Adaptação Dinâmica Utilizando Agentes Móveis em Computação Ubíqua

Roberto Speicys Cardoso

Dissertação apresentada ao

Instituto de Matemática e Estatística

da Universidade de São Paulo

para obtenção do grau de

Mestre em Ciência da Computação

Área de Concentração: Sistemas Distribuídos

Orientador: Prof. Dr. Fabio Kon

São Paulo, Agosto de 2005

Adaptação Dinâmica Utilizando Agentes Móveis em Computação Ubíqua

Este exemplar corresponde à redação da dissertação final corrigida e defendida por Roberto Speicys Cardoso e aprovada pela comissão examinadora.

São Paulo, Agosto de 2005

Banca examinadora:

- Prof. Dr. Fabio Kon (orientador) - IME/USP
- Prof. Dr. Alfredo Goldman vel Lejbman - IME/USP
- Prof. Dr. Renato Fontoura de Gusmão Cerqueira - PUC-RJ

Suplentes:

- Prof. Dr. Fabio M. Costa - UFG
- Prof. Dr. Francisco Reverbel - IME/USP

Agradecimentos

Agradeço primeiramente aos meus pais e meu irmão que me ajudaram e incentivaram durante estes anos do mestrado. Agradeço também à minha namorada, que foi compreensiva e me apoiou em todas as vezes que tive que me privar da sua companhia em função deste trabalho e que me ajudou a conseguir algumas horas a mais para me dedicar a esta pesquisa.

Agradeço também aos meus companheiros de GSD por todas as conversas e dicas que me foram dadas durante as reuniões, fondues e demais encontros. Agradeço especialmente ao Jeferson que me ajudou e incentivou persistentemente por ICQ durante as piores fases e ao Nelson que se dispôs a abrir mão de seus compromissos particulares para me ajudar em minha pesquisa em turnos de 24 horas. Não posso me esquecer dos administradores de rede Alexandre, Braga e Dairton que fizeram o melhor para que eu concluísse este trabalho, nem do Prof. Alfredo Goldman, cuja convivência por uma semana contribuiu muito para o meu amadurecimento como pesquisador.

Esta pesquisa não poderia ser realizada sem o apoio da empresa em que trabalho, a F9C Security. Agradeço a todos que trabalharam e que trabalham lá por compreenderem minhas ausências e por se adaptarem à minha agenda. Sou grato a todos, principalmente ao Jorge e ao Estevão pela oportunidade dada e ao Guto que não mediu esforços para me ajudar sempre que precisei.

Sou grato também a todos os meus amigos que, em um jogo de futebol, numa mesa de bar, num ensaio ou comendo um sushi me incentivaram, apoiaram e sonharam comigo. Vocês são muito importantes para mim e este trabalho tem um pedacinho de cada um de vocês.

Finalmente, meus agradecimentos mais sinceros ao meu orientador, Prof. Fabio Kon. Sua paciência, disponibilidade, conhecimento e força de vontade são exemplos que eu levarei por toda a vida. Obrigado pelo seu imensurável apoio e por todas oportunidades que me foram dadas.

Resumo

Aplicações distribuídas com necessidades de qualidade de serviço (QoS) como algumas aplicações Multimídia e de Computação Ubíqua podem se beneficiar bastante do fornecimento de garantias de QoS por parte do sistema ou da infra-estrutura de middleware. Estas aplicações devem evitar possíveis falhas em sua saída que afetem a percepção do usuário.

A maior parte da pesquisa em garantias de QoS para sistemas distribuídos foca três aspectos do gerenciamento de QoS: controle de admissão, reserva de recursos e agendamento de reservas. Entretanto, em ambientes altamente dinâmicos e distribuídos, formas eficientes de negociação e re-negociação de QoS também são fundamentais.

