Composição e Performance Musical Utilizando Agentes Móveis

Leo Kazuhiro Ueda

Dissertação apresentada ao

Instituto de Matemática e Estatística da

Universidade de São Paulo

para a obtenção do grau de
Mestre em Ciências.

Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Fabio Kon

Durante o desenvolvimento deste trabalho, o autor recebeu apoio financeiro da CAPES.

São Paulo, outubro de 2004.

Composição e Performance Musical Utilizando Agentes Móveis

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Leo Kazuhiro Ueda e aprovada pela comissão julgadora.

São Paulo, 18 de novembro de 2004.

Comissão julgadora:

- Prof. Dr. Fabio Kon (orientador) - IME/USP
- Prof. Dr. Flávio Soares Corrêa da Silva - IME/USP
- Prof. Dr. Fernando Iazzetta - Depto. de Música da ECA/USP

Agradecimentos

Em primeiro lugar, agradeço a minha mãe e a meu pai, Yuzuri e Hiroomi, pelo carinho e pela enorme paciência. Agradeço também ao meu irmão Max e à minha irmã Erika por terem sempre acreditado em mim.

Sou muito grato ao Professor Fabio Kon, meu orientador, pelo inestimável apoio e pelas oportunidades que proporcionou a mim.

Ao Professor Fernando Iazzetta, agradeço por ter estado sempre à disposição para nos auxiliar. Agradecimentos especiais também ao nosso principal colaborador e usuário, Wilson Cerqueira Ferreira.

Agradeço à CAPES pelo apoio financeiro.

Gostaria de agradecer também à Professora Nami Kobayashi por me aconselhar desde antes do mestrado, à Professora Leliane Nunes de Barros pela ajuda no início e ao Professor Marcelo Gomes de Queiroz pela participação na banca do exame de qualificação.

Agradeço aos meus colegas do GSD, principalmente ao Nelson e ao Andrei, pelas contribuições ao nosso trabalho (e pelas reuniões animadas). Também aos meus amigos e amigas da Imescos™ e aos meus grandes amigos Otávio e André, que foram muito importantes para mim nesses últimos anos.

Resumo

Ao longo da história da música, compositores sempre manifestaram interesse pelas mais recentes descobertas científicas e, freqüentemente, utilizaram as tecnologias mais avançadas de seu tempo para produzir material musical. Desde meados do século XX, o uso de tecnologias computacionais para a produção e análise de música tem se tornado cada vez mais comum entre pesquisadores e compositores. Nos últimos anos, o uso de tecnologias de redes e de sistemas distribuídos na Computação Musical se tornou um objeto natural de pesquisa.

Nesta dissertação, investigamos o uso da tecnologia de agentes móveis para a criação e execução de música em ambientes computacionais distribuídos. Mostramos que esta tecnologia tem o potencial de promover novas formas de composição, distribuição e performance musicais.

Definimos o conceito de *agentes móveis musicais*, que são agentes móveis que participam de um ambiente musical distribuído. A partir disso, especificamos e implementamos o sistema Andante, uma infra-estrutura de software de código aberto para a construção de aplicações distribuídas de composição e performance musical baseadas em agentes móveis musicais. Usando essa infra-estrutura, construímos duas aplicações que foram utilizadas por um compositor colaborador.

Abstract

Across the Centuries, musicians have always had interest in the latest scientific achievements and have used the latest technologies of their time to produce musical material. Since the mid-20th Century, the use of computing technology for music production and analysis has been increasingly common among music researchers and composers. Continuing this trend, more recently, the use of network technologies in the field of Computer Music turned out to be a natural research goal.

In this work, we investigate the use of mobile agent technology for the creation and performance of music within a distributed computing environment. We show that this technology has the potential to foster new ways of composing, distributing, and performing music.

We define the concept of *mobile musical agents*, which are mobile agents that integrate a distributed musical environment. Using this idea, we designed and implemented the Andante system, an open-source infrastructure for the construction of distributed applications for music composition and performance based on mobile musical agents. We also built two sample applications on top of this infrastructure that were used by a composer.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Um agente móvel é um programa de computador capaz de migrar de uma máquina para outra através de uma rede. No processo de migração, o agente suspende sua execução na máquina em que se encontra, gera uma representação do seu estado interno, tem seu código e estado transferidos através da rede para uma outra máquina e, finalmente, continua a execução com o estado que possuía no momento da suspensão [Kotz and Gray, 1999]. Além dessa característica, um agente tem uma certa autonomia, podendo reagir a mudanças no ambiente onde está sendo executado e decidir por si só o que fazer e quando migrar para outra máquina. Os agentes móveis possuem semelhanças com os difundidos Applets Java [Applets, sítio]. Porém, diferentemente de um agente móvel, a mobilidade de um Applet se limita ao momento em que um usuário o requisita e, conseqüentemente, sua execução ocorre inteiramente em uma mesma máquina.

Nesta dissertação, mostramos como aplicamos o conceito e a tecnologia de agentes móveis para idealizar e implementar um ambiente musical computacional.

1.1 Motivação

Compositores têm sempre acompanhado as descobertas e inovações da ciência para criar novas formas de se fazer música. A própria música tradicional foi sendo transformada à medida que novos instrumentos, ou novas formas de se produzir som, foram sendo explorados.

Houve, especialmente durante o século XX, uma intensificação dessa relação entre música e

ciência, principalmente a partir da década de 50, com a música eletroacústica e o uso de computadores. Mais recentemente, devido ao surpreendente avanço da computação, passou a ser possível equipar computadores pessoais com dispositivos relativamente baratos para sintetizar e reproduzir som de alta qualidade. Neste período, desenvolveu-se um vasto leque de técnicas e modelos para composição e execução de música usando computadores [Miranda, 2001, Roads, 1996]. As tecnologias de rede e principalmente a Internet também trouxeram novas possibilidades para produção de música [Kon and Iazzetta, 1998].

Seguindo essa tendência, idealizamos o projeto *Andante*. O conceito de agentes móveis, introduzido no início da década de 90 [Johansen et al., 1994], trouxe um novo paradigma para construção de sistemas computacionais distribuídos e móveis. A partir do final da mesma década, trabalhos concretos têm surgido, já existem várias plataformas de agentes móveis implementadas [Johansen et al., 1995, Johansen et al., 2002, Gray et al., 1998, Lange and Oshima, 1998b, Grasshopper, sítio]. O desenvolvimento do potencial do modelo de agentes móveis é então recente, por isso tivemos interesse em investigar as aplicações do conceito de agentes móveis na criação de novas formas de composição e performance musical. Definimos o conceito de *agentes móveis musicais*, que, em poucas palavras, são agentes móveis que participam de uma atividade de produção musical.

1.2 O Projeto Andante

O sistema Andante fornece uma infra-estrutura de software de código aberto que permite a construção de aplicações distribuídas baseadas em agentes móveis musicais para criação, distribuição e execução de música. Usando o Andante, programadores podem implementar seus próprios agentes ou utilizar os agentes básicos da plataforma para construir tais aplicações.

Para implementarmos a infra-estrutura, apresentamos antes um modelo de ambiente musical onde agentes móveis interagem. Fizemos uma analogia onde um agente representa um músico e uma máquina da rede representa um palco. Um agente no nosso ambiente, no entanto, pode interromper a sua performance numa máquina, instantaneamente se transportar para outra e lá continuar sua performance.

Mas o projeto Andante pretende ser mais que um sistema computacional. Realizamos colaborações com músicos e pesquisadores visando o desenvolvimento de obras artísticas distribuídas utilizando essa infra-estrutura. Esperamos com isso criar uma comunidade onde artistas e pesquisadores colaboram com idéias musicais, agentes móveis e desenvolvimento da infra-estrutura. Para tanto, desenvolvemos um sítio na Internet [Andante, sítio].

Neste trabalho, especificamos e implementamos uma versão inicial dessa infra-estrutura, bem como duas aplicações construídas a partir dela.

1.3 Computação Musical

A Computação Musical é uma área de pesquisa multidisciplinar e muito abrangente. Aqui nos limitamos a explorar três importantes sub-áreas onde o conceito de agentes móveis musicais pode ser aplicado: composição algorítmica, sistemas musicais interativos e música na Internet.

1.3.1 Composição Algorítmica

Há séculos, compositores trabalham com a idéia de que certos aspectos da composição musical podem ser formalizados em procedimentos bem definidos. A composição algorítmica consiste basicamente no uso de processos formais para a criação de música [Roads, 1996, Loy, 1989, Hiller, 1970].

O método de Guido d'Arezzo para compor cantos como acompanhamentos de textos foi desenvolvido por volta do ano de 1026; é o registro mais antigo de uma formalização nesse sentido [Roads, 1996]. Neste método, para criar uma melodia a partir de um texto, cada sílaba do texto é associada a uma nota de acordo com a vogal da sílaba e com uma tabela que relaciona notas e vogais.

Até o aparecimento dos computadores, muitos outros métodos foram desenvolvidos. Nos motetos isorrítmicos, compostos por Guillame de Machaut e outros entre os anos de 1300 e 1450, melodias são encaixadas em padrões rítmicos recorrentes. No século, XV surgiram técnicas que utilizavam inversões de intervalo e retrogradações de seqüências de notas.

Um exemplo de algoritmo de composição anterior ao século XX muito conhecido é o *Musikalisches Würfelspiel* (Jogo de Dados Musical) de Mozart. Trata-se de um jogo onde um minueto é

composto juntando-se pequenos fragmentos musicais pré-definidos [Roads, 1996]. Esses fragmentos são escolhidos ao acaso jogando dados e consultando uma tabela que relaciona cada possível resultado dos dados com vários fragmentos. O fragmento escolhido depende também do número de jogadas já realizadas.

No século XX, ainda antes do computador, a tendência era o uso de procedimentos matemáticos e estatísticos na composição e também de equipamentos eletrônicos. O serialismo, por exemplo, ao tentar controlar todos os parâmetros musicais possíveis, poderia ser considerado um processo de composição algorítmico.

O uso de computadores a partir da década de 50, como ocorreu com diversas áreas, trouxe novas possibilidades à composição algorítmica. É importante notar que a aplicação de computadores na música acompanhou a evolução dos computadores praticamente desde a sua criação. Na verdade, pela própria característica de automação da composição algorítmica, o uso de máquinas anteriores ao computador já era comum. Na metade do século XIX, Ada Lovelace fez o seguinte comentário sobre o computador mecânico de Babbage (*Analytical Engine*) [Roads, 1996].

> *"The operating mechanism [of the Analytical Engine] ... might act upon other things besides number, were* [sic] *objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the Engine. Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the Engine might compose and elaborate scientific pieces of music of any degree of complexity or extent."*

Lejaren Hiller foi um pioneiro na computação musical e a composição algorítmica era um dos seus interesses principais. Em meados da década de 1950, quando ele iniciou seus experimentos, era comum os grandes centros de pesquisa projetarem e construírem as suas próprias máquinas. Entre 1955 e 1956, Hiller e Leonard Isaacson utilizaram o computador Illiac da Universidade de Illinois em Urbana-Champaign para criar a que é normalmente considerada a primeira composição gerada por um computador: a *Illiac Suite* para quarteto de corda. O papel do computador foi

gerar a partitura, que foi traduzida para notação musical tradicional. Em julho de 1956, foi realizado um concerto onde trechos dessa obra foram executados.

Outro pioneiro na área foi Iannis Xenakis, que escreveu o programa de composição *Stochastic Music Program* (SMP) [Xenakis, 1971]. Assim como na *Illiac Suite*, a idéia original era que a música resultante do uso do SMP fosse executada por instrumentos tradicionais[1]. O SMP utiliza modelos estocásticos construídos originalmente para descrever o comportamento de moléculas em gases. A música no SMP é modelada como uma seqüência de blocos, onde cada bloco tem uma duração e uma densidade de notas. O usuário interage com o programa definindo parâmetros globais da música (como, por exemplo, duração média dos blocos, densidades mínima e máxima dos blocos e parâmetros de mudança de timbre). Esse esquema é reflexo da visão de Xenakis da música do século XX, onde o desenho global da obra é mais importante do que as suas pequenas formas. Em relação a esse fato, é importante destacar que Xenakis já possuía trabalhos nessa linha antes mesmo de usar computadores. Sua obra *Metastasis*, por exemplo, construída a partir de fórmulas estocásticas trabalhadas à mão, foi estreada em 1955.

Já nos dois exemplos pioneiros apresentados, podemos identificar duas classes distintas de algoritmos de composição: determinísticos e não-determinísticos. Outra diferença existe em relação ao que é feito com o material produzido pelos algoritmos. Ele pode ser considerado o resultado final, sem ser alterado pelo compositor, ou pode ser considerado como um material a partir do qual o compositor compõe.

Muitos outros trabalhos pioneiros além dos de Xenakis seguiram os de Hiller. Para mais informações e referências sobre a história da composição algorítmica, veja [Roads, 1996, Loy, 1989, Hiller, 1970].

Um agente é praticamente um programa de computador, portanto, a princípio, é possível implementar nele diversos algoritmos de composição. Na Seção 5.1 descreveremos uma aplicação construída sobre a infra-estrutura do Andante que utiliza um algoritmo estocástico para a composição de melodias.

[1] Apesar de posteriormente passar a ser possível conectar sintetizadores digitais ao SMP.

1.3.2 Sistemas Musicais Interativos

Embora o surgimento do computador tenha causado impacto, como foi visto na Seção 1.3.1, grande parte dos trabalhos iniciais em composição algorítmica buscava a semelhança com a música feita na época. Isto é, os sons produzidos tentavam seguir as tendências correntes, sendo o computador uma ferramenta com pouca influência estética sobre o material produzido.

Num cenário mais recente, o computador passa a inovar a forma de se produzir, executar e até de se escutar música. Os sistemas musicais interativos são uma evidência dessa nova tendência. Nesses sistemas, o computador "ouve" e reage à música que é produzida em sua volta.

Segundo Rowe [Rowe, 1993], sistemas musicais interativos são aqueles cujo comportamento muda em resposta ao recebimento de informações musicais. Essa classe de sistemas tem sido muito explorada pela música erudita na última década. Observa-se uma profusão de obras musicais onde o computador aparece com um elemento que gera música baseada na interação com outros músicos, dançarinos, atores e até mesmo com o público [Iazzetta, 2003].

Os sistemas interativos existentes se diferenciam pelo nível de interação proporcionado por cada um deles. Em relação ao modo de funcionamento, Rowe definiu três dimensões para classificá-los. O objetivo era permitir o estudo e a discussão desses sistemas.

- Em relação à recepção da informação musical, um sistema pode usar partituras armazenadas em memória e compará-las com a entrada para decidir suas ações (*score-driven*); ou pode usar outros métodos para analisar a entrada (*performance-driven*).

- Quanto à maneira de reagir à entrada, o sistema pode: utilizar a informação musical de entrada para gerar a saída (*transformative methods*); utilizar apenas algoritmos e material musical elementar como escalas ou tabelas de durações (*generative algorithms*); ou reproduzir trechos de músicas armazenados em memória (*sequenced techniques*).

- E finalmente, de acordo com a funcionalidade, o sistema pode servir como um instrumento tocado pelo usuário (*instrument paradigm*); ou como um instrumentista que acompanha o usuário (*player paradigm*).

A autonomia dos agentes e a sua capacidade de interagir com o ambiente no qual está inserido

e reagir a estímulos dele provenientes faz dos agentes musicais um mecanismo particularmente apropriado para a construção de sistemas musicais interativos.

1.3.3 Música na Internet

Uma conseqüência natural do crescimento expressivo da Internet e do acesso cada vez mais comum de computadores pessoais a recursos de multimídia foi o surgimento do interesse em sistemas multimídia distribuídos. Mas devido à característica estritamente temporal da música e o grande volume de dados de que sua representação digital necessita, as dificuldades envolvidas em sua implementação são numerosas [Kon and Iazzetta, 1998].

O sistema *NetSound* [Casey and Smaragdis, 1996] tenta resolver o problema da largura de banda utilizando uma representação alternativa do som. Em vez de enviar o som digitalizado pela rede, o *NetSound* envia uma descrição matemática das ondas do som, que é reconstruído com alta qualidade na máquina de destino. Em ambientes onde as exigências musicais não são tão estritas, também são comuns sistemas como o *Real Audio* [Real Audio, sítio], que utilizam áudio comprimido com qualidade reduzida.

O projeto *Global Visual Music* [Global Visual Music, sítio] está desenvolvendo uma infra-estrutura para a realização de performances com improvisação através da distribuição de conteúdo multimídia em tempo real para vários pontos.

