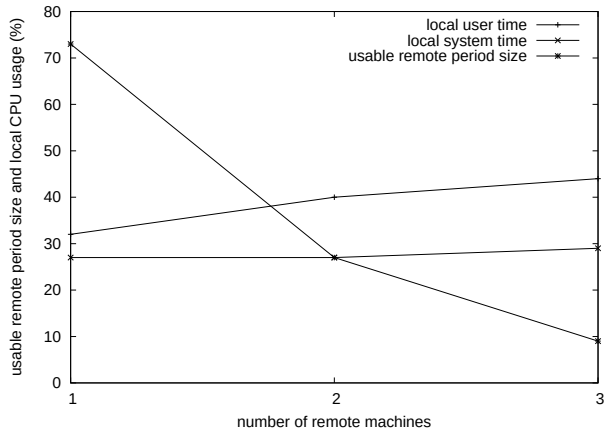


Figura 6.4: O tempo útil dos processadores remotos diminui rapidamente com o acréscimo de mais máquinas quando não se utilizam janelas deslizantes.

Esse experimento demonstrou que o sistema não é viável sem o uso de janelas deslizantes; não há ganho algum de desempenho nessa configuração. Os tempos de usuário e de sistema medidos na máquina central podem ser enganosos: eles correspondem primariamente à espera em laço, que envolve um laço dentro da aplicação que realiza chamadas ao sistema (read()). Seria possível utilizar esse tempo para o processamento de um módulo local, processando dados em paralelo com as máquinas remotas. Ainda assim, o tempo máximo do período disponível para processamento remoto é 73% do tamanho do período para uma máquina remota; com duas máquinas remotas, esse número cai para 27%; com três, cai ainda mais, para 9%. Essa configuração não justifica o uso de processamento distribuído.

A seguir, queríamos determinar o tempo máximo que os módulos poderiam utilizar a cada período quando executados em várias máquinas remotas utilizando-se janelas de tamanho 1. Executamos o experimento com quatro máquinas remotas a 96KHz utilizando diversos períodos e com sete máquinas remotas a 96KHz utilizando um período de 2.19ms (Figura 6.5). Esses experimentos mostraram que a utilização dos processadores remotos é excelente ao mesmo tempo em que a carga adicional na máquina central é razoavelmente baixa, mesmo com um grande número de máquinas: com períodos

de 2.19ms numa taxa de amostragem de 96KHz, os módulos waste_time em execução nas máquinas remotas puderam utilizar 96% do período para o processamento, permitindo uma utilização de 99% da capacidade dos processadores remotos. A carga no processador da máquina central foi aproximadamente a mesma que a medida no experimento anterior, 45% de tempo de usuário e 20% de tempo de sistema.

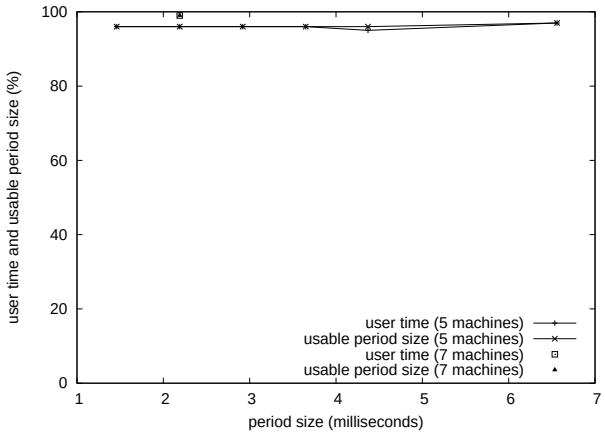


Figura 6.5: Usando o mecanismo de janelas deslizantes (de tamanho um), o tempo do processador disponível para os módulos nas máquinas remotas é similar ao disponível em uma única máquina.

Finalmente, quisemos determinar a carga adicional introduzida pelo sistema na máquina central sem nenhuma espera em laço; para realizar esse experimento, configuramos a camada de comunicação para utilizar janelas de tamanho dois; a seguir, executamos o experimento com períodos de 2.18ms a 44.1KHz com uma, duas, três e quatro máquinas remotas, bem como a 96KHz com quatro e sete máquinas remotas. (Figura 6.6).

A carga gerada pelo sistema sobre a máquina central, embora significativa, é totalmente aceitável; ela também cresce de forma aproximadamente linear, como esperado. Também verificamos que a carga gerada na máquina com quatro máquinas operando com taxa de amostragem de 96KHz é quase a mesma que a carga gerada com quatro máquinas operando a 44.1KHz, o que foi inesperado.