Nós acreditamos que os agentes móveis, devido a sua inerente flexibilidade e agilidade, podem ter um papel importante neste cenário, especialmente durante o processo de adaptação da aplicação. Nós desenvolvemos uma infra-estrutura baseada em agentes móveis capaz de fornecer serviços como monitoração de recursos, *brokering* de contratos de QoS e garantia de requisitos. Além disso, nossa arquitetura possui um mecanismo poderoso de negociação de QoS.

Neste trabalho nós descrevemos a arquitetura e a implementação de um protótipo desta infra-estrutura. Nós discutimos as motivações e os trabalhos relacionados, apresentamos a arquitetura e discutimos questões relacionadas à implementação deste protótipo. Finalmente nós introduzimos duas aplicações de exemplo, apresentamos nossos resultados experimentais e discutimos trabalhos futuros.

Abstract

QoS-aware distributed applications such as certain Multimedia and Ubiquitous Computing applications can benefit greatly from the provision of QoS guarantees from the underlying system and middleware infrastructure. They must avoid execution glitches that affect the user's perception of the application output.

Most research in QoS support for distributed systems focuses on three aspects of QoS management: admission control, resource reservation, and scheduling. However, in highly dynamic distributed environments, effective means for QoS negotiation and re-negotiation are also essential.

We believe that mobile agents, due to its inherent flexibility and agility, can play an important role in this scenario, specially during the application adaptation process. We designed a mobile-agent-based infrastructure that provides services such as resource monitoring, QoS brokering, and QoS enforcement. Furthermore, our infrastructure offers a powerful mechanism for QoS negotiation.

In this work, we describe the architecture and prototype implementation of this infrastructure. We discuss the motivations, related works, present the architectural design and discuss implementation issues concerning the infrastructure prototype. Finally, we introduce two sample applications, present some experimental results and discuss future work.

Sumário

1 Introdução **1**
- 1.1 A Necessidade de Mobilidade . . . 2
- 1.2 A Importância da Qualidade de Serviço . . . 2
- 1.3 Requisitos da Arquitetura . . . 4
- 1.4 Modelo da Arquitetura . . . 5

2 Trabalhos Relacionados **8**
- 2.1 Sistemas para Computação Ubíqua . . . 8
  - 2.1.1 Gaia - um sistema operacional para espaços ativos . . . 8
  - 2.1.2 one.world . . . 11
  - 2.1.3 Ninja . . . 12
  - 2.1.4 Context Toolkit . . . 13
  - 2.1.5 Oxygen . . . 15
  - 2.1.6 Jini . . . 16
- 2.2 Sistema de Garantias de Qualidade de Serviço – QuO . . . 18
- 2.3 Sistema para Monitoração de Recursos – ReMoS . . . 20
- 2.4 Conclusão . . . 21

3 Tecnologias Utilizadas **22**
- 3.1 Aglets - Ambiente para Desenvolvimento de Agentes Móveis . . . 22
  - 3.1.1 Estrutura da API de um Aglet . . . 22
  - 3.1.2 Segurança . . . 24
  - 3.1.3 Exemplo de agente utilizando o ASDK . . . 24
- 3.2 DSRT - Sistema de Garantia de Tempo Real Flexível . . . 24
  - 3.2.1 Servidor de Processamento . . . 26
  - 3.2.2 Servidor de Memória . . . 27
- 3.3 QML - Linguagem para Especificação de Requisitos de Qualidade de Serviço . . . 28
  - 3.3.1 Estrutura da Linguagem . . . 28
  - 3.3.2 Exemplo . . . 29
  - 3.3.3 QML *versus* XML . . . 31
- 3.4 Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos . . . 31
  - 3.4.1 Monitoração . . . 31
  - 3.4.2 Detecção de Eventos . . . 32
  - 3.4.3 Reconfiguração Dinâmica . . . 32
- 3.5 Conclusão . . . 32