O sistema *Musical Agents* [Fonseka, 2000] gera música de forma distribuída utilizando a colaboração entre agentes que executam algoritmos de composição. Nele, os usuários escrevem os algoritmos numa linguagem de script que é entendida pelos agentes. A geração de som é baseada no protocolo MIDI. Cada agente é executado em uma máquina da rede e executa um script dado por um usuário. Os agentes participantes comunicam-se entre si pela rede e todos recebem a música resultante.

Os *Musical Agents* utilizam um modelo semelhante ao do sistema DASE (Distributed Audio Sequencer) [DASE, sítio]. Nesse sistema, cada usuário executa uma aplicação cliente para compor um pequeno trecho de música em *loop*. Cada trecho é enviado a um servidor central, onde todos são tocados juntos para compor uma música.

À medida que os problemas de largura de banda e atraso da rede são equacionados, os sistemas de música na Internet caminham em direção à interação com o usuário e à possibilidade de colaboração entre diversos usuários.

1.4 Organização do texto

Nesta dissertação, apresentamos o modelo de agentes móveis musicais e descrevemos uma implementação desse modelo. Levando em consideração que temos um grupo heterogêneo de leitores, já que este texto é voltado para a comunidade de computação musical, selecionamos e organizamos o conteúdo da seguinte maneira.

No Capítulo 2, introduzimos o conceito de código móvel para, na Seção 2.2.4, discutirmos o paradigma de agentes móveis, que é central para o nosso trabalho. Apoiado na Seção 2.2.4, o Capítulo 3 apresenta o nosso modelo de agentes móveis musicais e descreve alguns exemplos de sistemas baseados nesse modelo.

A descrição da arquitetura da infra-estrutura do sistema Andante, bem como da sua implementação, é feita no Capítulo 4. Na Seção 4.2 também analisamos as tecnologias empregadas. O Capítulo 5 demonstra a viabilidade da arquitetura apresentando as duas aplicações construídas sobre a infra-estrutura: *NoiseWeaver* e *Maestro*.

Concluímos com o Capítulo 6, onde discutimos possíveis projetos baseados no sistema Andante.

Capítulo 2

Código Móvel

Acompanhando o rápido avanço das tecnologias relacionadas à computação, as redes de computadores se tornam cada vez maiores e mais presentes. A Internet, exemplo mais evidente desse fenômeno, difundiu o uso das tecnologias de rede até mesmo para o público leigo e com isso possibilitou, ou pelo menos impulsionou, a criação de novas aplicações e serviços como, por exemplo, comércio eletrônico e distribuição de áudio.

Mas tal fenômeno não se restringiu apenas à Internet. Áreas de pesquisa como computação ubíqua [Weiser, 1991] e móvel [Mateus and Loureiro, 1998, Stojmenovic, 2002] são, ao mesmo tempo, causa e conseqüência disso. A computação ubíqua estuda maneiras de tornar a computação permeada no ambiente e invisível ao usuário, geralmente através do uso de um número elevado de pequenos dispositivos conectados em rede. A computação móvel investiga aplicações de dispositivos portáteis que se utilizam de tecnologias de rede sem fio.

Assim, toda essa evolução traz inúmeros desafios e novas perspectivas para a área de sistemas distribuídos. Um dos principais problemas é a *escalabilidade*, ou seja, como lidar com redes formadas por um número cada vez maior de máquinas. As soluções desenhadas para pequenas redes locais normalmente não são praticáveis num ambiente como a Internet. As redes sem fio também apresentam dificuldades [Forman and Zahorjan, 1994] principalmente devido ao fato dos nós da rede poderem se mover e se desconectar freqüentemente. O surgimento de uma grande variedade de aplicações e serviços destinados a diferentes setores da sociedade também trazem a

necessidade de sistemas personalizáveis, flexíveis e extensíveis.

Esses problemas representam tópicos de pesquisa em aberto, muitos paradigmas de construção de sistemas distribuídos foram desenvolvidos em função deles. Neste capítulo discutiremos uma classe de tais modelos denominada *código móvel*, tomando como base uma simplificação [Tanenbaum and Steen, 2002] do arcabouço teórico apresentado por Fuggetta, Picco e Vigna [Fuggetta et al., 1998]. Nosso o objetivo é introduzir alguns modelos de código móvel.

A mobilidade se refere à capacidade de se mudar a localização do código, ou parte dele, durante a sua execução [Carzaniga et al., 1997]. Usa-se também o termo *migração de código* para esse procedimento.

Inicialmente, a principal motivação para a mobilidade do código em sistemas distribuídos era o balanceamento de carga. Nesse processo, um programa em execução é inteiramente movido de uma máquina sobrecarregada para continuar sendo executado numa máquina menos utilizada. A carga de uma máquina geralmente é uma medida proporcional ao nível de uso do seu processador.

Mas existem casos em que o tempo e recursos gastos com comunicação são mais relevantes para o desempenho do sistema. Por exemplo, considere um sistema cliente/servidor onde o servidor armazena um banco de dados imenso e um cliente faz diversas consultas ao banco para realizar um cálculo. Essa situação possivelmente ocasionaria um tráfego de dados pela rede grande o suficiente para comprometer o desempenho do sistema. Uma alternativa seria executar no servidor a parte do código do cliente que efetua o cálculo. Nesse caso, a transferência dos dados do banco não ocorreria pela rede e sim dentro do servidor apenas. Somente o código do cliente a ser executado no servidor e o resultado final consumiriam os recursos da rede. Do ponto de vista da rede, trata-se de mover a computação para perto dos dados.

A mesma estratégia pode ser usada no sentido inverso, ou seja, pode ser vantajoso mover parte do servidor para o cliente. Em muitos sistemas cliente/servidor que utilizam banco de dados, existe uma interface no cliente onde o usuário preenche um formulário que é enviado ao servidor. As informações contidas no formulário precisam ser validadas pelo servidor antes de serem processadas de fato. Portanto, erros ou inconsistências no formulário resultariam em uma troca de muitas pequenas mensagens pela rede. Em vez disso, o servidor poderia enviar ao cliente

a parte do seu código que valida o formulário. O formulário seria enviado pela rede somente se fosse válido.

Além de melhorar o desempenho geral do sistema, a migração de código pode prover mais flexibilidade à construção de sistemas distribuídos. Nos sistemas tradicionais, as funcionalidades e a localização de cada componente são definidas durante a implementação. Com a mobilidade, por outro lado, sistemas distribuídos podem ser configurados em tempo de execução. Uma máquina cliente não precisaria de todos os componentes do sistema pré-instalados, apenas dos que fossem utilizados. Ademais, a atualização ou mudança de implementação de componentes individuais pode ser facilitada.

Entretanto, o suporte a código móvel num sistema pode trazer desvantagens. Uma delas é justamente a necessidade de sistemas adicionais para o suporte. Isso pode trazer conseqüências no desempenho individual de cada máquina pelo custo adicional imposto e também na complexidade do sistema. Mas o maior problema é a questão da segurança [Vigna, 1998]. No caso geral, não é uma boa idéia confiar cegamente em programas vindos de outras máquinas. É necessária alguma espécie de autenticação para garantir a confiabilidade do código ou uma restrição de acesso aos recursos da própria máquina para que impedir que um código possivelmente malicioso cause danos. Por outro lado, a máquina de destino também pode agir indevidamente, por exemplo, alterando o resultado da computação, ou ainda, se o resultado representar informações confidenciais, elas podem ser roubadas. Portanto é desejável que o código e dados enviados a uma máquina também sejam protegidos.

2.1 Classificação

Em sistemas distribuídos convencionais, a comunicação consiste na troca de dados entre programas. Em sistemas que utilizam a migração de código, programas são movidos entre máquinas com o intuito de serem executados no destino. Tal afirmação é uma generalização do conceito de código móvel. Na prática, os sistemas implementam diferentes modelos para obter a mobilidade em diversos níveis.

Para organizar a pesquisa na área de código móvel, Fuggetta, Picco e Vigna propuseram um

arcabouço teórico [Fuggetta et al., 1998] que identifica os diversos mecanismos de mobilidade. Isso permite analisar tanto os paradigmas aplicados na construção de sistemas distribuídos baseados em código móvel como os próprios sistemas implementados. A discussão que segue é uma adaptação da parte do arcabouço que permite classificar as tecnologias implementadas.

Um programa é composto por três segmentos. O segmento de *código* é o conjunto de instruções que formam o programa em si. O segmento de *recursos* contém dados e referências para recursos externos utilizados pelo programa como arquivos, dispositivos e outros programas. O segmento de *execução* armazena o estado corrente do programa em execução.

Tipos de Migração

A forma mais simples de mobilidade é a chamada *migração fraca*, onde somente o segmento de código é transportado com, possivelmente, alguns dados de inicialização. O atrativo dessa alternativa é justamente a simplicidade.

Em sistemas distribuídos heterogêneos, isto é, constituídos de máquinas de diferentes plataformas, as dificuldades de se executar o mesmo código em diversas máquinas são numerosas. Ainda mais desafiador é transferir o estado de execução do programa em ambientes heterogêneos. No extremo oposto em relação à mobilidade fraca, é isso que a *migração forte*, provê. Neste modelo, um programa pode ser interrompido numa máquina e continuar a execução em outra a partir do ponto em que foi interrompido. É uma alternativa muito mais poderosa mas de implementação muito mais complexa.

Além de poder ser forte ou fraca, a migração pode ser *proativa* ou *reativa*. Na migração proativa, o próprio programa decide realizar a migração. Na migração reativa, a iniciativa vem de um outro programa que, por exemplo, pode estar sendo executado na máquina de destino.

Em relação ainda ao programa que toma a iniciativa, a migração pode ser *síncrona* ou *assíncrona*, dependendo se o programa espera ou não pela execução do código que migrou.

A Figura 2.1 mostra as alternativas de tipos de migração que apresentamos.

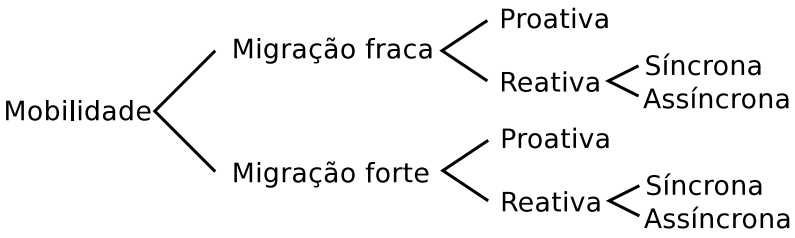

Figura 2.1: Tipos de migração

Migração de Recursos

Um problema que ainda não discutimos é o da migração do segmento de recursos. Em alguns casos, os recursos não podem ser simplesmente movidos junto com os outros segmentos. Por exemplo, considere um programa que utiliza uma porta TCP para se comunicar com outros programas. O segmento de recursos desse programa contém uma referência para esse recurso. Se o programa migrar para outra máquina, o recurso não será mais válido. O programa precisará obter uma referência para uma nova porta. Por outro lado, se o recurso for um arquivo remoto identificado por uma URL, ele poderá ser transferido sem modificações.

Para entender as implicações da migração sobre o segmento de recursos, Fuggetta et al. fazem uma análise completa de questões como as que acabamos de mencionar [Fuggetta et al., 1998].

2.2 Paradigmas

Paradigmas de desenvolvimento de sistemas representam arquiteturas de software com características semelhantes. Eles descrevem o desenho geral de uma arquitetura e as interações entre seus componentes de forma que isso possa ser mapeado para uma arquitetura real. São, portanto, independentes das tecnologias no sentido de que uma mesma tecnologia pode ser empregada na implementação de paradigmas diferentes.

Modelos tradicionais de construção de software nem sempre funcionam bem em sistemas distribuídos, principalmente nos que se utilizam da mobilidade de código. Isso se deve ao fato das

interações entre componentes que residem no mesmo espaço de endereçamento de uma máquina serem distintas das interações entre componentes em diferentes máquinas da rede. Com componentes distribuídos, o desenvolvedor precisa estar ciente da latência, usar diferentes modelos de memória e considerar mais questões de concorrência e falhas [Waldo et al., 1997].

Ainda utilizando o arcabouço de Fuggetta et al., faremos uma introdução de alguns dos principais paradigmas de código móvel: *avaliação remota*, *código sob demanda* e *agentes móveis*, sendo este último de maior importância para nossa pesquisa. Além desses, discutiremos o paradigma *cliente/servidor* para efeito de comparação e, como um paradigma relacionado a esse e que também interessa à nossa pesquisa, o modelo de *objetos remotos*.

Definições

Em nossa descrição, a arquitetura de um sistema distribuído é formada por três elementos: *componentes*, *interações* e *locais*.

Os *componentes* podem ser: *de tarefa*, que representa o *know-how*, isto é, as instruções para a execução de uma tarefa computacional; *de recursos*, que representa dados ou dispositivos utilizados durante a execução de uma tarefa; e *de computação*, que é um elemento capaz de executar uma tarefa, dadas as instruções. O código é representado pelos componentes de tarefa e de computação. *Interações* são eventos que envolvem dois ou mais componentes, por exemplo, uma troca de mensagens entre dois componentes de computação. *Locais* representam a noção intuitiva de locais físicos, eles hospedam componentes e possibilitam a execução de componentes de computação.

Interações entre componentes localizados em um mesmo local são consideradas menos dispendiosas do que interações entre locais diferentes. Ademais, uma tarefa pode ser executada somente se as instruções de como executá-la, os recursos necessários e o componente de computação responsável por ela estiverem todos no mesmo local.

Descreveremos os paradigmas em função do padrão de interações que definem as localizações e a coordenação entre os componentes necessários a uma tarefa. Consideraremos um cenário onde um componente de computação A, localizado no local L_A, precisa do resultado de uma tarefa. Suporemos também a existência de um outro componente de computação B e um outro local L_B,

Paradigma	Antes		Depois	
	L_A	L_B	L_A	L_B
Cliente/Servidor	A	instruções recursos B	A	instruções recursos B
Avaliação Remota	instruções A	recursos B	A	*instruções* recursos B
Código sob demanda	recursos A	instruções B	recursos *instruções* A	B
Agentes móveis	instruções A	recursos	–	*instruções* recursos *A*

Tabela 2.1: Localização dos componentes antes e depois da realização da tarefa

que estarão envolvidos na realização da tarefa. Para cada paradigma, mostraremos a localização dos componentes antes e depois da realização da tarefa, qual componente é responsável pela execução das instruções e o local onde a tarefa é de fato realizada. A Tabela 2.2 sintetiza essa discussão.

2.2.1 Cliente/Servidor

O modelo cliente/servidor é bastante conhecido e utilizado. Na sua forma básica, um componente de computação B localizado em um local L_B age como um *servidor* fornecendo um conjunto de serviços. As *instruções* e os *recursos* necessários para os serviços também se encontram em L_B. Um componente A localizado em L_A, o *cliente*, faz a requisição de um serviço através de uma *interação* com o componente B. Em resposta ao pedido, B realiza a tarefa executando as instruções e acessando os recursos. Em muitos casos, a tarefa gera um resultado que é devolvido para A.

Objetos Remotos

Uma forma de comunicação em sistemas distribuídos muito empregada é a *chamada de procedimento remota* (RPC - *Remote Procedure Call*). Introduzida por Birrell e Nel-

son [Birrell and Nelson, 1984], ela permite que programas executem código localizado em outras máquinas. Podemos associar esse método ao modelo cliente/servidor: o componente A, localizado em L_A, informa a B, localizado em L_B, o nome do procedimento desejado e os parâmetros de entrada. Os procedimentos que B pode executar são representados pelas instruções localizadas em L_B. B então executa a tarefa e devolve o resultado a A. Para o programador da aplicação que faz a chamada, a troca de mensagens pela rede não é visível, tudo se passa como se fosse realizada uma chamada de procedimento convencional.

A tecnologia de objetos tem sido aplicada com sucesso no desenvolvimento de sistemas computacionais em geral. Um objeto encapsula um comportamento e um estado, escondendo o seu funcionamento interno por meio de uma interface de uso bem definida. A interface dá acesso aos *métodos* de um objeto, esses métodos representam o comportamento do objeto e somente eles podem alterar o seu estado. Essa abordagem permite que objetos possam ser facilmente trocados ou modificados, contanto que a interface permaneça a mesma.

Na prática, os métodos são semelhantes a procedimentos, isto é, são comandos que podem receber parâmetros de entrada e possivelmente devolvem um resultado. Portanto, com o desenvolvimento dos mecanismos de RPC, os desenvolvedores perceberam que o mesmo princípio poderia ser utilizado com objetos. Disso surge então a tecnologia de *objetos remotos*, cuja característica básica é a chamada remota de métodos.