6.3.4 Discussão dos resultados

Como vimos, o processamento verdadeiramente síncrono (utilizando janelas de tamanho zero) não é uma abordagem interessante para o problema da distribuição do processamento de multimídia com baixa latência. A utilização das janelas deslizantes, no entanto, mostrou-se uma técnica eficiente: com janelas de tamanho 1, foi possível distribuir o processamento de áudio com taxa de amostragem de 96KHz entre sete máquinas remotas com baixíssima carga adicional nessas máquinas remotas (permitindo o uso de quase 100% da capacidade do processador para o processamento) e com uma carga aceitável na máquina central. É razoável esperar que o número máximo de máquinas remotas que podem ser usadas numa configuração similar com o mesmo hardware utilizado seja ainda maior, já que nem a largura

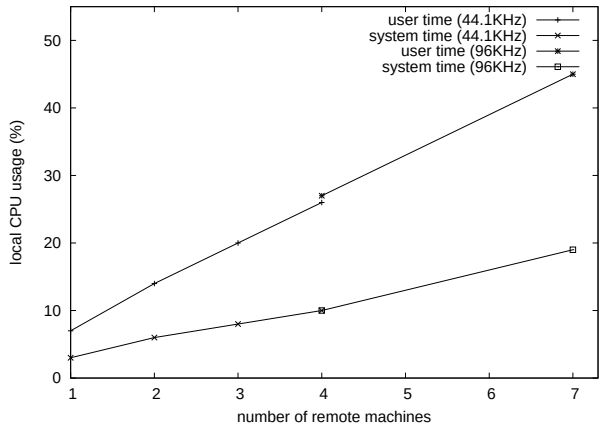


Figura 6.6: A carga na máquina central cresce linearmente com o número de máquinas remotas.

de banda nem a capacidade de processamento da máquina central atingiram seus máximos durante os experimentos. Com processadores mais rápidos e placas de rede capazes de onerar menos o processador para a comunicação, o limite provavelmente seria definido pela velocidade física da rede (como visto anteriormente — Seção 5.1.4, equivalente a cerca de 30 canais de áudio), pelo menos no caso de redes padrão Fast Ethernet. Experimentos com redes mais rápidas, como IEEE 1394 e Gibabit Ethernet, são necessários para avaliar o desempenho do sistema nesses tipos de rede.

Capítulo 7

Conclusão

Como vimos, praticamente não existem sistemas para o processamento de multimídia em tempo real que tirem proveito de técnicas de programação paralela em sistemas de uso geral, como multiprocessamento ou processamento distribuído. No entanto, apesar das dificuldades para o processamento distribuído em tempo real, a possibilidade de realizar processamento de multimídia em um sistema distribuído abre a possibilidade da redução de custos e aumento de flexibilidade para aplicações multimídia. O uso de softwares livre, por outro lado, traz consigo diversos benefícios, e o incentivo ao crescimento do uso de software livre no Brasil pode trazer diversas vantagens.

A forte presença da música "pop" na cultura brasileira faz com que haja uma vasta produção musical por parte de músicos amadores e semi-profissionais, com o conseqüente crescimento de estúdios domésticos e de pequeno porte. Tais pequenos estúdios obviamente dispõem de recursos limitados para a compra de equipamentos; sistemas de baixo custo, portanto, podem abrir espaço cada vez maior para a produção musical nesses ambientes. Portanto, dentro do universo de possibilidades do processamento de multimídia, o caso específico do áudio se apresenta como particularmente interessante para o desenvolvimento de sistemas distribuídos de baixo custo.

Assim, apresentamos, neste trabalho, o sistema DLADSPA, voltado para o processamento distribuído de áudio em tempo real e com baixa latência em redes locais em ambiente Linux. O sistema, embora ainda deva ser considerado experimental, já está em condições de ser usado para a produção musical na prática. Esse sistema está disponível sob licença GPL na web em <http://gsd.ime.usp.br/software/DistributedAudio>, e consiste em cerca de 7300 linhas de código: 2000 que compõem o middleware e 5300 que compõem a aplicação DLADSPA, num total de 27 classes C++.

Ao longo deste trabalho, discutimos as vantagens do uso de funções *callback* em sistemas para processamento síncrono em tempo real com baixa latência, bem como as características gerais de sistemas de tempo real, de sistemas paralelos e distribuídos e de sistemas distribuídos de tempo real. Abordamos, em particular, o uso de linhas de produção para o processamento paralelo e distribuído e mostramos que elas oferecem diversas possibilidades no equilíbrio entre aumento na paralelização e redução na latência. Também vimos que é razoavelmente simples utilizar o mecanismo de linhas de produção para introduzir a paralelização no processamento em grande parte das aplicações de áudio e que esse mecanismo

é compatível com o uso de funções *callback*. A seguir, discutimos a relevância do processamento com baixa latência em sistemas interativos e sua relação com a percepção humana. Finalmente, mostramos como um sistema distribuído de tempo real pode fazer uso de linhas de produção para o processamento síncrono e com baixa latência.