4 Negociação de Requisitos de Qualidade de Serviço (*QoS*) 34
4.1 Definição de Requisitos . . . 35
4.1.1 Implementação . . . 37
4.2 *Broker* de Recursos . . . 39
4.2.1 Aspectos Importantes dos *Leases* . . . 40
4.2.2 Implementação . . . 41
4.3 Processo de Negociação . . . 43
4.3.1 Estratégias de Procura . . . 43
4.4 Integração com as Aplicações . . . 44

5 Mecanismo de Adaptação 46
5.1 Monitoração . . . 47
5.2 Adaptação . . . 49
5.3 Integração com as Aplicações . . . 51

6 Aplicações 53
6.1 Refletor de Áudio/Vídeo . . . 53
6.2 Implementação do Refletor de Áudio/Vídeo . . . 54
6.3 Multiplexador de Áudio . . . 58

7 Experimentos 62
7.1 Descrição do Ambiente de Testes . . . 62
7.2 Experimentos Realizados . . . 63
7.3 Resultados dos Experimentos . . . 64

8 Principais Contribuições 71
8.1 Arquitetura de Negociação . . . 71
8.2 *Parser* QML para o DSRT . . . 71
8.3 *Broker* CORBA para o DSRT . . . 72
8.4 Artigos Publicados . . . 73
8.5 Trabalhos Futuros . . . 74
8.5.1 Reserva de Requisitos de Rede . . . 74
8.5.2 Definição do Posicionamento dos Agentes . . . 74
8.5.3 Estratégia de Procura *peer-to-peer* . . . 75
8.5.4 Integração com o Gaia . . . 75
8.6 Conclusões . . . 76

Lista de Figuras

1.1 Modelo da Arquitetura 6

2.1 Estrutura do Gaia 9
2.2 Estrutura do one.world 11
2.3 Estrutura do Ninja 12
2.4 Estrutura do Context Toolkit 14
2.5 Estrutura do Jini 17
2.6 Estrutura do QuO 19

3.1 Código exemplo de um Aglet 25
3.2 Interface do serviço de taxas de câmbio 29
3.3 Especificação de QoS do serviço de taxas de câmbio 30

4.1 Definição em QML dos tipos de contrato aceitos pelo DSRT 36
4.2 Definição em QML do contrato e perfil de QoS de uma aplicação 37
4.3 Definição dos tokens no JFlex 38
4.4 Definição da gramática no BYacc/J 38
4.5 Comunicação entre o agente de negociação, *broker* de recursos e o DSRT 39
4.6 Interação do Agente de Negociação com o *broker* 42
4.7 UML das estratégias de procura 44
4.8 Mensagens trocadas durante uma negociação 45

5.1 Possíveis estratégias de adaptação 46
5.2 Diagrama de funcionamento do Arcabouço 47
5.3 Integração com o Arcabouço 48
5.4 Diagrama UML das estratégias de adaptação 49
5.5 Implementação da estratégia de migração 50
5.6 Integração da aplicação com a estratégia de migração 51

6.1 Arquitetura típica de envio de conteúdo multimídia pela Internet 53
6.2 Especificação de requisitos do Refletor Multimídia 54
6.3 Criação do refletor multimídia 56
6.4 Chegada do refletor multimídia a um novo nó 56
6.5 Verificação dos eventos detectados pelo Arcabouço 57
6.6 Início do processo de adaptação 57
6.7 Estratégia de adaptação - Migração 58
6.8 Conferência utilizando apenas refletores 59

6.9 Conferência utilizando o mixer de áudio 60

7.1 Ambiente utilizado para testes 62
7.2 Tempo de negociação da estratégia linear em cada cenário 64
7.3 Total por etapa da estratégia linear - Cenário 1 65
7.4 Total por etapa da estratégia linear - Cenário 2 65
7.5 Total por etapa da estratégia linear - Cenário 3 66
7.6 Tempo de negociação da estratégia paralela em cada cenário 66
7.7 Total por etapa da estratégia paralela - Cenário 1 67
7.8 Total por etapa da estratégia paralela - Cenário 2 68
7.9 Total por etapa da estratégia paralela - Cenário 3 68
7.10 Comparação entre estratégia paralela e linear - Cenário 1 69
7.11 Comparação entre estratégia paralela e linear - Cenário 2 69
7.12 Comparação entre estratégia paralela e linear - Cenário 3 70