O componente A em L_A informa a B em L_B o objeto alvo, o nome do método e os parâmetros. A implementação do objeto também se localiza em L_B, e B se encarrega de acionar o método com os parâmetros dados e de devolver a A uma eventual resposta.

Assim como no caso da RPC, o processo todo de comunicação é transparente ao programador, isto é, do ponto de vista do desenvolvedor usuário do objeto, a invocação de um objeto remoto é feita da mesma maneira que é feita a de um objeto local. Para que isso seja possível, deve existir em L_A um representante do objeto em questão. Esse representante possui a mesma interface do objeto original, porém, não contém as instruções necessárias para a realização da tarefa. O seu papel é repassar a requisição para o objeto em L_B e posteriormente repassar o resultado a A.

2.2.2 Avaliação Remota

Neste paradigma, o componente A em L_A tem as instruções necessárias para a realizar a tarefa mas não possui os recursos, que estão em um outro local L_B. O componente A então envia as instruções para o componente B que também se localiza em B. Tendo acesso aos recursos, B executa as instruções e devolve o resultado a A.

Um exemplo de tecnologia que implementa esse paradigma é o das impressoras PostScript. O componente A pode ser um editor de texto, as instruções o arquivo na linguagem PostScript a ser impresso, os recursos os dispositivos da impressora capazes de imprimir o arquivo e B o programa da impressora que interpreta os arquivos recebidos.

A avaliação remota pode ser considerada um caso especial do modelo cliente/servidor. De fato, o servidor pode oferecer um serviço de execução de código que recebe um fragmento de código como parâmetro. Mas há uma distinção importante, que é uma característica dos paradigmas de código móvel: a capacidade de estender a funcionalidade do servidor dinamicamente, isto é, depois que ele foi construído e executado.

2.2.3 Código sob Demanda

Neste modelo, o componente A tem acesso aos recursos necessários, estando todos localizados em L_A. Porém, as instruções necessárias para manipular os recursos não se encontram em L_A. Tais instruções podem ser encontradas em L_B, onde também se encontra B. Portanto, através de uma interação, A pede a B que lhe envie as instruções. Ao receber as instruções, A pode executá-las.

Uma tecnologia muito utilizada que aplica este paradigma são os Applets [Applets, sítio]. Um Applet é um programa escrito em Java que pode ser integrado a uma página HTML. O programa em si reside em um servidor de páginas (B), quando um cliente utilizando um navegador (A) requisita a página, o Applet é transferido para a máquina do cliente (L_A), onde é executado.

2.2.4 Agentes Móveis

Um agente móvel é representado pelo componente A, que possui as instruções necessárias. Ambos se encontram inicialmente em L_A, mas parte dos recursos localizam-se em L_B. Para realizar a

tarefa, A migra para L_B levando consigo as instruções e possivelmente o resultado de alguma computação realizada anteriormente. Estando em L_B, A pode completar ou continuar o serviço usando os recursos de lá.

O conceito de *agente* em computação não tem uma definição única. Em geral, cada autor apresenta uma definição de acordo com sua necessidade, já que o termo é utilizado em sub-áreas diversas e portanto com abordagens diferentes, como em Inteligência Artificial [Hayes, 1999, Jennings and Wooldridge, 1998, Wooldridge, 1998, Russell and Norvig, 2003] e Sistemas Distribuídos. Franklin e Graesser propuseram uma definição ampla discutindo várias outras definições e construindo uma taxonomia de agentes [Franklin and Graesser, 1996].

Trabalharemos com uma definição baseada na de Lange e Oshima [Lange and Oshima, 1998a] que, além de ser semelhante à de Franklin e Graesser, trata-se de uma definição voltada à área de agentes móveis.

Um agente é um programa que

- localiza-se em um ambiente de execução;
- possui obrigatoriamente as seguintes propriedades:
  - *reação*: percebe e reage a mudanças no ambiente;
  - *autonomia*: tem controle sobre as próprias ações;
  - *proatividade*: realiza ações que alteram o ambiente (não apenas reage a alterações);
- e possui uma ou mais das seguintes propriedades opcionais:
  - *comunicação*: pode se comunicar com outros programas;
  - *mobilidade*: pode migrar de um ambiente para outro;
  - *adaptatividade*: capaz de aprender.

Portanto, um agente que possui pelo menos a propriedade opcional de mobilidade é um *agente móvel*.

No modelo de agentes móveis, temos o *sistema de agentes* que fornece o ambiente de execução. Esse sistema também é responsável por fornecer serviços de migração, criação e localização de agentes.

Vantagens e Desvantagens

Além do benefício conceitual de possibilitar abordagens diferentes para problemas conhecidos, o paradigma de agentes móveis pode trazer uma série de vantagens técnicas em relação aos modelos mencionados anteriormente, especialmente ao cliente/servidor.

Lange e Oshima apresentam sete vantagens [Lange and Oshima, 1999, Lange and Oshima, 1998a].

1. *Redução do tráfego na rede*: sistemas distribuídos utilizam protocolos de comunicação que geralmente envolvem muitas trocas de mensagens pela rede. Com agentes móveis, que são um paradigma de código móvel, ocorre a migração do código para a máquina de destino e, dessa forma, as interações são realizadas localmente. O tráfego é reduzido às migrações dos agentes.

2. *Eliminação da latência*: pelo mesmo motivo acima, isto é, por interagirem localmente, a latência das comunicações pela rede deixa de ser um problema.

3. *Encapsulamento de protocolos*: os protocolos de comunicação permitem a troca de dados entre os componentes de um sistema distribuído. À medida que um protocolo evolui, em geral por motivos de segurança, eficiência ou flexibilidade, os componentes que implementam o protocolo precisam ser atualizados. Isso pode ser trabalhoso em algumas situações. No caso dos agentes móveis, e também dos modelos de código móvel, um trecho de código poderia migrar para um determinado componente e estabelecer um canal de comunicação com o componente de origem. Como o código que implementa a comunicação de rede pertence ao componente de origem, o componente de destino não precisa conhecer o protocolo que será utilizado na rede.

4. *Execução autônoma e assíncrona*: ao migrar para uma máquina, um agente móvel pode

se tornar independente da aplicação que o criou, podendo executar uma tarefa de forma autônoma e assíncrona. Não é necessário manter conexões abertas entre as duas partes e, por isso, sistemas que dependem de conexões frágeis, como os sistemas móveis, podem se beneficiar dessa característica.

5. *Adaptação dinâmica*: agentes móveis podem receber informações sobre o ambiente onde está sendo executado e reagir a elas.

6. *Heterogeneidade*: as redes de computadores geralmente são heterogêneas, tanto em relação ao hardware como ao software. Os sistemas de agentes móveis são projetados de forma a lidar com essa heterogeneidade fazendo com que o agente seja dependente apenas de um mesmo ambiente de execução que é implementado em cada combinação de hardware e software. Portanto, ao utilizar agentes móveis, um sistema distribuído ganha uma forma de integrar sistemas heterogêneos.

7. *Tolerância a falhas*: as características de execução autônoma e assíncrona e adaptação dinâmica favorecem a construção de sistemas tolerantes a falhas. Em especial, elas permitem que um agente tome uma atitude em situações desfavoráveis. Por exemplo, se uma máquina precisa ser desligada, os agentes nessa máquina podem ser avisados e com isso migrar para outra máquina da rede para que possam continuar o seu trabalho.

Os agentes móveis também trazem as desvantagens do código móvel mencionadas anteriormente. Vigna ainda apresenta dez dificuldades associadas exclusivamente a agentes móveis, a maioria relacionada à segurança e ao excesso de complexidade, e defende que os paradigmas de código móvel mais simples oferecem soluções melhores [Vigna, 2004].

Em relação à segurança, Jansen e Karygiannis classificam diversas possíveis ameaças a que um sistema que utiliza agentes móveis está sujeito [Jansen and Karygiannis, 1999]. Os ataques podem ser divididos em quatro categorias: um agente atacando o sistema de agentes em que se encontra; o sistema atacando um agente que está hospedando; um agente atacando outro agente no mesmo sistema; e outra entidade externa (agente em outro sistema, outro sistema ou outro programa) atacando o sistema. E podem ser de três tipos básicos: exposição de informação,

negação de serviço e corrupção de informação.

Por exemplo, um agente móvel pode atacar um sistema consumindo uma quantidade excessiva dos recursos da máquina. Esse ataque pode ocorrer intencionalmente explorando vulnerabilidades do sistema, mas também pode ocorrer sem intenção por um erro de programação do agente. O ataque é caracterizado como negação de serviço. Para mais detalhes sobre este e os outros possíveis ataques, bem como muitas outras questões sobre segurança em agentes móveis, consulte [Jansen and Karygiannis, 1999].

2.3 Conclusão

Fizemos uma breve introdução do conceito de código móvel com foco voltado ao modelo de agentes móveis. Apresentamos alguns benefícios e dificuldades associados ao uso dessa tecnologia. Nosso interesse, entretanto, diverge um pouco dessas questões por estarmos lidando com uma aplicação inédita: os agentes móveis musicais, que serão discutidos no Capítulo 3.

Capítulo 3

Agentes Móveis Musicais

Neste capítulo introduzimos e discutimos o conceito central da nossa pesquisa. Desenvolvemos a definição de um *agente móvel musical* e construímos um modelo abstrato de sistema musical que se utiliza dele. Em seguida apresentamos possíveis implementações reais desse modelo.

3.1 Definição

Um agente móvel musical é um agente móvel que participa de uma atividade de geração ou processamento de música. Ele pode fazer isso realizando uma ou mais das seguintes ações.

Encapsular um algoritmo de composição: assim como um programa de computador, um agente pode implementar um algoritmo de composição [Miranda, 2001, Roads, 1996, Rowe, 1993]. Esse algoritmo pode precisar de dados de entrada que podem ser levados com o agente, o que permite que ele gere música autonomamente.

Interagir e trocar informações com outros agentes musicais: numa analogia com músicos tocando num palco real, agentes numa mesma máquina podem interagir e comunicar-se uns com os outros trocando informações musicais.

Interagir com músicos humanos: um agente pode receber comandos ou sons gravados por um músico humano. Os comandos podem ser, por exemplo, notas tocadas num teclado MIDI (que seriam utilizadas pelo agente para produzir música) ou novos parâmetros para o

algoritmo de composição executado pelo agente. O som de um instrumento acústico pode também ser enviado e digitalizado para um agente e processado e/ou reproduzido por ele.

Reagir a sensores: os agentes podem também receber comandos de sensores, permitindo que eles reajam a certos eventos como, por exemplo, os movimentos de uma bailarina em um palco ou a movimentação do público em um museu.

Migrar: o processo de migração pode ser acionado pelas ações descritas acima. Em outras palavras, um agente pode decidir migrar para outra máquina

- estocasticamente ou deterministicamente, baseado em um algoritmo;
- baseado na interação com outros agentes;
- baseado na interação com músicos humanos;
- em resposta a sinais de sensores.

Assim como um agente móvel normal, um agente musical pode continuar a sua performance ao chegar ao seu destino.

3.2 Modelo de Sistema de Agentes Móveis Musicais

Um agente móvel musical necessita de um ambiente de execução. Chamaremos tal ambiente de *palco*. O palco é, portanto, um lugar onde os agentes se encontram e podem realizar suas ações musicais. Um palco não corresponde necessariamente a um único ambiente de execução de agentes móveis. O palco pode ser distribuído, ou seja, representado por mais de um ambiente de execução. Numa outra possibilidade, uma máquina da rede pode conter vários palcos. Mas a situação mais esperada é a existência de diversos palcos, cada um localizado em uma máquina. Nesse cenário, é possível explorar melhor a mobilidade dos agentes.

Num mesmo palco, podem ser recebidos agentes de diferentes tipos. Embora seja desejável que interajam, os agentes num mesmo palco não precisam se comunicar uns com os outros, cada agente pode executar as suas ações independentemente.

3.3 Exemplos

Utilizando o modelo descrito e agentes móveis musicais de vários tipos, podemos construir sistemas de composição e performance musical. Vejamos alguns exemplos.

1. *Geração de música estocástica:* Nesse sistema, vários agentes encapsulam algoritmos estocásticos de geração de melodias. Um dos agentes funciona como um metrônomo, fornecendo para todos os outros agentes os tempos do compasso no momento certo. O resultado da execução desses vários agentes numa mesma máquina é a geração de música sincronizada.

2. *Performance distribuída:* Cada músico humano é representado por um ou mais agentes. Cada agente recebe a música gerada pelo músico e a reproduz em tempo real no palco em que se encontra. Dessa forma, um músico pode estar virtualmente presente em mais de um palco. Agentes desse tipo poderiam ser utilizados para a realização de uma performance distribuída: cada músico numa localidade receberia em seu palco agentes que representariam os outros músicos e também enviaria agentes que o representariam para os palcos dos outros músicos.

3. *Sistema musical interativo:* Cada agente recebe informações musicais de outro agente ou de um instrumento musical tocado por um humano. O agente então processa essas informações e gera música baseada no processamento ou repassa a informação processada para outro agente.

4. *Música distribuída:* Considere um museu ou uma sala de exibição onde se encontram vários computadores ligados em rede. Cada computador é equipado com sensores de movimento e hospeda alguns agentes. Os agentes geram uma música distribuída de uma maneira sincronizada. Um dos agentes recebe informações geradas pelos sensores de movimento, de modo que ele possa seguir pessoas que estejam andando pela sala (usando a capacidade de migrar). A sensação do ouvinte é de que parte da música o está seguindo. Da mesma maneira, é possível fazer com que uma outra parte da música sempre se afaste do ouvinte. Sons gerados pelo público (por exemplo, frases) podem ser dinamicamente incorporadas à paisagem musical gerada pelo sistema.

Compositores e pesquisadores envolvidos em música contemporânea no Brasil e no exterior já manifestaram grande interesse em estudar as possibilidades de aplicação do paradigma de agentes móveis à música. A colaboração destes músicos nestas pesquisas será essencial para vislumbramos as reais possibilidades dessa nova tecnologia.

Enquanto isso, os exemplos acima já trazem à tona diversos problemas técnicos a serem superados. Em particular, questões de *tempo real*, *latência* e *qualidade de serviço* são importantes para a geração final do som. O suporte de agentes móveis para essas questões é um assunto ainda não estudado e que pretendemos explorar no futuro. Não abordamos essas questões neste trabalho, no entanto.

Capítulo 4

O Sistema Andante

O Sistema Andante provê uma infra-estrutura de software para a construção de aplicações baseadas em agentes móveis musicais. Neste capítulo, descrevemos a implementação da infra-estrutura apresentando a sua arquitetura e as tecnologias empregadas.

4.1 Arquitetura

No Andante, os agentes executam as suas ações numa rede de computadores heterogênea. Para isso, de forma análoga ao ambiente de execução dos agentes móveis mencionado na Seção 2.2.4, os computadores dessa rede executam um programa preparado para receber os agentes. Esse programa é um componente da arquitetura, funciona como um palco onde os agentes se encontram e "tocam" música. Chamamos esse componente de *Stage* ou Palco.

O Palco oferece serviços para que os agentes possam realizar suas ações. Para, por exemplo, tocar uma melodia, um agente precisa utilizar o serviço de geração de som do Palco. O Palco, por sua vez, utiliza um outro componente da infra-estrutura para fornecer o serviço de som. Esse componente é o *Audio Device* ou Dispositivo de Áudio.

Definimos com isso três elementos fundamentais da arquitetura: os agentes móveis musicais (*Mobile Musical Agents*), o Palco (*Stage*) e o Dispositivo de Áudio (*Audio Device*). A Figura 4.1 mostra uma visão abstrata da arquitetura.