Os experimentos realizados mostraram que o sistema permite o processamento distribuído de áudio com pelo menos oito máquinas simultaneamente, provavelmente mais, utilizando-se dispositivos de rede de baixo custo. Isso significa que, dadas as tecnologias de rede disponíveis atualmente por custos razoavelmente baixos, o processamento distribuído de áudio em tempo real com baixa latência é viável e oferece vantagens de desempenho. Experimentos futuros deverão determinar se esse modo de operação também é adequado para o processamento de outras mídias, em particular vídeo.

O sistema desenvolvido é compatível, graças à utilização da especificação LADSPA, com uma vasta gama de aplicações legadas voltadas para a música já existentes no ambiente Linux, além de permitir o uso dos diversos módulos LADSPA já existentes. O uso de uma especificação para interconexão de componentes, portanto, permitiu a compatibilidade do mecanismo aqui descrito com aplicações e módulos legados; seguindo o mesmo princípio, o sistema trata os elementos que distribui de forma modular. Para simplificar a reutilização do código por nós desenvolvido em outras aplicações, as partes do sistema responsáveis pela comunicação entre as máquinas que não são relacionadas diretamente com a especificação LADSPA foram desenvolvidas sob a forma de um pequeno middleware. Com base nesse middleware foi desenvolvida uma camada de compatibilidade com a especificação LADSPA, permitindo ao usuário concatenar seqüências de módulos LADSPA a serem processadas remotamente. Mecanismos similares podem ser usados para reutilizar o middleware desenvolvido em aplicações envolvendo outras mídias ou outras especificações (como a VST) ou, ainda, novas aplicações voltadas especificamente para a operação distribuída podem fazer uso desse middleware como base do seu sistema de distribuição.

7.1 Trabalhos futuros

O sistema DLADSPA está em condições de ser utilizado pelos interessados em processamento de áudio em ambiente Linux; apesar disso, diversas melhorias podem ser implementadas no sistema. Algumas dessas melhorias abrem a possibilidade de novas pesquisas na área de interesse do sistema, como a expansão para outras mídias, a possibilidade do uso de linhas de produção mais longas, o gerenciamento dos recursos remotos etc.

7.1.1 Middleware

Dentre as melhorias a serem implementadas no middleware, podemos destacar:

- Implementar melhorias no código, particularmente mecanismos mais robustos para o tratamento de erros. A implementação atual simplesmente aborta o programa em execução quando ocorre uma situação de erro (falha em uma conexão, falha ao carregar um módulo LADSPA etc.).

- Fazer uso do mecanismo de *templates* do C++ para as classes de comunicação. A implementação atual está baseada no uso do protocolo UDP para a comunicação entre as máquinas; no entanto, é interessante implementar o mecanismo de comunicação através de *templates*, que permitiriam o uso de diversos protocolos de comunicação sem impacto no desempenho.

- Possibilitar o uso de linhas de produção mais longas. A implementação atual considera que cada fluxo de dados deve ser processado remotamente por apenas uma máquina. No entanto, em algumas aplicações pode ser interessante permitir que um fluxo de dados seja processado seqüencialmente em mais de uma máquina, a despeito do impacto correspondente sobre a latência de processamento.

- Possibilitar processamento não-circular. O sistema atual pressupõe que dados são gerados em uma máquina (por exemplo, capturados por um dispositivo de E/S), processados em outras máquinas e devolvidos para a máquina de origem (possivelmente para reprodução através do dispositivo de E/S). Seria interessante permitir que dispositivos de E/S distintos, em máquinas diferentes, pudessem ser usados simultaneamente; ou seja, permitir que dados sejam produzidos em uma máquina, processados em outra e reproduzidos ainda em outra. Para que isso seja possível mantendo as características de sincronização e baixa latência do sistema, esses dispositivos de E/S precisam gerar requisições de interrupções de forma sincronizada. Em aplicações de áudio, isso é possível com a utilização de hardware com suporte a sincronização unificada (*word clock*).

- Absorver parte do código específico da aplicação LADSPA. É possível que partes do código vinculadas ao sistema para a distribuição de módulos LADSPA possam ser tornadas mais genéricas e incorporadas ao middleware.

- Avaliar a possibilidade de subsituir *callbacks* por *functors* (Josuttis, 1999, p. 124–130; Stroustrup, 1997, p. 514). Ao invés de utilizar uma função *callback* registrada junto à classe Slave para realizar o processamento, o uso de um *functor* ofereceria a possibilidade de realizar o mesmo processamento mais eficientemente. No entanto, essa mudança pode limitar a possibilidade do sistema de permitir que diversos tipos de processamento sejam escolhidos em tempo de execução.