8.1 Funcionamento do *broker* original do DSRT comparado ao *broker* desenvolvido neste trabalho 72

Lista de Tabelas

3.1 Estrutura de Escalonamento 26

Capítulo 1

Introdução

O artigo publicado por Mark Weiser na revista *Scientific American* em Setembro de 1991 [Wei91] propôs um novo desafio à Ciência da Computação: como integrar computadores ao nosso dia-a-dia de forma imperceptível, tornando seu uso tão natural quanto outras tecnologias já estabelecidas como, por exemplo, a escrita. No cenário proposto por Weiser, computadores são utilizados naturalmente por usuários para realizar tarefas corriqueiras em casa e no trabalho, com o objetivo de facilitar sua execução e de permitir diversas possibilidades que não exisitiriam sem o seu uso. Esta aplicação da computação e os problemas a ela relacionados foram chamados de *Computação Ubíqua*.

Para que fosse possível a integração de milhares de dispositivos computacionais em nosso cotidiano, de acordo com Weiser, diversas mudanças deveriam ser realizadas na infra-estrutura computacional desenvolvida até então. Sistemas operacionais tradicionais não estariam preparados para trabalhar simultaneamente com milhares de dispositivos, que podem entrar e sair do sistema a qualquer instante. Equipamentos de alto processamento e de baixo custo e consumo de energia deveriam ser desenvolvidos para poderem ser espalhados pelo ambiente. As conexões de rede entre os equipamentos deveriam estar preparadas para conectar centenas de dispositivos em um pequeno espaço físico e deveriam possuir mecanismos que fossem capazes de utilizar as informações de mobilidade dos dispositivos. Um ambiente capaz de integrar estes diversos componentes computacionais para estender as percepções de um usuário e suas interações com o espaço ao redor é conhecido como um espaço ativo [RHC+02].

O desenvolvimento de sistemas com estas propriedades apresenta diversos problemas em áreas como Interfaces Homem-Computador e Redes. Dentro da área de pesquisa de Sistemas Distribuídos, o maior desafio é desenvolver um sistema operacional capaz de gerenciar os dispositivos existentes em um ambiente de computação ubíqua, bem como de oferecer os serviços necessários ao funcionamento deste ambiente. Diversas iniciativas foram tomadas com este objetivo [RC00, DA00, GDL+01, GWS01] e apenas agora alguns resultados começam a aparecer. Um sistema com diversas funcionalidades interessantes e que começa a ser testado com aplicações ubíquas é o Gaia, desenvolvido pela Universidade de Illinois em Urbana-Champaign. Apesar de ainda não possuir uma distribuição pública, já foram desenvolvidos o núcleo do sistema e alguns serviços como os serviços de Localização [RAMC+04], Sistema de Arquivos, Segurança, Presença e Qualidade de Serviço (QoS)[1].

[1] http://gaia.cs.uiuc.edu/html/projects.htm

1.1 A Necessidade de Mobilidade

Os sistemas de computação ubíqua se diferenciam dos sistemas computacionais tradicionais de várias formas. A possibilidade de se construir um ambiente contendo diversos equipamentos computacionais cria alternativas de uso totalmente distintas das aplicações tradicionais de computadores pessoais. A forma de interação entre usuários e aplicação precisa ser redefinida e as alternativas de cooperação entre os usuários de um tal ambiente devem ser incentivadas e facilitadas pela infra-estrutura.