A figura ainda ilustra uma possível aplicação construída sobre a infra-estrutura. A *GUI*

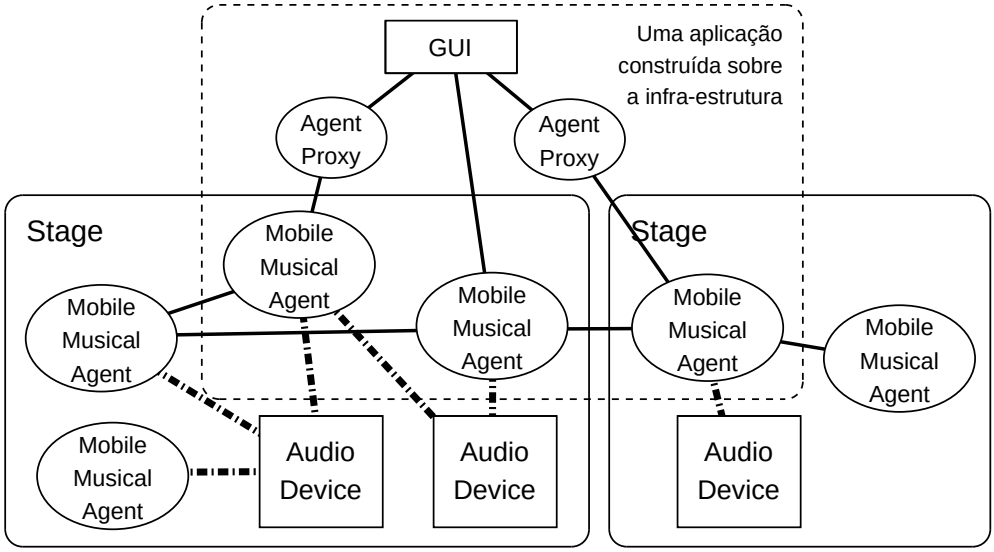

Figura 4.1: Visão geral da arquitetura

(*Graphical User Interface*) não é necessariamente parte da infra-estrutura, mas tem o importante papel de prover uma interface gráfica para interação entre agentes e usuários humanos. A aplicação se utiliza de um quarto componente da arquitetura: o Representante de Agente (*Agent Proxy*). Esse componente faz parte da aplicação e fornece transparência de localização de um ou mais agentes para a GUI. Na ocorrência de uma migração, um agente informa a sua nova localização ao seu representante, que por sua vez é responsável pela comunicação entre a GUI e o agente ou por repassar a nova localização para a GUI. A GUI também pode escolher comunicar-se diretamente com um agente ou ela própria ser representante de um ou mais agentes.

4.2 Tecnologias Utilizadas

As principais tecnologias empregadas na construção do sistema Andante foram Java, CORBA e Aglets. Descrevemos brevemente cada uma delas levando em conta a forma como foram aplicadas. Fizemos também uma tentativa de integração com o ambiente MAX/MSP, conforme veremos a seguir.

4.2.1 Java

Java [Java, sítio] é um middleware para a construção de sistemas em geral, mas com ênfase para computação distribuída. Podemos distinguir três componentes principais desse middleware extremamente abrangente.

- Uma linguagem de programação orientada a objetos [Joy et al., 2000, Arnold et al., 2000].
- Uma máquina virtual que representa uma plataforma computacional.
- Um conjunto imenso de bibliotecas e ferramentas para o desenvolvimento de aplicações.

Java possui características que favorecem especificamente o uso de código móvel, além de prover suporte para multimídia e interfaces gráficas. Sendo assim, é uma alternativa interessante para a implementação do sistema Andante, como veremos a seguir.

Independência de Plataforma. Java foi desenhado para operar em redes heterogêneas. A máquina virtual Java (JVM - *Java Virtual Machine*) permite que o mesmo código de uma aplicação Java seja executado em qualquer máquina da rede. Para isso, a aplicação na verdade não é compilada diretamente para código de máquina, que é dependente de plataforma, mas sim para uma linguagem intermediária chamada *bytecode*, que é interpretada pela máquina virtual. Sendo assim, a máquina virtual é quem precisa de uma implementação diferente para cada combinação de hardware e sistema operacional. A máquina virtual representa então uma plataforma virtual e, de uma certa maneira, torna homogêneo um sistema distribuído. Isso é claramente uma grande vantagem para o desenvolvimento de aplicações que utilizam código móvel, pois transfere o fardo da heterogeneidade dos programadores das aplicações para os programadores da máquina virtual.

A independência de plataforma é especialmente importante para o Andante porque esperamos que o sistema seja utilizado por programadores, compositores e instrumentistas Necessitamos então que o sistema possa ser executado em ambientes de hardware e software distintos, como os baseados em Linux (muito utilizado por programadores), MacOS (muito utilizado por compositores) e Windows (muito utilizado por instrumentistas).

Execução segura. Como foi feito para ser utilizado em redes como a Internet, o projeto da linguagem Java já levou em consideração algumas questões de segurança. Por exemplo, o modelo de acesso aos objetos impede que a memória seja corrompida ou acessada indevidamente. Mesmo que o *bytecode* seja alterado para tanto, a máquina virtual tem condições de barrar acessos não permitidos. Isso garante a proteção da máquina contra os numerosos tipos de ataques que exploram essa possibilidade. Portanto, em termos de acesso a informações privadas da máquina, é razoavelmente seguro receber agentes móveis de fontes não certificadas.

Carregamento dinâmico de classes. Esse mecanismo da máquina virtual permite carregar e definir classes em tempo de execução. É uma característica essencial para receber e executar agentes de outras máquinas.

Reflexão. A reflexão que permite a inspeção do sistema pelo próprio sistema em tempo de execução. Enquanto o sistema funciona, é possível obter informações de métodos de classes antes desconhecidas. Essa também é uma característica importante para executar código vindo de outras máquinas.

Seriação de objetos. Java permite que um objeto, na verdade o seu estado, seja *seriado*, isto é, transformado numa seqüência de bytes para ser armazenado. O mesmo mecanismo, evidentemente, permite recuperar o objeto a partir da seqüência de bytes. Esse recurso facilita a migração de objetos entre as máquinas do sistema distribuído.

Programação com múltiplos fluxos de execução (*multi-threading*). Um agente deve ser autônomo e por isso precisa poder executar suas ações independentemente das ações de outros agentes ou aplicações. A linguagem Java permite que um programa tenha mais de um fluxo de execução simultâneo, esse é um mecanismo muito útil para obtermos a autonomia.

Java Swing. Java oferece uma sólida biblioteca para construção de interfaces gráficas. O nosso interesse está em criar rapidamente interfaces independentes de plataforma e que ofereçam um alto grau de interatividade com o usuário.

Suporte a multimídia. Embora a implementação oficial da *Java Sound API*[1] [Java Sound API, sítio] ainda não esteja no nível desejado, com ela foi possível construir um protótipo da infra-estrutura em pouco tempo.

Mas existem também características da linguagem Java que desfavorecem aplicações que usam código móvel.

Falta de suporte para controle de recursos. Java não fornece meios do programador controlar o uso de recursos da máquina. Um objeto pode gastar ciclos do processador ou consumir memória sem necessidade. Esses dois casos podem ser caracterizados como ataques de negação de serviço (veja a Seção 2.2.4), representando assim riscos de segurança.

Impossibilidade de se desalocar objetos. Em Java, não é possível desalocar um objeto. Isso é feito pelo próprio sistema de coleta de lixo da máquina virtual quando não existem mais referências para o objeto, isto é, quando não é mais possível acessá-lo. Isso significa que não é possível garantir o fim da existência de um agente, já que uma outra parte do programa, por exemplo, um outro agente, pode possuir uma referência a ele.

Impossibilidade de se armazenar e restaurar o estado de execução. Um agente precisa poder interromper a sua execução antes de migrar e depois retomá-la assim que chegar ao destino. Mas em Java é impossível armazenar o estado de execução de um objeto, ou seja, um agente móvel em Java não pode realizar a migração forte (veja a Seção 2.1) com os mecanismos normais da linguagem.

A linguagem Java foi utilizada na construção de toda infra-estrutura do Andante e das duas aplicações de exemplo.

4.2.2 Tecnologias de Geração de Som

Apesar de termos apenas usado Java, pretendemos no futuro permitir que partes do sistema sejam implementadas em outras linguagens. O principal motivo para isso é possibilitar o uso de

[1] API (*Application Programming Interface*) é um conjunto de definições de como um software se comunica com outro.

outras tecnologias de geração de som, diferentes das providas pela Java Sound API. Empregamos CORBA para atingir esse objetivo. CORBA é um poderoso sistema de objetos distribuídos que, como veremos na Seção 4.2.3, permite a comunicação transparente entre programas escritos em linguagens diferentes e executando em sistemas operacionais diferentes.

Utilizamos as classes de MIDI [MIDI, sítio] fornecidas pela API do Java Sound para gerar o som. Essas classes dão acesso a um sintetizador MIDI implementado em software. Devido a isso, a versão atual da infra-estrutura do Andante é baseada no protocolo MIDI. Contudo, tomamos o cuidado de não deixar que a arquitetura fosse muito influenciada por esse protocolo, já que ele é considerado muito limitado para aplicações musicais sofisticadas [Moore, 1988, Puckette, 1994].

Como alternativa para geração de som, também realizamos experimentos com o ambiente MAX/MSP, que incorpora uma linguagem visual para construção de sistemas interativos e oferece suporte para síntese sonora em tempo real. Esse sistema foi criado originalmente pelo IRCAM (Institut de Recherche et Coordination Acoustique/Musique) [IRCAM, sítio] em Paris e hoje é desenvolvido e comercializado pela Cycling '74 nos EUA [Cycling '74, sítio].

Para a integração com o Andante, utilizamos o suporte não oficial do MAX/MSP para o protocolo OpenSound Control (OSC) [Wright and Freed, 1997, OSC, sítio]. Esse protocolo permite a comunicação pela rede entre sistemas multimídia.

Veremos um pouco mais de detalhes sobre as tecnologias de som na Seção 4.3.

4.2.3 CORBA

CORBA (Common Object Request Broker Architecture) [Tanenbaum and Steen, 2002, Siegel, 2000, Henning and Vinoski, 1999] é uma especificação aberta [OMG, 2002] de uma arquitetura distribuída baseada em objetos. Essa especificação vem sendo construída pelo OMG (Object Management Group) [OMG, sítio], uma organização sem fins lucrativos formada por centenas de entidades de governos, de universidades e principalmente da indústria. A primeira versão foi lançada em 1991 e a mais recente, CORBA 3.0, em 2001.

Redes de computadores são tipicamente heterogêneas e um dos motivos para isso é a evolução da tecnologia. Geralmente, as redes crescem ao longo do tempo, em vez de serem construídas de

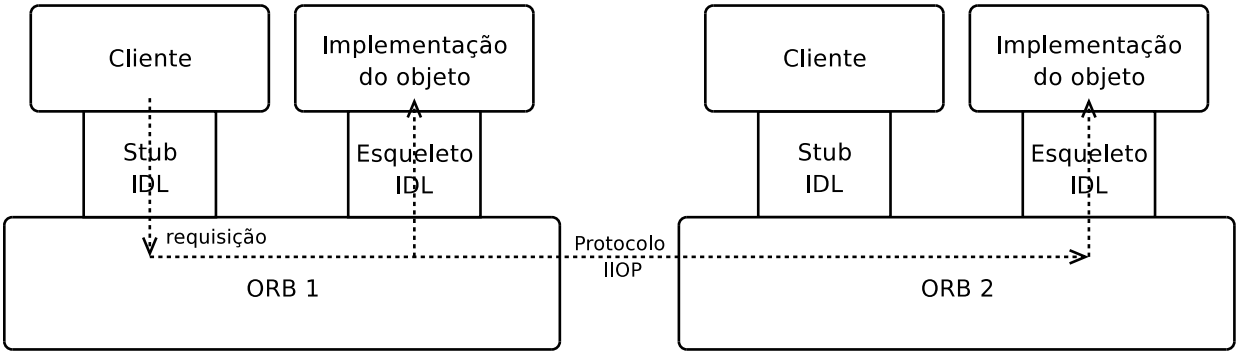

Figura 4.2: Arquitetura básica de um sistema que utiliza CORBA

uma só vez. Em cada momento de expansão da rede, a tecnologia que parece ser a melhor alternativa quase sempre difere das adotadas anteriormente. Além disso, a heterogeneidade também tem vantagens, já que nenhuma combinação de máquina e sistema operacional é a melhor para todas as tarefas. Outra vantagem é que problemas que atingem uma determinada plataforma provavelmente não atingem outras, o que torna a rede mais tolerante a falhas.

A heterogeneidade é, portanto, inevitável e traz muitos desafios para os desenvolvedores de sistemas distribuídos. O objetivo de CORBA é facilitar a construção de sistemas distribuídos heterogêneos. CORBA especifica uma arquitetura e um conjunto de serviços voltados a soluções de problemas relacionados à heterogeneidade. O componente principal da especificação descreve um sistema de objetos remotos que segue um modelo muito semelhante ao descrito na Seção 2.2.1. A Figura 4.2 mostra a arquitetura desse sistema, que discutimos rapidamente a seguir.

CORBA permite que os objetos que compõem uma aplicação estejam distribuídos em máquinas diferentes da rede. Cada tipo de objeto deve ter uma interface escrita na linguagem OMG IDL (*Interface Definition Language*), que é uma descrição dos serviços que o objeto fornece. Essa interface em IDL é mapeada para diversas linguagens de programação, o que permite que o programa que utiliza o objeto possa ser escrito em uma linguagem diferente da linguagem em que é implementado o objeto. Sendo assim, para o cliente usuário do objeto, a interface IDL é mapeada para o Stub IDL, enquanto que para o servidor que fornece o objeto, é mapeada para o Esqueleto IDL. O cliente faz a requisição para o Stub IDL do objeto e o Esqueleto IDL transmite a requisição para o objeto.

A infra-estrutura CORBA, representada pelos ORBs (*Object Request Brokers*) se encarrega de traduzir e enviar a requisição para o objeto em questão. Esse objeto pode estar sendo gerenciado por outro ORB, tipicamente localizado em outra máquina. Nesse caso, a requisição é enviada pela rede utilizando o protocolo IIOP (*Internet Inter-ORB Protocol*). Esse fato, combinado com o mapeamento das interfaces IDL para outras linguagens, permite que os clientes utilizem objetos não só escritos em linguagens diferentes, mas também escritos para máquinas de plataformas diferentes. Essa é a base da interoperabilidade oferecida por CORBA.

A especificação CORBA também descreve serviços [OMG, 1998] que os desenvolvedores de aplicações podem utilizar para gerenciar os objetos. Por exemplo, o Serviço de Nomes (*Naming Service*) permite encontrar objetos através de um identificador único.

É importante notar que CORBA é somente a especificação de uma arquitetura. Existem várias implementações, nem todas completas. A plataforma Java oferece uma implementação que inclui a parte básica descrita até agora, incluindo também o Serviço de Nomes. Essa implementação foi utilizada na construção do sistema Andante.

4.2.4 Aglets

Na infra-estrutura do Andante, o gerenciamento de agentes é baseado em Aglets, um middleware que dá suporte ao desenvolvimento de sistemas baseados em agentes móveis. Ele é distribuído na forma de um pacote chamado *Aglets Software Development Kit* (ASDK), que é escrito em Java e foi originalmente desenvolvido pela IBM [IBM Aglets, sítio]. Hoje é um projeto de código aberto [Aglets, sítio]. O ASDK oferece uma API e aplicações para a implementação e gerenciamento em tempo de execução de agentes móveis [Lange and Oshima, 1998b].

Aglets oferecem apenas a migração fraca (veja a Seção 2.1) devido à limitação de Java de não permitir a manipulação do estado de execução. Sempre que chega ao destino, um agente retoma a sua execução a partir do início. Embora a migração forte seja desejável, principalmente para agentes móveis, o programador do agente, com um esforço adicional, pode obter um resultado semelhante utilizando o estado do próprio agente. Antes de migrar, um agente pode armazenar informações que o ajudem a continuar a execução depois de chegar ao destino.

O Modelo dos Aglets

Descrevemos a seguir os principais elementos da API do ASDK que permitem a criação de um sistema de agentes móveis.

- *Aglet*: é o agente móvel em si. É um objeto Java móvel que visita pontos da rede habilitados para recepção de Aglets, isto é, pontos que possuem um Context ativo. Um Aglet tem seu próprio fluxo de execução e é capaz de receber e enviar mensagens.
- *Context*: é o o ambiente de execução de um Aglet, criado em uma máquina para onde se deseja enviar Aglets. Cada máquina da rede pode ter vários Contexts sendo executados simultaneamente.
- *Identifier*: é uma identificação única e global de um Aglet.

E a seguir estão algumas operações fundamentais que envolvem Aglets.

- *Criação (Creation)*: a criação de um Aglet ocorre dentro de um Context. O Aglet recebe um identificador, é adicionado ao Context e inicializado.
- *Clonagem (Cloning)*: produz uma cópia quase idêntica do Aglet clonado. O novo Aglet recebe um identificador diferente, e a execução é reiniciada.
- *Envio (Dispatching)*: retira o Aglet do seu Context de origem e o adiciona a um dado Context de destino, onde ele é reiniciado.
- *Retração (Retraction)*: retira o Aglet do Context em que está e o adiciona ao Context de onde partiu o pedido de retração.
- *Destruição (Disposal)*: quando um Aglet é destruído, ele interrompe a sua execução e é apagado do Context em que se encontra.