- Implementar um sistema para gerência dos recursos remotos. No sistema atual, o usuário é responsável por escolher quais partes do processamento devem ser realizadas por quais máquinas. Seria interessante que o sistema automaticamente fosse capaz de gerenciar os recursos remotos e decidisse automaticamente por uma configuração. Para que isso seja possível, é preciso implementar políticas de escalonamento de tempo real em cada uma das máquinas remotas e, se possível, implementar um sistema para a migração do processamento entre as máquinas.

- Implementar novas aplicações para o middleware. Seria interessante permitir, por exemplo, que *patches* do sistema jMAX possam ser executados remotamente. Uma abordagem possível seria implementar um *driver* para o sistema JACK baseado no middleware aqui descrito. De forma similar, permitir uma melhor integração com o sistema *gstreamer* simplificaria o desenvolvimento

de aplicações especificamente voltadas para o processamento distribuído de multimídia. Outras aplicações envolvendo outras mídias também seriam de interesse.

7.1.2 Aplicação DLADSPA

De forma similar, seria interessante implementar melhorias na aplicação DLADSPA desenvolvida, tais como:

- Adaptar o gerenciador e o servidor remoto para o uso com outros tipos de módulos, possivelmente em outras plataformas (como VST em Windows). O sistema atual já é razoavelmente genérico nesse sentido, mas seria interessante permitir que uma única instância de cada um desses programas fosse capaz de criar e gerenciar módulos de diferentes tipos simultaneamente. Esse mecanismo poderia, inclusive, ser estendido para outras mídias.

- Possibilitar à aplicação que faz uso do sistema compensar a latência introduzida. Algumas aplicações para o processamento de áudio (como, por exemplo, o Ardour) são capazes de compensar a latência introduzida por um módulo DSP. Para que isso seja possível, é preciso que o módulo seja capaz de informar a aplicação sobre o tamanho da latência introduzida. A especificação LADSPA não oferece um mecanismo para isso, mas alguns desenvolvedores têm utilizado uma porta LADSPA comum como via para esse tipo de comunicação. Seria interessante implementar esse mecanismo no sistema.

- Avaliar o real interesse do uso de CORBA no sistema descrito. Como as aplicações previstas são baseadas em redes locais, é possível que configurações menos flexíveis que as possíveis com o uso de CORBA sejam suficientes para o sistema, simplificando diversos aspectos do desenvolvimento e da interface com o usuário.

- Possibilitar a criação de novas cadeias de processamento em tempo de execução. A especificação LADSPA pressupõe que o conjunto de módulos LADSPA disponíveis para a aplicação é fixo a cada execução da aplicação. Como conseqüência, cada vez que o usuário precisa definir uma nova cadeia de módulos LADSPA a serem processados remotamente, é preciso finalizar e reiniciar a aplicação que fará uso dessa nova cadeia. Seria interessante permitir que o usuário fosse capaz de definir uma cadeia a ser processada remotamente no momento da instanciação de uma nova cadeia. Para que isso seja possível, é necessária uma mudança significativa no paradigma de funcionamento do LADSPA. Uma maneira de contornar esse problema seria através do desenvolvimento de uma aplicação específica, similar ao jack-rack (talvez baseada nele), capaz de definir os módulos a serem processados remotamente e solicitar a instanciação remota desses módulos.

- Permitir a criação de grafos de módulos. A implementação atual permite ao usuário criar cadeias de módulos LADSPA a serem executados remotamente. Embora isso seja suficiente (e típico) para a maioria das aplicações, seria interessante oferecer a possibilidade de criação de grafos de módulos.

Bibliografia

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. Disponível em: <http://human-factors.arc.nasa.gov/ihh/spatial/papers/pdfs_bda/Adelstein_2003_Haptic_Audio_Asynchrony.pdf> Acesso em: 10 mar 2004.

[ALSA] <http://www.alsa-project.org>. *Sítio na Web do sistema ALSA*. Acesso em: 8 fev 2004.

[ARDOUR] <http://www.ardour.org>. *Sítio na Web do programa Ardour*. Acesso em: 8 fev 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>. *Sítio na Web do padrão ASIO*. Acesso em: 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. Disponível em: <http://www.fsmlabs.com/articles/archive/lj.pdf> Acesso em: 5 abr 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. Disponível em: <http://citeseer.nj.nec.com/bernat99weakly.html> Acesso em: 2 fev 2004.