Como resultado, uma das principais características das aplicações de computação ubíqua é o foco no usuário. Estas aplicações devem ser desenhadas com o objetivo de permitir a utilização do espaço por um usuário leigo, que deve concentrar seus esforços na realização de uma atividade e não nos detalhes de utilização dos equipamentos.

Para facilitar esta interação, todas informações disponíveis devem ser utilizadas. Em particular, as informações contextuais podem simplificar a comunicação entre usuários e aplicações; já as informações físicas podem auxiliar as aplicações a se adaptarem dinamicamente. O foco no usuário, entretanto, cria uma nova necessidade: a mobilidade das aplicações.

Usuários de espaços ativos são naturalmente móveis. Eles podem sair de uma reunião para atender a um telefonema, ou deixar sua casa em direção ao trabalho. As aplicações de espaços ativos devem se adaptar a estas movimentações, acompanhando o usuário por onde ele for.

Em [RC02] são definidos dois tipos de mobilidade de aplicações neste cenário: a movimentação entre espaços ativos e a movimentação dentro de um mesmo espaço ativo. A movimentação entre espaços ativos é normalmente utilizada para que as aplicações possam seguir o usuário por onde ele for. As anotações feitas durante uma reunião podem ser transferidas para a sala particular de um usuário, por exemplo. Já a movimentação dentro de um mesmo espaço ativo é utilizada para explorar a diversidade de dispositivos de um tal espaço. Uma apresentação mostrada em um monitor para poucas pessoas pode migrar para uma tela maior conforme aumente a quantidade de usuários.

A infra-estrutura de software deve prover mecanismos que possibilitem e facilitem a movimentação das aplicações. Além disso, ela deve garantir que a migração das aplicações seja feita de forma eficiente, respeitando os requisitos de qualidade da aplicação em particular, e dos espaços ativos em geral. É indesejável, por exemplo, que uma apresentação, ao se transferir de um monitor pequeno para uma tela grande, gere atrasos na sua utilização. As aplicações devem utilizar os recursos disponíveis para facilitar a interação com os usuários, mas sem comprometer a usabilidade do sistema.

Nossa pesquisa visa propor soluções para este problema: como garantir às aplicações os recursos necessários para sua execução durante a sua movimentação entre dispositivos e entre espaços ativos.

1.2 A Importância da Qualidade de Serviço

As questões relativas à qualidade de serviço em um espaço ativo são de fundamental importância. O sucesso de um espaço ativo está diretamente relacionado à percepção que o usuário tem do ambiente. Em tal espaço, os computadores devem "desaparecer": os usuários devem utilizar os equipamentos naturalmente, desapercebidos da presença de dispositivos computacionais. As aplicações devem evitar falhas de execução que alterem a percepção dos usuários em relação ao ambiente e devem responder prontamente aos seus estímulos.

O gerenciamento da qualidade de serviço das aplicações não é importante apenas em espaços ativos e ambientes de computação ubíqua. Aplicações multimídia distribuídas (tais como transmissões de vídeo e áudio em redes de computadores) também devem evitar perdas de desempenho que possam afetar o resultado da aplicação e a experiência do usuário. Serviços que facilitem o gerenciamento de qualidade de serviço auxiliam o desenvolvimento de aplicações com estas características.

A pesquisa atual em qualidade de serviço se concentra em resolver problemas relacionados a controle de admissão, reserva de recursos e escalonamento. Entretanto, em ambientes altamente distribuídos como espaços ativos ou aplicações multimídia móveis, estes serviços não são suficientes para criar uma estrutura eficaz de gerenciamento da qualidade de serviço. Em tais circunstâncias, os problemas relacionados à *negociação* da qualidade de serviço se tornam importantes, demandando um tratamento especial por parte do middleware.