O modelo de programação dos Aglets é baseado em eventos. Isto é, um programador pode definir ações que são executadas quando ocorrem determinados eventos. Por exemplo, é possível definir uma ação que é executada quando um Aglet chega a um Context. Esse evento de chegada em um Context em particular ocorre no final de uma operação de envio.

Vejamos então um exemplo de implementação de agente, mostrado na Figura 4.3. Um agente do tipo `ListingAglet` executa a tarefa de capturar o conteúdo de um diretório de uma máquina remota. Ele implementa o método `onCreation` que instala um `MobilityListener` (linhas 17–42) para lidar com os eventos de mudança de local do Aglet. Ainda no momento da criação, o Aglet migra para uma máquina remota (linha 49). Quando o Aglet chega no seu destino, ocorre o evento de chegada e, conseqüentemente, o método `onArrival` instalado é executado. O Aglet então percebe que precisa obter a listagem do diretório (linhas 21-32). Quando a listagem é completada (linhas 23–27), ele volta para o seu local de origem (linha 27). Ao chegar à origem, ocorre novamente o evento de chegada, mas dessa vez o Aglet imprime a listagem obtida (linhas 34–38).

4.3 Implementação

Implementamos a arquitetura descrita na Seção 4.1 na linguagem Java, conforme mostra o diagrama de classes da Figura 4.4.

O Palco, representado pela classe `Stage`, oferece os seguintes serviços para os agentes.

- `channel()`: fornece um canal de comunicação por onde passam mensagens de geração de som, semelhantes às do protocolo MIDI (mas não limitadas a este). Em geral, cada palco possui vários canais, cada um permitindo diferentes configurações em seus parâmetros (por exemplo, timbre utilizado para tocar as notas, como veremos mais adiante na descrição da implementação de `Channel`).

- `metronome()`: fornece um objeto que funciona como um metrônomo. Esse objeto mede o tempo de acordo com a fórmula de compasso que armazena. A fórmula pode ser alterada pela mensagem `setTimeSignature` enviada ao metrônomo. Ele também recebe pedidos de registros de agentes através da mensagem `register` e envia a mensagem `pulse` em cada tempo do compasso a todos os agentes registrados.

Para implementar um agente, o usuário da infra-estrutura precisa criar uma nova subclasse da classe abstrata `MusicalAgent` implementando as mensagens necessárias. Algumas mensagens

```
1  // Todo agente deve ser subclasse de 'Aglet' (que é uma classe abstrata).
2  public class ListingAglet extends Aglet {
3
4    boolean done = false;                // agente completou a tarefa?
5    File dir = new File("/tmp/PUBLIC"); // diretório a ser listado
6    String[] list;                       // conteúdo do diretório
7    URL origin = null;                   // endereço do Context de origem
8
9    // Este método é chamado na ocorrência do evento de criação do Aglet.
10   // O parâmetro 'o' tem conteúdo especificado pelo programador do
11   // agente (pode conter qualquer coisa, mas neste caso não contém nada).
12   public void onCreation(Object o) {
13
14     // Adiciona um 'listener' para eventos relacionados a mobilidade. Um
15     // 'listener' um objeto com métodos que são chamados quando ocorrem
16     // determinados eventos.
17     addMobilityListener(
18       new MobilityAdapter() {
19         // Método chamado sempre que o agente chega a uma nova máquina.
20         public void onArrival(MobilityEvent me) {
21           if (!done) {
22             try {
23               // obtém a listagem do diretório:
24               list = dir.list();
25               done = true;
26               // e volta para o Context de origem:
27               dispatch(origin);
28             }
29             catch (Exception e) {
30               dispose();
31             }
32           }
33           else { // done == true
34             // o agente já tem a listagem, então a imprime:
35             for (int i=0; i < list.length; i++)
36               System.out.println(i + ": " + list[i]);
37             // e se destrói:
38             dispose();
39           }
40         }
41       }
42     );
43
44     // descobre o endereço do Context de origem (lembrando que neste
45     // ponto estamos tratando o evento de criação do Aglet):
46     origin = getAgletContext().getHostingURL();
47     // finalmente, envia o agente para um nó remoto:
48     try {
49       dispatch(new URL("atp://villa.ime.usp.br:4444"));
50     }
51     catch (Exception e) {
52       System.out.println("Failed to dispatch Aglet");
53     }
54   }
55 }
```

Figura 4.3: Exemplo de Aglet

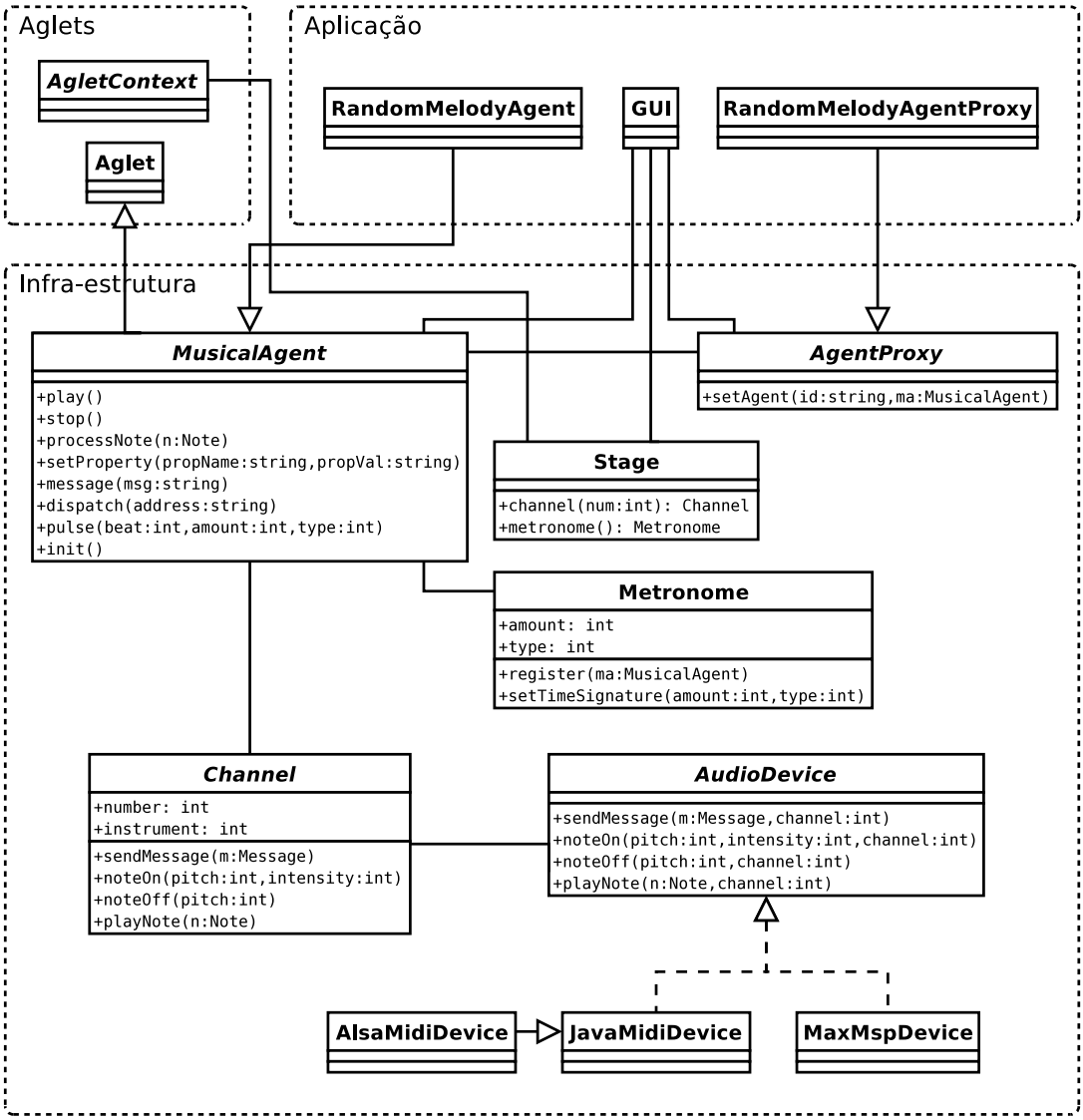

Figura 4.4: Diagrama de classes da implementação

representam comandos que o agente oferece para uma aplicação Andante e outras são mensagens enviadas pela infra-estrutura para gerenciar os agentes. As mais importantes são as seguintes.

- `init`: ao chegar a um palco, o agente pode precisar inicializar algumas variáveis para restaurar o seu estado de execução. O palco envia a mensagem `init` ao agente logo após a sua chegada, dando, portanto, a chance para o agente fazer isso.

- `pulse(int beat, int amount, int type)`: caso o agente se registre no serviço de metrônomo do palco, ele recebe a mensagem `pulse` em tempos regulares, de acordo com a fórmula de compasso do metrônomo. O agente pode executar uma ação quando isso ocorre. O parâmetro `beat` indica o tempo do compasso a que corresponde a mensagem `pulse`, os parâmetros `amount`, `type` indicam a fórmula do compasso.

- `play`: ordena o agente a iniciar ou continuar a sua performance.

- `stop`: ordena o agente a interromper a sua performance.

- `processNote(Note n)`: faz com que o agente execute alguma ação usando a nota `n`. Cabe ao agente decidir o que fazer, ele pode simplesmente reproduzir a nota, por exemplo.

- `setProperty(String propName, String propVal)`: determina que a propriedade de nome `propName` do agente deve assumir o valor `propVal`. Cada tipo de agente possui um conjunto de propriedades, o significado de cada propriedade é livre, depende apenas do autor do agente.

- `message(String msg)`: envia um texto qualquer para o agente. Assim como em `setProperty`, o significado do texto depende da implementação do agente.

- `dispatch(String address)`: ordena o agente a migrar para o palco cujo endereço é `address`.

O Dispositivo de Áudio, representado por `AudioDevice`, fornece seus serviços através de canais, representados por objetos da classe `Channel`. A interface das duas classes é semelhante, um canal recebe as mensagens de um agente e as repassa para o dispositivo associado a ele. As mensagens que um agente pode enviar a um canal são as seguintes.

- `sendMessage(Message m)`: a mensagem `m` é semelhante a uma mensagem no protocolo MIDI, ela determina uma ação a ser executada (por exemplo, ligar uma nota ou mudar um timbre) e os seus parâmetros (por exemplo, a nota a ser ligada ou o timbre a ser usado).
- `noteOn(int pitch, int intensity)`: liga a nota de altura `pitch` e intensidade `intensity`.
- `noteOff(int pitch)`: desliga a nota de altura `pitch`.
- `playNote(Note n)`: gera a nota `n`. A nota `n` contém as informações de altura, intensidade e duração.

O canal ainda armazena o timbre (ou instrumento) com que as notas devem ser geradas. Usando a mensagem `sendMessage`, podemos obter o mesmo efeito das três últimas mensagens.

A Figura 4.5 mostra um exemplo de um agente móvel musical implementado sobre a infra-estrutura do Andante. Esse agente não implementa todas as mensagens, apenas reproduz uma melodia gerada aleatoriamente em tempo real. A seguinte descrição ilustra um possível uso do agente `RandomMelodyAgent` na infra-estrutura descrita até agora.

1. *Implementação de agentes*: já temos um agente implementado, o `RandomMelodyAgent` (veja a Figura 4.5). Por simplicidade, usaremos apenas uma instância desse agente neste exemplo. Vamos chamar essa instância de *rmAgent*.
2. *Envio de agentes*: depois de criado, *rmAgent* deve ser enviado a uma instância da classe `Palco`.
3. *Procedimentos de chegada*: ao chegar ao Palco, o campo herdado `stage` recebe automaticamente uma referência para o Palco. Além disso, é iniciado o tratamento do evento de chegada. Esse tratamento é feito pelo código da superclasse `MusicalAgent`. Ao final do tratamento, o método `init` do próprio agente é chamado. No caso de *rmAgent*, esse método obtém um canal (campo `ch` na Figura 4.5) e chama o método `play` (linhas 16–17).
4. *Performance do agente*: com a chamada ao método `play`, *rmAgent* inicia sua execução. Ele usa as operações de `ch` para tocar notas musicais aleatórias na escala diatônica de dó (linhas 24–34).

```
1  import java.util.Random;
2
3  // Todo agente móvel musical deve ser subclasse da classe abstrata
4  // 'MobileMusicalAgent'.
5  public class RandomMelodyAgent extends MobileMusicalAgent {
6
7    boolean _play = false;
8    short [] _cMaj = {60,62,64,65,67,69,71};
9    Random _random;
10   Channel _channel;
11
12   // Método chamado no final do tratamento do evento de chegada
13   // do agente a um novo palco.
14   public void init() {
15     _random = new Random();
16     _channel = _stage.channel(1);
17     play();
18   }
19
20   public void play() {
21     _play = true;
22     int pitch, intensity, duration;
23
24     while (_play) {
25       pitch = _cMaj[_random.nextInt(_cMaj.length)];
26       intensity = _random.nextInt(128);
27       duration = _random.nextInt(1000);
28       _channel.noteOn(pitch, intensity);
29       try {
30         Thread.sleep(duration);
31       }
32       catch (InterruptedException ie) {}
33       _channel.noteOff(pitch);
34     }
35   }
36
37   public void stop() {
38     _play = false;
39   }
40
41 }
```

Figura 4.5: `RandomMelodyAgent`: exemplo de um agente Andante

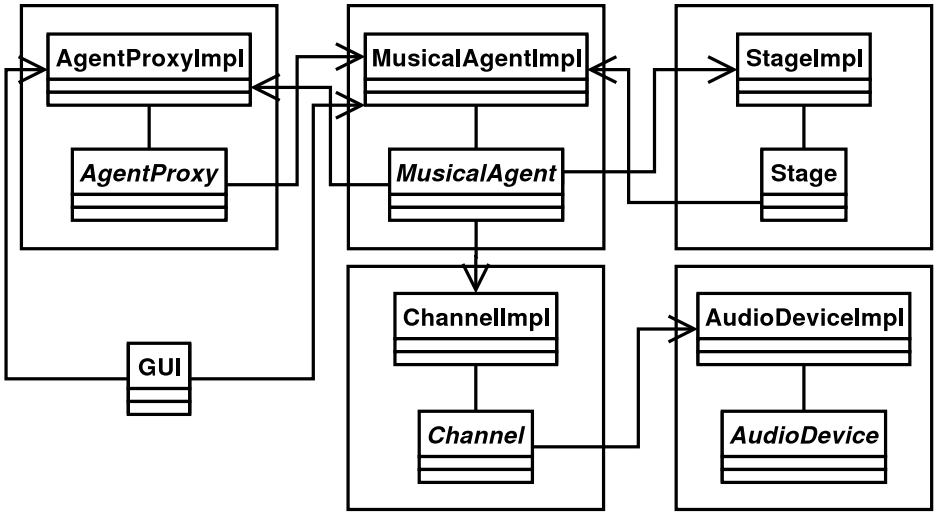

Figura 4.6: Diagrama de classes com a camada CORBA

5. *Controle do agente*: é possível implementar uma interface gráfica que envia comandos para *rmAgent*. Nesse caso, isso poderia ser útil para chamar o método `stop` desse agente ou transferi-lo para outro palco.

Camada CORBA. Como já mencionamos, queremos permitir que certos elementos, mais precisamente `AudioDevices` e `GUIs`, sejam implementados em outras linguagens e, eventualmente, utilizem sistemas legados como CSound ou jMax. Para tornar isso possível, existe uma camada CORBA para a comunicação entre os elementos da arquitetura. Essa camada, que foi omitida da descrição até agora, é ilustrada na Figura 4.6 (por simplicidade, os métodos e metrônomo não estão na figura). As classes com o sufixo *Impl* no nome representam o envólucro CORBA.

Não implementamos de fato um dispositivo de áudio em outra linguagem, mas a infra-estrutura Andante oferece mais de um dispositivo.

- `JavaMidiDevice`: é o dispositivo padrão, utiliza o sintetizador MIDI por software da implementação oficial da Java Sound API.

- `AlsaMidiDevice`: uma opção apenas para Linux, utiliza uma implementação independente da Java Sound API chamada Tritonus [Tritonus, sítio], que acessa o sintetizador MIDI da placa de som (se disponível). O acesso é feito por drivers e bibliotecas ALSA (*Advanced*

Linux Sound Architecture) [ALSA, sítio]. Este dispositivo foi implementado para contornar o problema de latência e *jitter* observados nos sintetizadores MIDI das implementações oficiais para Windows e Linux[2].