BILMES, Jeffrey Adam. *Timing is of the Essence: Perceptual and Computational Techniques for Representing, Learning, and Reproducing Expressive Timing in Percussive Rhythm*. Dissertação (Mestrado) — MIT, Cambridge, Massachusetts, 1993. Disponível em: <http://www.icsi.berkeley.edu/~bilmes/mitthesis/mit-thesis.pdf> Acesso em: 10 mar 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. Disponível em: <ftp://ftp.pinnaclesys.com/Steinberg/download/Documents/Nuendo_2/english/Nuendo_21_Operation_Manual_en_11082K.pdf> Acesso em: 5 abr 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>. *Sítio na Web do programa Ceres*. Acesso em: 5 abr 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. Disponível em: <http://citeseer.nj.nec.com/choudhary94optimal.html> Acesso em: 15 fev 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. Disponível em: <http://www.superconductor.com/clynes/entitie.htm> Acesso em: 10 mar 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. Disponível em: <http://www.superconductor.com/clynes/cognit.htm> Acesso em: 10 mar 2004.

[COMMONMUSIC] <http://www-ccrma.stanford.edu/software/cm/doc/cm.html>. *Sítio na Web do programa Common Music*. Acesso em: 8 fev 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>. *Sítio na Web do padrão CoreAudio*. Acesso em: 5 abr 2004.

D'Antonio, P. Minimizing acoustic distortion in project studios. Disponível em: <http://www.rpginc.com/cgi-bin/byteserver.pl/news/library/PS_AcD.pdf> Acesso em: 5 abr 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*. Acesso em: 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. Disponível em: <http://www-unix.mcs.anl.gov/dbpp> Acesso em: 15 fev 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>. *Sítio na Web da Free Software Foundation*. Acesso em: 8 fev 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. Disponível em: <http://www.oefai.at/cgi-bin/get-tr?paper=oefai-tr-2003-11.pdf> Acesso em: 10 mar 2004.

[GSTREAMER] <http://www.gstreamer.net>. *Sítio na Web do sistema gstreamer*. Acesso em: 10 fev 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.]. Disponível em: <http://www.nd.edu/~lemmon/rtss03_submitted.pdf> Acesso em: 2 fev 2004.

IYER, Vijay S. *Microstructures of Feel, Macrostructures of Sound: Embodied Cognition in West African and African-American Musics*. Tese (Doutorado) — University of California, Berkeley, Berkeley, California, 1998. Disponível em: <http://cnmat.cnmat.berkeley.edu/People/Vijay/%20THESIS.html> Acesso em: 10 mar 2004.

[JACK] <http://jackit.sourceforge.net>. *Sítio na Web do padrão JACK*. Acesso em: 5 abr 2004.

[JACKRACK] <http://arb.bash.sh/~rah/software/jack-rack/>. *Sítio na Web do programa Jack-Rack*. Acesso em: 5 abr 2004.

[JMAX] <http://freesoftware.ircam.fr/rubrique.php3?id_rubrique=14>. *Sítio na Web do programa jMAX*. Acesso em: 8 fev 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 e a Questão Social*. São Paulo, maio 2001. Disponível em: <http://www.ime.usp.br/~kon/papers/RT-SoftwareAberto.pdf> Acesso em: 5 abr 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/>. *Sítio na Web da lista de desenvolvedores de sistemas para áudio em Linux*. Acesso em: 5 abr 2004.

[LADSPA] <http://www.ladspa.org>. *Sítio na Web do padrão LADSPA*. Acesso em: 5 abr 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>. *Sítio na Web do programa latencytest*. Acesso em: 8 fev 2004.

LEVITIN, Daniel J. et al. The perception of cross-modal simultaneity. Disponível em: <http://ccrma-www.stanford.edu/~lonny/papers/casys1999.pdf> Acesso em: 10 mar 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>. *Sítio na Web do patch de baixa latência para o Linux 2.2*. Acesso em: 8 fev 2004.

[LLPATCH2.4] <http://www.zip.com.au/~akpm/linux/schedlat.html>. *Latência de agendamento em Linux*. Página do patch de baixa latência para o Linux. Accesso em: 8 fev 2004.

[LLPATCH2.6] <http://www.kernel.org/pub/linux/kernel/people/rml/preempt-kernel/v2.4>. *Sítio na Web do patch para mudanças de contexto no núcleo do Linux*. Incorporado ao Linux 2.6. Acesso em: 8 fev 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. Disponível em: <http://gigue.peabody.jhu.edu/~ich/research/icmc01/latency-icmc2001.pdf> Acesso em: 5 abr 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>. *Sítio na Web do distribuidor do programa MAX/MSP*. Acesso em: 8 fev 2004.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. 1995. Disponível em: <http://www.mpi-forum.org/docs/mpi-11.ps> Acesso em: 5 abr 2004.

Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. 1997. Disponível em: <http://www.mpi-forum.org/docs/mpi-20.ps> Acesso em: 5 abr 2004.

Metts, Allan. Sounds from another planet. *Electronic Musician*, January 2004. Disponível em: <http://emusician.com/ar/emusic_sounds_planet/index.htm> Acesso em: 08 fev 2004.

[MIDI] <http://www.midi.org>. *Sítio na Web do consórcio de empresas responsável pela especificação MIDI*. Acesso em: 10 fev 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. Disponível em: <http://www.faqs.org/rfcs/rfc1305.html> Acesso em: 5 abr 2004.

[MIXVIEWS] <http://www.ccmrc.ucsb.edu/~doug/htmls/MiXViews.html>. *Sítio na Web do programa MiXViews*. Acesso em: 5 abr 2004.

[MUSE] <http://muse.seh.de>. *Sítio na Web do programa Muse*. Acesso em: 5 abr 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>. *Sítio na Web da Open Source Initiative*. Acesso em: 8 fev 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>. *Sítio na Web do programa Patchwork*. Acesso em: 5 abr 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>. *Sítio na Web da coleção de módulos LADSPA de Steve Harris*. Acesso em: 5 abr 2004.

[POSIX] <http://www.pasc.org>. *Sítio na Web do Portable Application Standards Comitee*. Responsáveis pelos padrões POSIX do IEEE. Acesso em: 8 fev 2004.

[PROTOOLS] <http://www.digidesign.com>. *Sídio na Web do fabricante do sistema Pro Tools*. Acesso em: 10 fev 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. Há uma versão online em: <http://www.catb.org/~esr/writings/cathedral-bazaar> Acesso em: 5 abr 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. Disponível em: <http://webster.cs.ucr.edu/AoA/DOS/> Acesso em: 5 abr 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. Disponível em: <http://www-ccrma.stanford.edu/groups/soundwire/performdelay.pdf> Acesso em: 10 mar 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. Disponível em: <http://www.faqs.org/rfcs/rfc3551.html> Acesso em: 5 abr 2004.

SCHULZRINNE, H. et al. *RFC3550: RTP: A Transport Protocol for Real-Time Applications*. 2003. Disponível em: <http://www.faqs.org/rfcs/rfc3550.html> Acesso em: 5 abr 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 e inclusão digital*. São Paulo: Conrad Editora do Brasil, 2003.

[SOUNDAPPS] <http://sound.condorow.net>. *Sítio na Web com uma lista bastante completa e atualizada de programas para música e áudio em ambiente Linux*. Acesso em: 8 fev 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. Disponível em: <http://www.ittc.ku.edu/kurt/papers/conference.ps.gz> Acesso em: 5 abr 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. Disponível em: <http://www.idt.mdh.se/kurser/ct3200/regular/ht03/download/articles/rt-intro.pdf> Acesso em: 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. Disponível em: <http://jacquespaillard.apinc.org/deafferented/pdf/stenneken-et-al-ms-03.pdf> Acesso em: 10 mar 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. Disponível em: <http://www.unix-systems.org/version3/online.html> Acesso em: 5 abr 2004.

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

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

[VST] <http://www.steinberg.net/en/ps/support/3rdparty/vst_sdk/index.php?sid=0>. *Sítio na Web da especificação VST*. Acesso em: 24 jul 2003.

[VSTSERVER] <http://www.notam02.no/arkiv/src>. *Sítio na Web do programa VSTserver*. Acesso em: 5 abr 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. Disponível em: <http://www.cnmat.berkeley.edu/ICMC97/papers-html/OpenSoundControl.html> Acesso em: 5 abr 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. Disponível em: <http://www.cnmat.berkeley.edu/Research/NIME2003/NIME03_Wright.pdf> Acesso em: 5 abr 2004.

[XORG] <http://www.x.org>. *Sítio na Web do X Consortium, responsável pelo sistema de janelas X.* Acesso em: 5 abr 2004.

Índice Remissivo

/proc, 87

activate(), 83
adaptação, 46, 48
adapter, *veja* padrão de projeto
adiantamento da melodia, 26
agendamento de tarefas, 61
agitação, 20, 22
ALSA, 34, 35, 88
arcabouço, 77
ardour, 7, 12, 96
arquiteturas fechadas, *veja* sistemas fechados
ASIO, 6, 33, 34
assincronia, 23, 25, 27
ataque, 24–26
audição, 20, 21
automóvel, 21

baixa latência, *veja* latência, baixa
best-effort, *veja* melhor esforço
biprocessamento, 7, 9
blocking call, *veja* bloqueio
bloqueio, 34–36, 59, 73
Buarque, Chico, *veja* Chico Buarque
buffer, 30, 32, 34, 35, 39, 58, 67, 69–72, 83, 88
busy-wait, *veja* espera em laço, 87