- `MaxMspDevice`: é o dispositivo utilizado nos experimento para integração do ambiente MAX/MSP com o Andante.

  A Figura 4.7 mostra o *patch* MAX construído para essa integração. Apenas dois comandos são processados pelo *patch*: *noteout*, usado para gerar notas MIDI, e *pgmout*, usado para mudar o instrumento de canais MIDI.

  Em vez de enviar comandos para um dispositivo real, o `MaxMspDevice` gera mensagens no protocolo OpenSound Control e as envia pela rede utilizando o protocolo UDP. O recurso básico oferecido pelo protocolo OpenSound Control e utilizado no Andante é o envio de uma requisição de execução de um comando juntamente com os parâmetros de entrada.

  O *patch* então recebe as requisições e executa o comando desejado da seguinte forma. As mensagens UDP são recebidas pelo objeto otudp (veja a Figura 4.7) e traduzidas de volta para o protocolo OpenSound Control. O objeto OpenSoundControl interpreta a mensagem e a passa adiante. A caixa de mensagem 'prepend set \;' envia o comando e os parâmetros para o objeto r correspondente, que por sua vez envia os parâmetros para o objeto que executa o comando (noteout ou pgmout).

  A composição da direita da Figura 4.7 é apenas um teste de envio de mensagens no sentido contrário, isto é, do ambiente MAX/MSP para o Andante. As mensagens não são interpretadas na implementação atual.

[2] O problema foi observado nas versões até a 1.4.2. Na versão para MacOS, no entanto, não foram observados valores significativos.

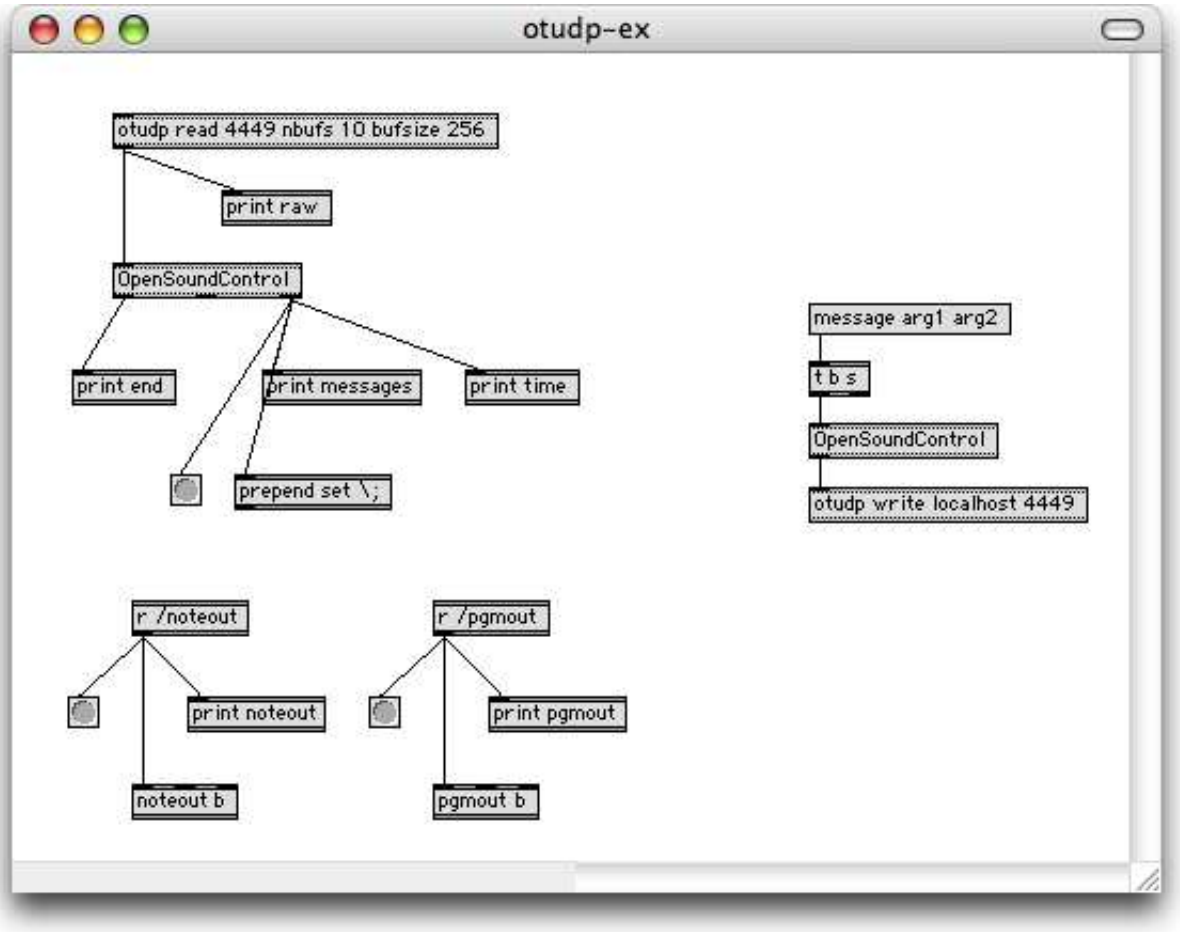

Figura 4.7: Patch MAX para integração com o Andante

Capítulo 5

Aplicações de Exemplo

Neste capítulo descrevemos duas aplicações que construímos utilizando a infra-estrutura apresentada no Capítulo 4.

5.1 NoiseWeaver

NoiseWeaver é uma a aplicação para geração de música estocástica distribuída. Somente um tipo de agente é utilizado: o *NoiseAgent*. Ele utiliza ruídos $\frac{1}{f^\beta}$ para compor uma melodia simples [Roads, 1996] que é reproduzida pelo agente em tempo real. Nela, seqüências de números que simulam tipos de ruídos selecionados pelo usuário determinam os parâmetros de altura, intensidade e duração de cada nota. Por exemplo, um agente do tipo NoiseAgent pode gerar uma melodia cujas alturas das notas são determinadas por uma seqüência de números que simula o ruído rosa. O mesmo agente pode ter as durações das notas determinadas por um ruído browniano e as intensidades determinadas por um ruído branco. Simplificando, no contexto desta aplicação, *ruído* significa uma seqüência de números inteiros gerada através de um algoritmo estocástico e que é mapeada para parâmetros musicais de uma melodia.

Os agentes usam o serviço de metrônomo do Palco para sincronizar as melodias. Todos os agentes se registram no serviço e o metrônomo passa então a enviar a mensagem `pulse` a todos eles em intervalos de tempo regulares. Ao receber a mensagem `pulse`, um NoiseAgent gera uma nota da melodia (ou não faz nada, se a última nota ainda estiver soando). Cada agente possui

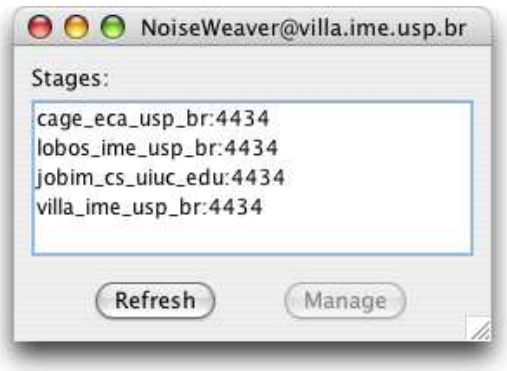

Figura 5.1: Escolha de Palcos

diversas propriedades que determinam o modo como a sua melodia deve ser gerada e reproduzida. Essas propriedades serão descritas mais adiante.

A aplicação NoiseWeaver fornece uma interface gráfica para controlar vários NoiseAgents hospedados em vários Palcos. Tanto os Palcos como os NoiseAgents são localizados através do Serviço de Nomes CORBA [OMG, 1998]. A Figura 5.1 mostra a janela com os Palcos encontrados. Para controlar os NoiseAgents de um determinado Palco, o usuário deve selecionar o Palco e pressionar o botão "*Manage*". Com isso, abre-se uma janela semelhante à exibida na Figura 5.2. Essa janela permite que um usuário altere as propriedades do metrônomo do Palco e dos agentes, mesmo enquanto as melodias estão sendo produzidas.

Na Figura 5.2, o painel *Metronome* permite definir os parâmetros do metrônomo. *Tempo* é o andamento da música em pulsos por minuto; *Amount* e *Type* são a fórmula do compasso; e *Play pulses* determina se o metrônomo emite ou não um som a cada tempo do compasso. O painel *Agents* lista os vários agentes hospedados no palco. O painel *Dispatch* se refere ao agente selecionado na lista e permite enviar uma ordem de migração para outro palco.

Quando um agente é selecionado, o painel da metade direita da janela é ativado. Esse painel permite controlar as propriedades do agente. No painel *Commands*, temos:

- *Start*: faz o agente começar (ou continuar) a tocar;
- *Stop*: faz o agente parar de tocar.

No painel *Properties*, temos:

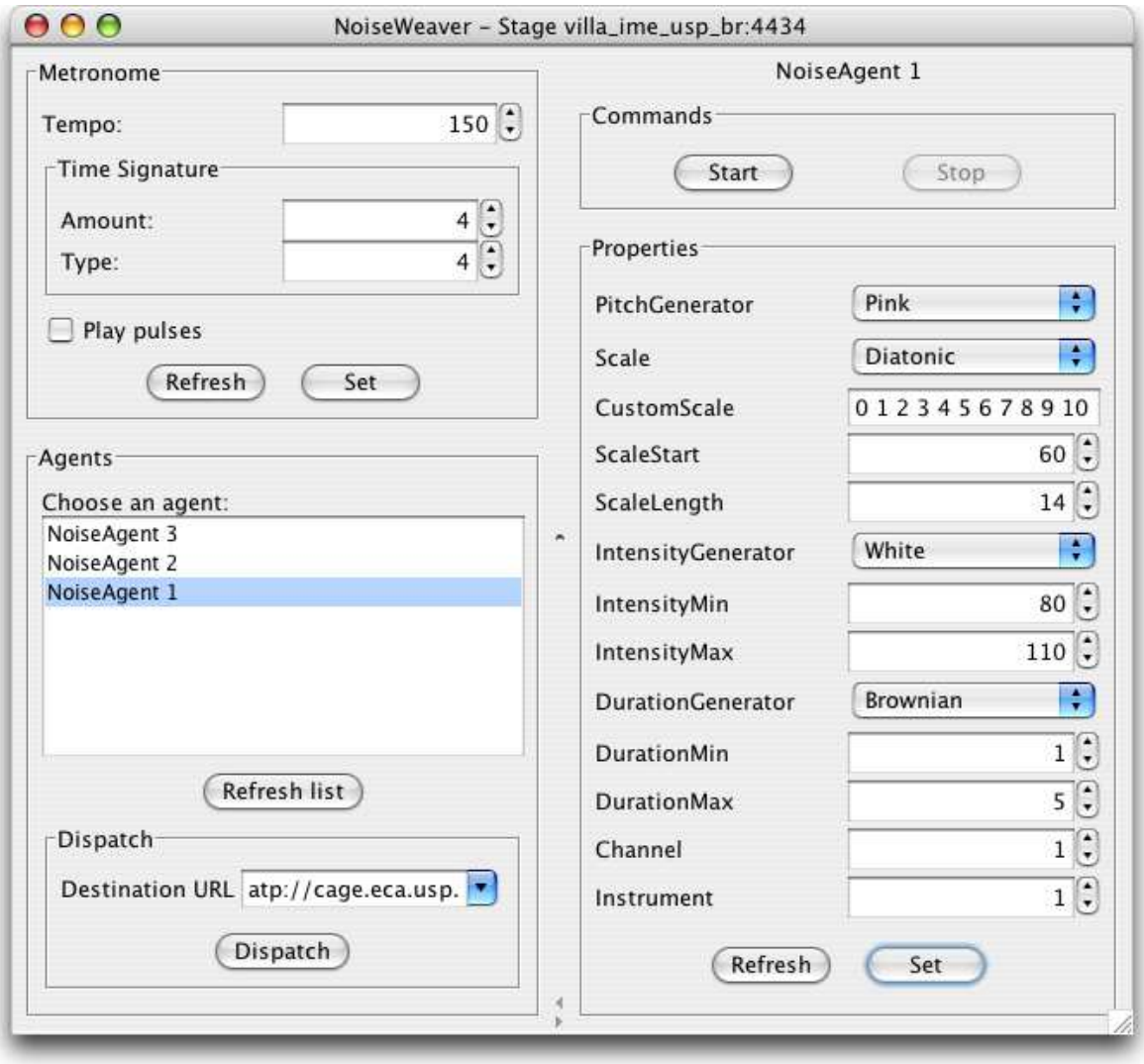

Figura 5.2: Janela para controle de um Palco

- **Altura**: *PitchGenerator* define o tipo de ruído (ou seja, o algoritmo) que será utilizado para gerar as alturas (freqüências) das notas. Os números gerados pela simulação do ruído são mapeados para notas da escala definida por *Scale*. *ScaleStart* e *ScaleLength* determinam qual gama de notas dessa escala será utilizada. *ScaleStart* é a primeira nota da escala, sendo que 60 é o dó central. *ScaleLength* é o número de notas da escala que serão utilizadas, contando a partir de *ScaleStart*.

- **Intensidade**: *IntensityGenerator* define o tipo de ruído que gerará a intensidade das notas. A intensidade é um inteiro entre 0 e 127. *IntensityMin* e *IntensityMax* determinam o intervalo de valores possíveis.

- **Duração**: *DurationGenerator*, *DurationMin* e *DurationMax* são as propriedades de duração das notas, funcionam de forma análoga às propriedades de intensidade.

- **Channel**: o canal que o agente usa para enviar as notas para o dispositivo de áudio.

- **Instrument**: o instrumento usado para tocar a melodia.

Os possíveis valores para os geradores de números baseados em ruídos são: *Constant*, *White*, *Pink* e *Brownian*. As possíveis escalas são: *Diatonic*, *WholeTone*, *Chromatic*, *HarmonicMinor*, *HarmonicNatural*, *Pentatonic*, *Blues*, *PentaBlues* e *Hirajoshi*. A propriedade *CustomScale* permite a utilização de outras escalas definidas pelo usuário.

A aplicação NoiseWeaver permite então gerar diversas melodias estocásticas simultâneas através dos NoiseAgents e controlá-las usando a interface gráfica.

5.2 Maestro

A aplicação *Maestro* foi construída como uma extensão da aplicação *NoiseWeaver*. Além de serem controlados por uma interface gráfica, uma coleção distribuída de NoiseAgents é coordenada por um Maestro, que por sua vez é controlado por um script. O elemento principal do script é a partitura, onde são determinadas mudanças ao longo do tempo nas propriedades principalmente

dos agentes. Essas mudanças são feitas através da interface `MusicalAgent`, isto é, cada mudança é um envio de mensagem para um agente.

Apesar de inicialmente ser uma extensão do NoiseWeaver, o Maestro permite controlar qualquer tipo de agente, não apenas NoiseAgents. Agentes implementados no futuro também poderão ser usados sem alterações no código do Maestro. Essa flexibilidade foi obtida através da API de reflexão de Java (veja a Seção 4.2.1), as classes de agentes podem ser definidas em tempo de execução. O mesmo é feito com as mensagens passadas para os agentes. Qualquer mensagem nova incluída na interface `MusicalAgent` que receba parâmetros do tipo `String` pode ser chamada sem alterações no Maestro.

A aplicação Maestro fornece uma interface gráfica para edição do script, como mostra a Figura 5.3. Além das funcionalidades básicas de edição de texto, a interface permite iniciar ou encerrar a execução do script.

A Figura 5.4 mostra uma descrição na BNF da sintaxe do script de entrada para o Maestro. O script é dividido em três partes: as Declarações (*Declarations*), as Inicializações (*Initializations*) e a Partitura (*Score*).

Declarações. Nesta seção são declarados os agentes e os palcos utilizados no script. Uma declaração de agente define o tipo do agente (através da classe Java) e associa a ele um identificador. A declaração do palco associa um identificador a um palco já existente. Os identificadores são usados ao longo do script para fazer referência aos agentes e palcos declarados. Um agente declarado é criado pelo Maestro, ao contrário de um palco, que já deve ter sido criado pela infra-estrutura do Andante.

Inicializações. Aqui são dadas as mensagens que serão enviadas aos agentes e palcos antes do início da execução do script.