C++, 67, 87, 93, 95
cálculos astronômicos, 42
cache, 83
Caetano Veloso, 3
callback, 17, 30–36, 39, 58, 60, 67, 84, 93, 95
camadas, 2, 10
canto, 29
cassete, fita, *veja* fita cassete
catástrofe, *veja* falha catastrófica
Chico Buarque, 3
cinestesia, *veja* tátil-cinestésica, sensação
colisão, 21, 52
comb filtering, 23
componentes, 13
composite, *veja* padrão de projeto
computação imprecisa, 49, 51
computação musical, 2, 21, 25
connect_port(), 83
Container, 80, 84
contratempo, 25
CORBA, 43, 58, 66, 75, 80, 85, 96
coreAudio, 6, 33, 34
correctReceivedData(), 73
cultura musical, 4

DAG, 41
DataBlock, 72
DataBlockSet, 72
deactivate(), 83
debian, 88
dedicado, hardware, *veja* hardware dedicado
Delta44, 88
demo, 3
design pattern, *veja* padrão de projeto
dinâmica, 26, 28
distorcedor, 10
distributed LADSPA, *veja* DLADSPA
DLADSPA, 9, 56, 76–87, 93, 94, 96

dlopen(), 83
DSP, 13, 28, 34, 37, 56, 96

E/S, 30–32, 34, 35, 59, 61, 77
endianness, *veja* ordenação de bytes
Entrada e Saída, *veja* E/S
escalonamento de processos, *veja* escalonamento de tarefas
escalonamento de tarefas, 31, 36
espacialização sonora, 23
específico, hardware, *veja* hardware dedicado
espera em laço, 90
estúdio, 2, 3, 93
experimentação, 6

factory method, *veja* padrão de projeto
FactoryManager, 87
falha catastrófica, 46
fast ethernet, 2, 17, 61, 62, 92
feedback, 27, 29
firewire, *veja* IEEE 1394
firm real-time, *veja* tempo real firme
fita cassete, 3
framework, *veja* arcabouço
free software foundation, 4
full-duplex, 33, 65
functionToCall, 83, 84
functors, 69, 95

Gil, Gilberto, *veja* Gilberto Gil
Gilberto Gil, 3
GPL, 9, 93
grafo, 41, 96
granularidade, 77
gravadores multipista, 3
gstreamer, 12, 77
guitarra elétrica, 10

half-duplex, 62
handle, 37
haptics, *veja* tátil-cinestésica, sensação
hard way to make music, 1
hardware dedicado, 4, 7
hardware específico, *veja* hardware dedicado
htons(), 72

I/O, *veja* E/S
IEEE, 6
IEEE 1394, 63, 92
inclusão digital, 5
interatividade, 1, 10
  e latência, *veja* latência e interatividade
  e multimídia, *veja* multimídia e interatividade
Internet, 4, 17
interrupção, 16, 30–33, 58, 61
IPAddress, 69
isocronismo, 65, 70, 84

JACK, 6, 16, 33, 34, 77, 87, 95
jack-rack, 79, 82, 83, 88, 96
jackd, 5, 36, 77, 87, 88
janelas deslizantes, 16, 64, 90, 91
jitter, *veja* agitação
jMAX, *veja* MAX

kernel patch, *veja* patch para baixa latência

LAAGA, 36
LADSPA, 2, 6, 13, 28, 34, 37, 76, 77, 81, 83, 84, 88, 94
LADSPA_Descriptor, 37, 81
ladspa_descriptor(), 37
LADSPA_PROPERTY_INPLACE_BROKEN, 83
latência, 20
  baixa, 1, 10, 14, 52, 58, 61, 93
  de processamento, 52
  e desenvolvimento, 28, 32
  e interatividade, 9, 20–29
  e percepção, 10, 20–29, 94
  e redes, 21, 61–65

e sincronização, 24
patch, *veja* patch para baixa latência
libjack, 36
linguagem natural, 42
linha de produção, 9, 45, 53, 57, 64, 93, 95
linux, 2, 4, 8, 12, 32, 34, 49, 57, 93
lote, processamento, *veja* processamento em lote