Partitura. De forma semelhante às inicializações, na partitura se encontram as mensagens enviadas durante a execução do script. Cada mensagem, no entanto, tem a ela associada um instante de tempo. A unidade de tempo é um pulso do metrônomo do Maestro. As mensagens não precisam ser digitadas na ordem em que são enviadas no script, a ordem

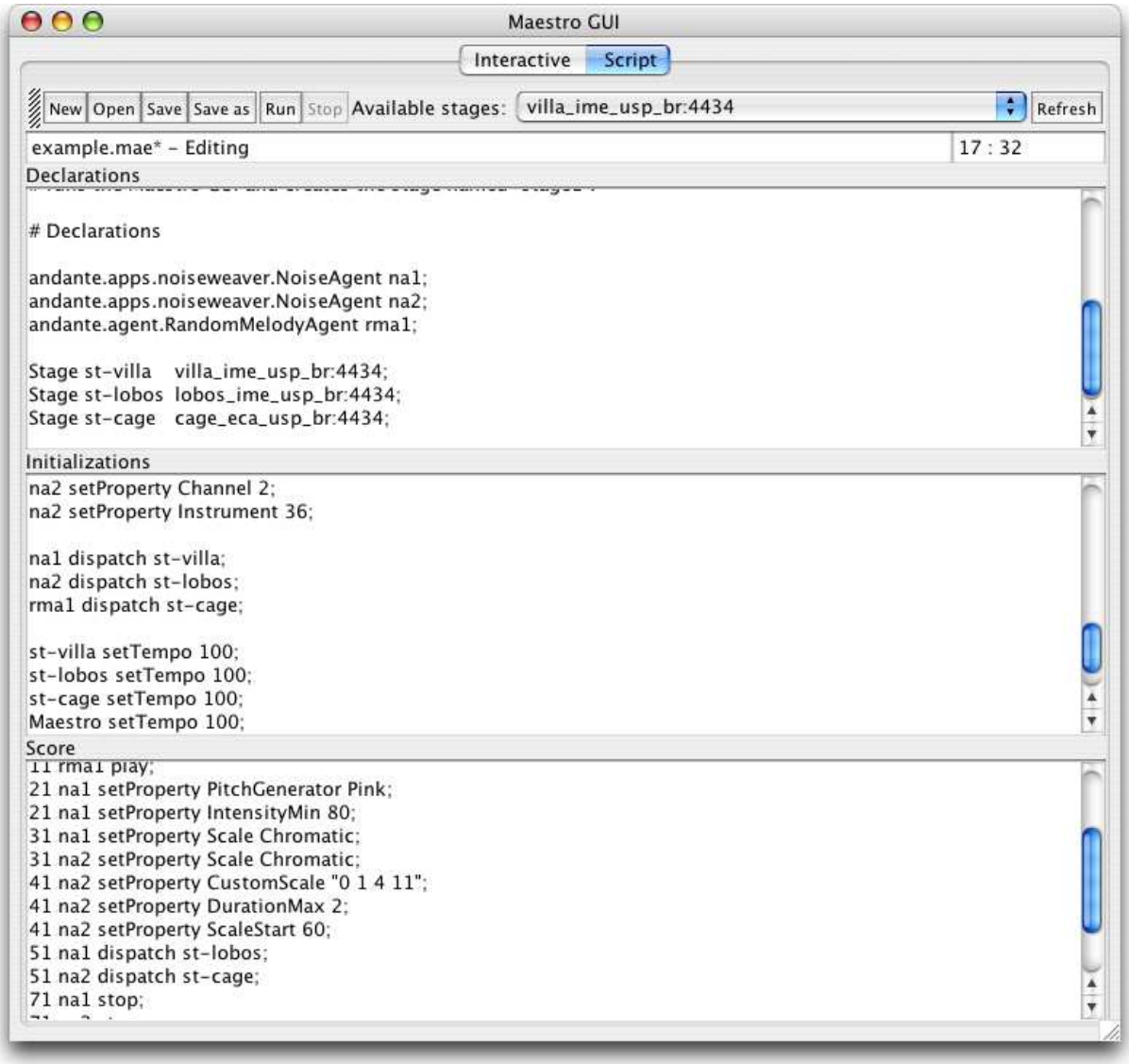

Figura 5.3: Interface do Maestro para o script

Formato do script ::=
<***Declarations***> -- <***Initializations***> -- <***Score***> --

<***Declarations***>	::=	<*vazio*> | <*Decl. de agente*>; <***Declarations***> | <*Decl. de palco*>; <***Declarations***>
<***Initializations***>	::=	<*vazio*> | <*Envio de mensagem*>; <***Initializations***>
<***Score***>	::=	<*vazio*> | <*Tempo*> <*Envio de mensagem*>; <***Score***>
<*Decl. de agente*>	::=	<*Classe*> <*Id*>
<*Decl. de palco*>	::=	`Stage` <*Id*> <*Palco*>
<*Envio de mensagem*>	::=	<*Id*> <*Mensagem*> <*Lista de parâmetros*>
<*Lista de parâmetros*>	::=	<*vazio*> | <*Parâmetro*> <*Lista de parâmetros*>

<*vazio*>	é um texto vazio (em branco).
<*Classe*>	é o nome de uma classe Java que define um agente.
<*Id*>	é um nome qualquer ou a palavra `Maestro`.
<*Palco*>	é o nome de um palco no servidor de nomes CORBA.
<*Mensagem*>	é o nome de um método da interface `MusicalAgent` ou a mensagem `setTempo`.
<*Parâmetro*>	é um nome qualquer sem espaços ou entre aspas.
<*Tempo*>	é um instante de tempo (um número inteiro).

Na ocorrência do símbolo `#`, todo o texto à direita, inclusive o próprio símbolo, é ignorado.

Figura 5.4: Sintaxe do script de entrada do Maestro

é determinada apenas pelo instante de tempo. É possível também determinar que sejam enviadas mais de uma mensagem no mesmo instante de tempo, basta que os instantes sejam iguais.

As mensagens podem ser enviadas para cada agente declarado, para cada palco declarado ou para o próprio Maestro. Para os agentes, as mensagens que podem ser enviadas são as mensagens da interface `MusicalAgent` que recebem apenas parâmetros do tipo `String` (veja a Seção 4.3). A única exceção é a mensagem `dispatch`, que em `MusicalAgent` recebe um endereço codificado em um tipo `String`, mas no Maestro recebe um identificador de palco.

Para os palcos e o Maestro, apenas uma mensagem pode ser enviada: `setTempo`, que recebe um inteiro como parâmetro. Essa mensagem altera o tempo em pulsos por minuto do metrônomo do palco ou do Maestro.

A Figura 5.5 mostra um script onde são usados dois NoiseAgents e dois palcos, contendo também exemplos de todas as possíveis mensagens.

Durante o ano de 2004, pudemos contar com a colaboração do compositor Wilson Cerqueira Ferreira, que contribuiu com idéias e, como usuário da aplicação Maestro, ajudou a encontrar e resolver diversos erros de implementação. A colaboração também resultou em duas peças musicais que foram divulgadas em [Andante, sítio].

Interface interativa. Uma outra opção para o controle dos agentes é a interface interativa, semelhante à do NoiseWeaver. Esta interface, no entanto, permite que qualquer tipo de agente seja controlado. Veja as Figuras 5.6 e 5.7. A interface tem quatro características adicionais importantes:

- *Armazenamento e recuperação de estado*: a interface permite gravar as propriedades dos agentes e dos metrônomos. O formato de arquivo gerado é o mesmo do script de entrada do Maestro.

- *Sincronização de metrônomos*: reinicia todos os metrônomos no mesmo instante, fazendo com que eles enviem a primeira mensagem `pulse` ao mesmo tempo. Porém, ainda não implementamos nenhum algoritmo especial de sincronização, os metrônomos são simplesmente

```
# Declarações

andante.apps.noiseweaver.NoiseAgent a1;
andante.apps.noiseweaver.NoiseAgent a2;

Stage s1 villa_ime_usp_br:4434;
Stage s2 lobos_ime_usp_br:4434;

--

# Inicializações

a1 setProperty PitchGenerator Constant;
a1 setProperty IntensityGenerator White;
a1 setProperty IntensityMin 30;
a1 setProperty IntensityMax 127;
a1 setProperty DurationGenerator Pink;
a1 setProperty DurationMin 1;
a1 setProperty DurationMax 1;
a1 setProperty Scale Diatonic;
a1 setProperty ScaleStart 36;
a1 setProperty ScaleLength 12;
a1 setProperty Channel 1;
a1 setProperty Instrument 1;

a2 setProperty PitchGenerator Pink;
a2 setProperty IntensityGenerator White;
a2 setProperty IntensityMin 80;
a2 setProperty IntensityMax 127;
a2 setProperty DurationGenerator Pink;
a2 setProperty DurationMin 1;
a2 setProperty DurationMax 4;
a2 setProperty Scale Diatonic;
a2 setProperty ScaleStart 72;
a2 setProperty ScaleLength 12;
a2 setProperty Channel 2;
a2 setProperty Instrument 36;

a1 dispatch s1;
a2 dispatch s2;

s1 setTempo 100;
s2 setTempo 100;
Maestro setTempo 100;

--

# Partitura

1  a1 play;
2  a1 message "Uma mensagem qualquer.";
11 a2 play;
21 a1 setProperty PitchGenerator Pink;
21 a1 setProperty IntensityMin 80;
31 a1 setProperty Scale Chromatic;
31 a2 setProperty Scale Chromatic;
41 a2 setProperty CustomScale "0 1 4 11";
41 a2 setProperty DurationMax 2;
41 a2 setProperty ScaleStart 60;
51 a1 dispatch s2;
51 a2 dispatch s1;
71 a1 stop;
71 a2 stop;

--
```

Figura 5.5: Exemplo de script de entrada do Maestro

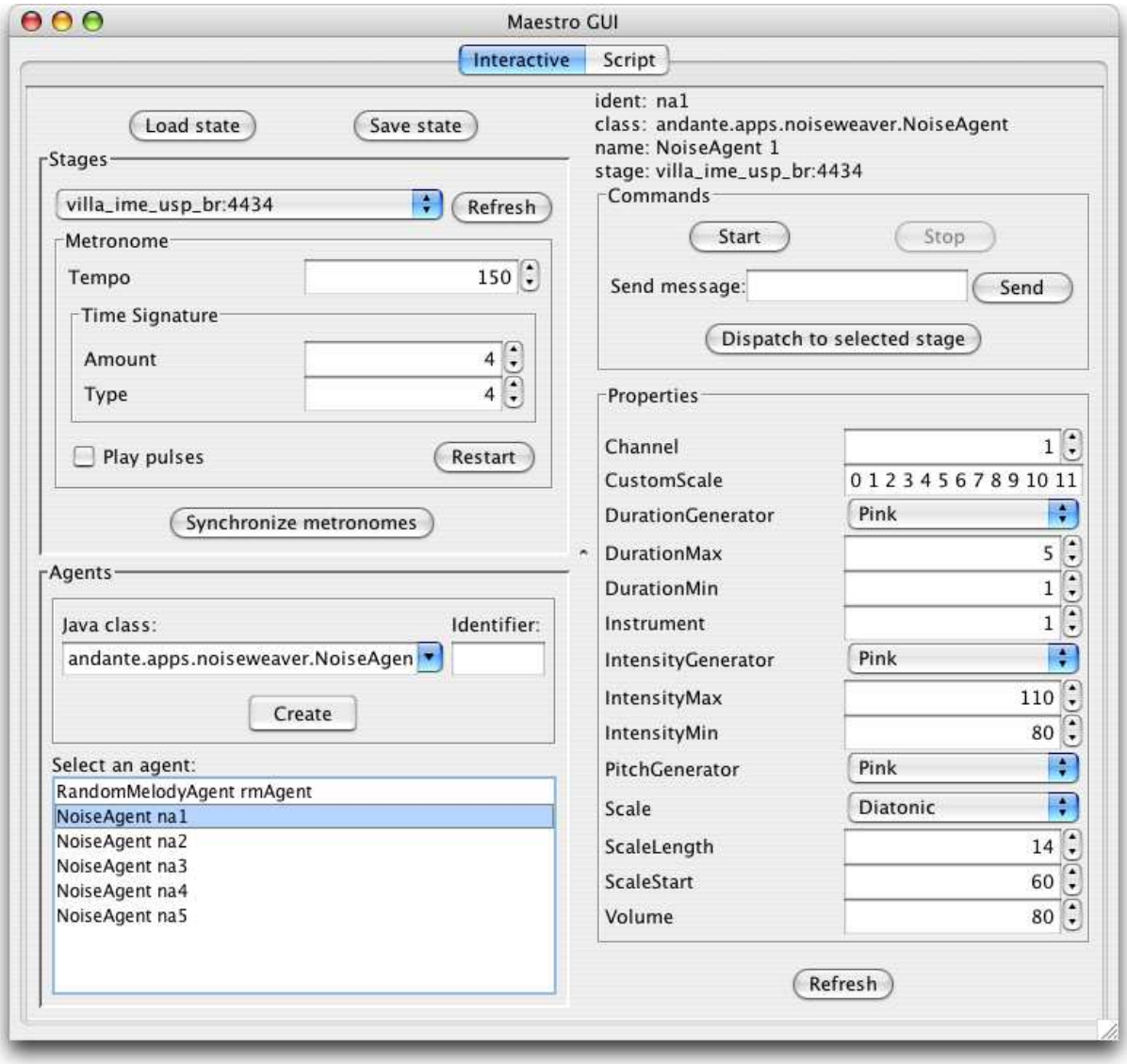

Figura 5.6: Interface interativa do Maestro, mostrando o painel construído para o NoiseAgent

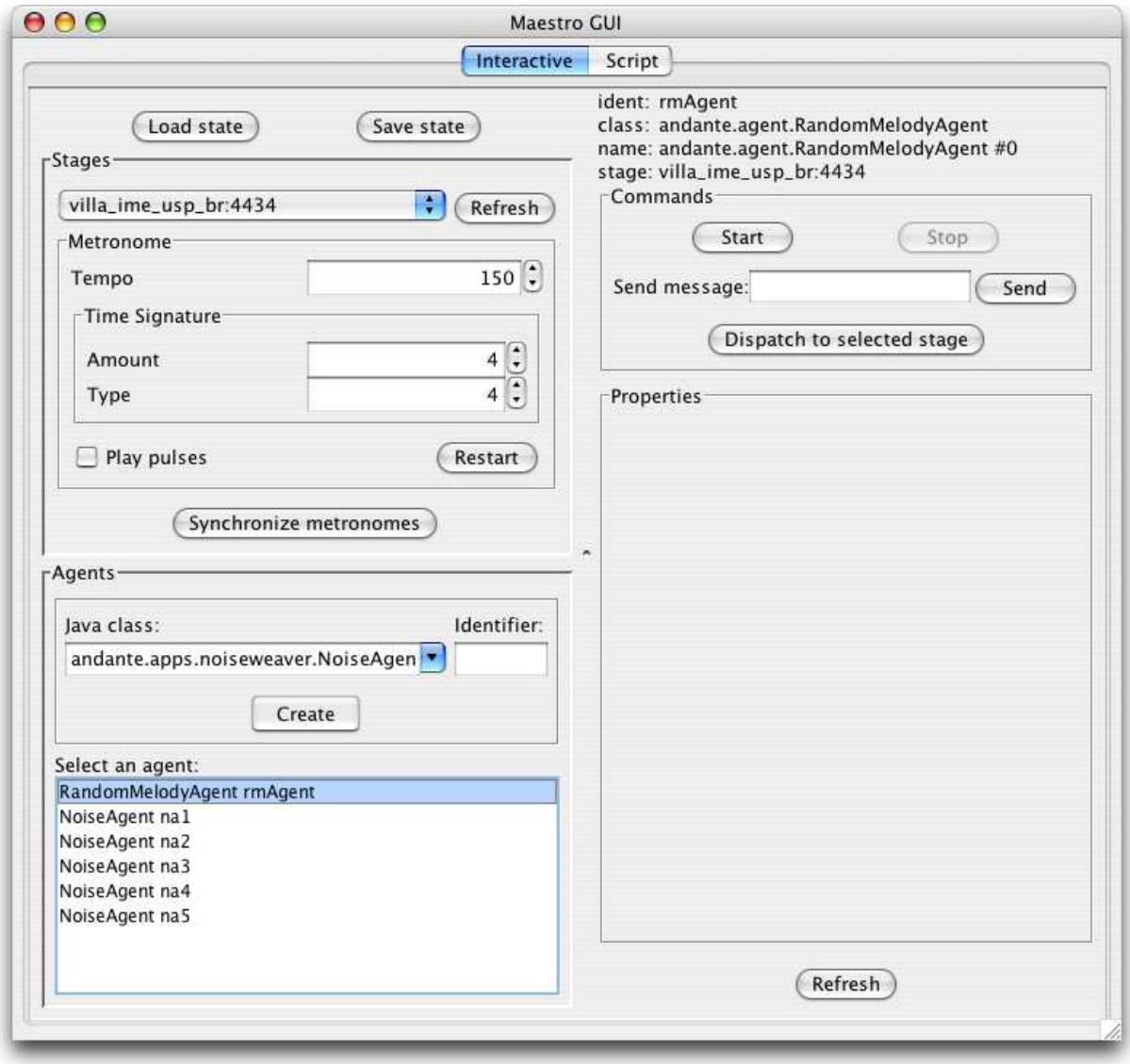

Figura 5.7: Interface interativa do Maestro, mostrando o painel construído para o RandomMelodyAgent

reiniciados em seqüência.

- *Criação de agentes*: permite que o usuário crie um agente e o associe a um identificador.
- *Definição de propriedades*: quando um agente é criado, a interface obtém informações sobre as propriedades do agente. Ela usa essas informações para construir o painel da metade direita da janela dinamicamente. A Figura 5.6 mostra o painel para o NoiseAgent, enquanto que a Figura 5.7 mostra o painel para o RandomMelodyAgent (que não possui nenhuma propriedade).

Capítulo 6

Conclusão

Inspirados na constante relação entre música e ciência em busca da inovação, criamos o conceito de agentes móveis musicais, que são programas autônomos e móveis capazes de gerar música. Produzimos um modelo onde tais agentes interagem entre si em diversos pontos de uma rede de computadores, resultando no que chamamos de música distribuída.

A partir disso, implementamos a versão inicial de uma infra-estrutura de software que permite a construção de aplicações baseadas nesse modelo. Utilizando essa infra-estrutura, construímos então duas aplicações que foram utilizadas por um compositor colaborador.

Todo o código produzido está disponível na Internet sob a licença GPL [GPL, sítio] através da Incubadora Virtual da FAPESP [Incubadora, sítio]. São cerca de 11200 linhas no total, sendo, em números aproximados: 3200 linhas de código Java correspondente à infra-estrutura, 6900 de código Java correspondente às aplicações e 1100 linhas em outras linguagens (OMG IDL, Bash script e XML).

Nosso trabalho resultou em três publicações:

- um artigo completo no IX Simpósio Brasileiro de Computação Musical [Ueda and Kon, 2003];
- um artigo completo na *International Computer Music Conference* [Ueda and Kon, 2004a];
- um pôster na *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '04) [Ueda and Kon, 2004b].

6.1 Trabalhos Futuros

A implementação apresentada nesta dissertação é apenas a primeira versão do sistema Andante. Há muito espaço para melhorias e extensões tanto na infra-estrutura quanto em relação às aplicações.

Infra-estrutura

- Para garantir a performance das peças musicais em tempo real, é necessário que a infra-estrutura do Andante forneça garantias às aplicações de que poderão executar dentro dos devidos prazos. A implementação desse suporte será necessária para atrair o interesse de mais usuários.

- A garantia de qualidade de serviço (*Quality of Service – QoS*) está relacionada ao suporte para tempo real. A idéia é permitir que as aplicações definam seus requisitos de recursos de sistema (como uso do processador, memória, largura de banda da rede, etc.) para poderem executar com uma qualidade satisfatória. Esperamos que o suporte a tempo real em Java (RTSJ) cuja especificação [RTSJ, sítio] foi concluída recentemente e cujas primeiras implementações [Timesys, sítio] estão surgindo, venha a ser o veículo ideal para aplicações musicais com seus fortes requisitos de qualidade de serviço.

- Seria interessante possibilitar a programação de agentes por usuários não programadores. A construção de um ambiente de programação baseado em modelos visuais pode ser um interessante projeto a ser integrado ao Andante no futuro.

- Uma alternativa para a tecnologia de geração de som, que atualmente é baseada em MIDI, se tornaria em outro atrativo para usuários compositores. Já demos um pequeno passo em direção à integração com o ambiente MAX/MSP. A mais recente versão desse ambiente, a 4.5, já fornece a integração completa com a linguagem Java, esse recurso pode ser explorado.

  Uma segunda alternativa é o sistema CSound, uma poderosa e difundida ferramenta para síntese de som [Boulanger, 2000].

- As alterações das propriedades dos agentes poderiam ser definidas por funções no tempo, em vez de pontualmente, de forma a permitir mudanças contínuas.

Aplicações

- Em relação ao NoiseWeaver.
  - Os NoiseAgents poderiam ter mais parâmetros determinados por ruídos, como, por exemplo, timbre e escala.
  - O ciclo de vida e a migração dos agentes poderiam também ser determinados estocasticamente.
  - A interface de mudança de parâmetros poderia expressar visualmente o estado do NoiseAgent, por exemplo, através do uso de barras de rolagem no lugar dos *spinners*. Isso permitiria o controle mais intuitivo dos agentes.
- Em relação ao Maestro.
  - Recursos mais avançados para a liguagem do script, como laços e variáveis, permitiriam a composição de peças musicais mais complexas.
  - Um retorno (*feedback*) da execução do script na interface gráfica precisa ser fornecido ao usuário. Na implementação atual, o usuário acompanha a execução apenas pelas mensagens de depuração impressas no console de texto.
  - As interfaces interativa e de script poderiam ser integradas, ou seja, os agentes criados em uma interface poderiam ser manipulados pela outra.
- É fundamental também a construção de novas aplicações que explorem melhor a interação humano-agente e agente-agente e a mobilidade dos agentes. A mobilidade pode existir entre computadores no mesmo ambiente e entre computadores presentes em espaços físicos geograficamente distantes. Essas novas aplicações trarão para a infra-estrutura a necessidade de mais extensões. Essas extensões, por sua vez, trarão novas idéias e possibilidades para aplicações.

Referências Bibliográficas

[Arnold et al., 2000] Arnold, K., Gosling, J., and Holmes, D. (2000). *The Java(TM) Programming Language*. Addison-Wesley (3rd Edition).

[Birrell and Nelson, 1984] Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59.

[Boulanger, 2000] Boulanger, R., editor (2000). *The CSound Book*. The MIT Press.

[Carzaniga et al., 1997] Carzaniga, A., Picco, G. P., and Vigna, G. (1997). Designing distributed applications with a mobile code paradigm. In Taylor, R., editor, *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32, Boston, MA, USA. ACM Press.

[Casey and Smaragdis, 1996] Casey, M. and Smaragdis, P. (1996). NetSound. In *Proceedings of the 1996 International Computer Music Conference*, Hong Kong.

[Fonseka, 2000] Fonseka, J. R. (2000). Musical agents. Honours Thesis, Monash University.

[Forman and Zahorjan, 1994] Forman, G. H. and Zahorjan, J. (1994). The challenges of mobile computing. *IEEE Computer*, 27(4):38–47.

[Franklin and Graesser, 1996] Franklin, S. and Graesser, A. (1996). Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–35. Springer-Verlag.

[Fuggetta et al., 1998] Fuggetta, A., Picco, G. P., and Vigna, G. (1998). Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

[Gray et al., 1998] Gray, R. S., Kotz, D., Cybenko, G., and Rus, D. (1998). D'Agents: Security in a Multiple-Language, Mobile-Agent System. In Vigna, G., editor, *Mobile Agents and Security*, volume LNCS 1419, pages 154–187. Springer-Verlag.

[Hayes, 1999] Hayes, C. C. (1999). Agents in a nutshell - a very brief introduction. *Knowledge and Data Engineering*, 11(1):127–132.

[Henning and Vinoski, 1999] Henning, M. and Vinoski, S. (1999). *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc.

[Hiller, 1970] Hiller, L. (1970). Music composed with computers: A historical survey. In Lincoln, H., editor, *The Computer and Music*, pages 42–96. Ithaca, NY: Cornell University Press.

[Iazzetta, 2003] Iazzetta, F. (2003). A performance interativa em Pele. In *Proceedings of the 9th Brazilian Symposium on Computer Music*, pages 249–255, Campinas, Brazil.

[Jansen and Karygiannis, 1999] Jansen, W. and Karygiannis, T. (1999). Mobile agent security. National Institute of Standards and Technology, Special Publication 800-19.

[Jennings and Wooldridge, 1998] Jennings, N. R. and Wooldridge, M. J. (1998). Applications of intelligent agents. In Jennings, N. R. and Wooldridge, M. J., editors, *Agent Technology: Foundations, Applications, and Markets*, pages 3–28. Springer-Verlag: Heidelberg, Germany.

[Johansen et al., 1994] Johansen, D. et al. (1994). Operating system support for mobile agents. Technical Report TR94-1468, Department of Computer Science, Cornell University, USA.

[Johansen et al., 1995] Johansen, D. et al. (1995). An introduction to the TACOMA distributed system version 1.0. Technical Report 95-23, University of Tromsø, Norway.

[Johansen et al., 2002] Johansen, D. et al. (2002). A TACOMA retrospective. *Software - Practice and Experience*, 32(6):605–619.

[Joy et al., 2000] Joy, B., Steele, G., Gosling, J., and Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley (2nd Edition).

[Kon and Iazzetta, 1998] Kon, F. and Iazzetta, F. (1998). Internet music: Dream or (virtual) reality? In *Proceedings of the 5th Brazilian Symposium on Computer Music*, Belo Horizonte, Brazil.

[Kotz and Gray, 1999] Kotz, D. and Gray, R. S. (1999). Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13.

[Lange and Oshima, 1998a] Lange, D. B. and Oshima, M. (1998a). Mobile agents with Java: The Aglet API. *World Wide Web*, 1(3).

[Lange and Oshima, 1998b] Lange, D. B. and Oshima, M. (1998b). *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley.

[Lange and Oshima, 1999] Lange, D. B. and Oshima, M. (1999). Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89.

[Loy, 1989] Loy, D. G. (1989). Composing with computers — a survey of some compositional formalisms and music programming languages. In Mathews, M. V. and Pierce, J. R., editors, *Current Directions in Computer Music Research*, pages 292–396. The MIT Press.

[Mateus and Loureiro, 1998] Mateus, G. R. and Loureiro, A. A. F. (1998). Introdução à computação móvel. In *11a Escola de Computação*. COPPE/Sistemas, NCE/UFRJ, Rio de Janeiro.

[Miranda, 2001] Miranda, E. R. (2001). *Composing Music with Computers*. Oxford (UK): Focal Press.

[Moore, 1988] Moore, F. R. (1988). The dysfunctions of MIDI. *Comput. Music J.*, 12(1):19–28.

[OMG, 1998] OMG (1998). *CORBAservices: Common Object Services Specification*. Object Management Group, Framingham, MA. OMG Document 98-12-09.

[OMG, 2002] OMG (2002). *CORBA v3.0 Specification*. Object Management Group, Needham, MA. OMG Document 02-06-33.

[Puckette, 1994] Puckette, M. (1994). Is there life after MIDI? In *Proceedings of the 1994 International Computer Music Conference*, page 2, Aarhus, Denmark.

[Roads, 1996] Roads, C. (1996). *The Computer Music Tutorial*. The MIT Press.

[Rowe, 1993] Rowe, R. (1993). *Interactive Music Systems*. The MIT Press.

[Russell and Norvig, 2003] Russell, S. and Norvig, P. (2003). *Artificial Intelligence - A Modern Approach*. Prentice-Hall (2nd Edition).

[Siegel, 2000] Siegel, J. (2000). *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, Inc.

[Stojmenovic, 2002] Stojmenovic, I., editor (2002). *Handbook of wireless networks and mobile computing*. John Wiley & Sons, Inc.

[Tanenbaum and Steen, 2002] Tanenbaum, A. S. and Steen, M. V. (2002). *Distributed Systems: Principles and Paradigms*. Upper Saddle River, NJ: Prentice-Hall.

[Ueda and Kon, 2003] Ueda, L. K. and Kon, F. (2003). Andante: A mobile musical agents infrastructure. In *Proceedings of the 9th Brazilian Symposium on Computer Music*, pages 87–94, Campinas, Brazil.

[Ueda and Kon, 2004a] Ueda, L. K. and Kon, F. (2004a). Andante: Composition and performance with mobile musical agents. In *Proceedings of the International Computer Music Conference 2004*, Miami, USA.

[Ueda and Kon, 2004b] Ueda, L. K. and Kon, F. (2004b). Mobile musical agents - the Andante Project (extended abstract). In *OOPSLA'04 Companion*, Vancouver, Canada.

[Vigna, 1998] Vigna, G., editor (1998). *Mobile Agents and Security*, volume 1419 of *LNCS State-of-the-Art Survey*. Springer-Verlag.

[Vigna, 2004] Vigna, G. (2004). Mobile Agents: Ten Reasons For Failure. In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM '04)*, pages 298–299, Berkeley, CA. Position Paper.

[Waldo et al., 1997] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany.

[Weiser, 1991] Weiser, M. (1991). The computer for the twenty-first century. *Scientific American*, pages 94–10. Disponível em <`http://www.ubiq.com/hypertext/weiser/UbiHome.html`> (último acesso em setembro/2004).

[Wooldridge, 1998] Wooldridge, M. (1998). Agent-based Computing. *Interoperable Communication Networks*, 1(1):71–97.

[Wright and Freed, 1997] Wright, M. and Freed, A. (1997). Open SoundControl: A new protocol for communicating with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, Thessaloniki, Greece.

[Xenakis, 1971] Xenakis, I. (1971). *Formalized Music*. Bloomington: Indiana University Press.

Referências a Sítios na Internet

[Aglets, sítio] Aglets. `http://aglets.sourceforge.net`. Último acesso em setembro/2004.

[ALSA, sítio] Advanced Linux Sound Architecture. `http://www.alsa-project.org`. Último acesso em setembro/2004.

[Andante, sítio] Projeto Andante. `http://gsd.ime.usp.br/andante`. Último acesso em setembro/2004.

[Applets, sítio] Java Applets. `http://java.sun.com/applets`. Último acesso em setembro/2004.

[Cycling '74, sítio] Cycling '74: Max/MSP. `http://www.cycling74.com/products/maxmsp.html`. Último acesso em setembro/2004.

[DASE, sítio] Distributed Audio Sequencer. `http://www.soundbyte.org/main.php?s=articles&artid=135`. Último acesso em setembro/2004.

[Global Visual Music, sítio] Global Visual Music. `http://visualmusic.org/gvm.htm`. Último acesso em setembro/2004.

[GPL, sítio] GNU General Public License. `http://www.gnu.org/copyleft/gpl.html`. Último acesso em setembro/2004.

[Grasshopper, sítio] Grasshopper. `http://www.grasshopper.de`. Último acesso em setembro/2004.

[IBM Aglets, sítio] Sítio dos Aglets da IBM. `http://www.trl.ibm.com/aglets`. Último acesso em setembro/2004.

[Incubadora, sítio] Página do Andante na Incubadora Virtual da FAPESP. `http://incubadora.fapesp.br/projects/andante`. Último acesso em setembro/2004.

[IRCAM, sítio] Institut de Recherche et Coordination Acoustique/Musique. `http://www.ircam.fr`. Último acesso em setembro/2004.

[Java, sítio] Java. `http://www.java.sun.com`. Último acesso em setembro/2004.

[Java Sound API, sítio] Java Sound API. `http://java.sun.com/products/java-media/sound`. Último acesso em setembro/2004.

[MIDI, sítio] MIDI Manufacturers Association. `http://www.midi.org`. Último acesso em setembro/2004.

[OMG, sítio] Object Management Group. `http://www.omg.org`. Último acesso em setembro/2004.

[OSC, sítio] OpenSound Control. `http://www.cnmat.berkeley.edu/OpenSoundControl`. Último acesso em setembro/2004.

[Real Audio, sítio] Real Audio. `http://www.real.com`. Último acesso em setembro/2004.

[RTSJ, sítio] The Real Time Specification for Java. `https://rtsj.dev.java.net`. Último acesso em setembro/2004.

[Timesys, sítio] Timesys Java - Reference Implementation. `http://www.timesys.com/index.cfm?bdy=java_bdy_ri.cfm`. Último acesso em setembro/2004.

[Tritonus, sítio] Tritonus: Open Source Java Sound. `http://www.tritonus.org`. Último acesso em setembro/2004.

Índice Remissivo