M-Audio, 88
módulos, 10, 12, 13, 34, 37, 56, 70, 76–78, 82, 94, 96
música étnica, 22
música de câmara, 25
música folclórica, 22
música independente, 4
música pop, 3, 22, 25, 93
música popular, *veja* música pop
mídia contínua, 1, 10, 14, 15, 17, 20, 21, 52, 56
macOS, 7, 8, 12, 32, 34, 37, 49
manager, 85, 86
Master, 67, 73, 83
MAX, 3, 12, 95
melhor esforço, 47, 50, 57
melody lead, *veja* adiantamento da melodia
Message-Passing Interface, *veja* MPI
middleware, 2, 14, 43, 56–76, 94
MIDI, 3, 15, 16
mouse, 21
MP3, 66
MPB, *veja* música pop
MPEG, 66
MPI, 18
MultiChannelPlugin, 81
MultiChannelPluginType, 81
multimídia, 56
e baixa latência, 10
e fluxos de dados, 11
e interatividade, 1, 10, 52
e latência, 21, 26, 32
e paralelismo, 8
multipista, gravadores, *veja* gravadores multipista
multiprocessamento, 2, 7
mundo real, *veja* tempo real

non-real-time synthesis, 1
ntohs(), 72

Open Sound Control, *veja* OSC
open source initiative, 4
operator()(), 69
ordenação de bytes, 70, 72
orquestra, 25
OSC, 16

padrão de projeto
adapter, 76
composite, 81
factory method, 87
proxy, 78, 86
strategy, 67
patch para baixa latência, 6, 35, 88
pegada, 20, 25
período, 22, 23, 52
percepção
e latência, *veja* latência e percepção
e relação entre ação e reação, 26
e simultaneidade, 25
espacial, 23
rítmica, 24
performance, 2, 10
piano, 26, 28
pipeline, *veja* linha de produção
plug-ins, *veja* módulos
plugin_factory, 85
PluginChain, 83, 84
PluginChainType, 83
plugins, *veja* módulos
POSIX, 6, 35, 69

prazo firme, 50, 57
prazo flexível, 46, 47, 49
prazo rígido, 46, 47, 49
prepareToSendData(), 73
previsibilidade, 47
Pro Tools, 7, 12, 62
process(), 67, 73, 84
Processamento Digital de Sinais, *veja* DSP
processamento distribuído, 1, 13, 14
processamento em lote, 7
processamento paralelo, 8, 13, 42–45, 52–54, 56, 93
proprietary systems, *veja* sistemas fechados
Proxy, 80, 83, 84
proxy, *veja* padrão de projeto
pulso, 22, 24–28

read(), 32, 60, 90
readv(), 69, 70
real-time, *veja* tempo real
Real-Time Protocol, *veja* RTP
reconhecimento de padrões, 10, 14, 20, 42, 44
redes, 15, 17, 42, 58, 60
    e latência, *veja* latência e redes
    e tempo real, 61–65
remote_plugins.so, 83
reserva de recursos, 47
response time, *veja* latência
resposta de melhor esforço, *veja* melhor esforço
resposta garantida, 47, 50
resposta, tempo de, *veja* latência
robô, 20, 42
RTP, 17
run(), 39, 83
run_adding(), 83

sampleCount, 83
SCHED_FIFO, 35–37
semáforo, 36
simulação, 42
sincronização, 9, 16, 22, 25, 31–33, 43, 58, 61
    e multimídia, 11
sincronização unificada, 58, 95
sistema de janelas X, *veja* X, sistema de janelas
sistema operacional, 10, 11, 30, 31, 33, 47–49, 60
sistemas de tempo real, *veja* tempo real
sistemas distribuídos, 42–45, 58, 93
sistemas fechados, 4, 8
sistemas reativos, *veja* tempo real
Slave, 67, 73, 84
socket, *veja* soquete
software livre, 4, 93
soquete, 69, 81, 86
strategy, *veja* padrão de projeto
streaming, 15
struct iovec, 69, 70
system time, 89

tátil-cinestésica, sensação, 26
tato, *veja* tátil-cinestésica, sensação
taxa de amostragem, 11, 22, 23, 52, 88
TCP, 60, 81, 86
TCP/IP, 60, 69
template, 67, 87, 95
tempo de processamento, 22
tempo de resposta, *veja* latência
tempo real, 2, 10, 14, 20, 33–35, 45–54, 57, 60, 93
    e redes, *veja* redes e tempo real
tempo real firme, 48–49, 58
textura, 22
throughput, *veja* vazão de processamento
timbre, 23

UDP, 17, 60, 67, 69, 95
UdpRead, 69
UdpWrite, 69
UNIX, 6, 12, 35
user time, 89

vídeo-conferência, 15, 17
vazão de processamento, 52
Veloso, Caetano, *veja* Caetano Veloso
visão, 21
visão computacional, 20, 52
VST, 6, 34, 37, 94, 96
VST System Link, 15, 77
VSTServer, 37

waste_time, 87, 88, 91
windows, 7, 8, 12, 32, 34, 37, 49, 96
word clock, *veja* sincronização unificada
write(), 32, 60
writev(), 69, 70

X, sistema de janelas, 77
XML, 79, 83, 86, 87
xruns, 88