Mecanismos de negociação de qualidade de serviço são um requisito fundamental em ambientes altamente distribuídos e dinâmicos. Nestes ambientes, a disponibilidade de recursos e as necessidades de QoS dos usuários finais variam muito ao longo do tempo. As aplicações preparadas para trabalhar com requisitos de qualidade de serviço devem, portanto, negociar e *renegociar* suas necessidades para manter a sua qualidade. O Serviço de Localização de um Sistema de Computação Ubíqua, por exemplo, deve responder rapidamente às requisições dos clientes para evitar que todo ambiente fique inoperante. As suas necessidades de recursos podem variar de acordo com o número de clientes, a quantidade de entidades e a disponibilidade de recursos do ambiente. O sistema que serve de base para as aplicações deve prover mecanismos que permitam ao Serviço de Localização negociar seus requisitos de qualidade de serviço a fim de se adaptar às constantes mudanças.

O processo de negociação de requisitos em um sistema distribuído envolve a troca de diversas mensagens entre os nós participantes da negociação. A arquitetura tradicional de cliente/servidor para negociação de requisitos de qualidade de serviço é problemática em ambientes altamente distribuídos e dinâmicos por diversas razões. A negociação baseada na arquitetura cliente/servidor pode gerar um alto volume de tráfego de rede, impactando o ambiente distribuído como um todo. Além disso, devido ao grande número de possibilidades de configurações de qualidade de serviço em um ambiente distribuído, o modelo cliente/servidor é extremamente ineficiente, pois as diversas possibilidades de configuração implicariam em um grande número de mensagens trocadas entre cliente e servidor,

As características inerentes aos agentes móveis [LA99] fazem deles uma opção interessante para resolver diversos problemas relacionados ao gerenciamento de qualidade de serviço em sistemas distribuídos dinâmicos, especialmente para resolver os problemas associados à negociação de requisitos de qualidade de serviço. Agentes móveis são programas capazes de suspender sua execução e migrar para outros nós da rede, prosseguindo sua execução no novo nó a partir do ponto em que foram interrompidos [KG99].

Nossa intenção inicial era desenvolver um sistema capaz de atacar os problemas relacionados à negociação de requisitos de qualidade de serviço e integrá-lo aos sistemas de computação ubíqua em desenvolvimento atualmente. Infelizmente, nenhum destes sistemas está maduro o suficiente para possuir uma distribuição pública. Desta forma, desenvolvemos nossa arquitetura paralelamente e esperamos, no futuro, poder integrá-la com os sistemas citados anteriormente.

A arquitetura apresentada neste trabalho utiliza as propriedades dos agentes móveis para oferecer às aplicações, mecanismos eficientes de adaptação e de negociação de requisitos de qualidade de serviço. A arquitetura também fornece outros mecanismos relacionados ao gerenciamento da qualidade de serviço, tais como: especificação de requisitos, *brokering* de QoS, controle de admissão

e garantia de qualidade de serviço.

Nas próximas seções faremos uma introdução ao sistema desenvolvido detalhando sua arquitetura e descrevendo, de forma geral, cada um de seus componentes com ênfase no principal foco do nosso trabalho: a negociação de requisitos de qualidade de serviço. Uma descrição detalhada do funcionamento deste sistema pode ser encontrada no Capítulo 4. O Capítulo 2 identifica trabalhos relacionados com o tema abordado nesta tese, enquanto o Capítulo 3 descreve as tecnologias utilizadas para a implementação do sistema. O Capítulo 5 detalha os mecanismos de adaptação possibilitados pela nossa arquitetura. Duas aplicações de referência que utilizam nosso sistema para negociação de requisitos são descritas no Capítulo 6 e os resultados dos experimentos realizados com elas são apresentados no Capítulo 7. Finalmente, o Capítulo 8 lista as principais contribuições do nosso trabalho, os próximos passos em nossos estudos e relata as conclusões obtidas durante a realização desta pesquisa.

1.3 Requisitos da Arquitetura

A arquitetura que vislumbramos tem como principal objetivo facilitar o gerenciamento de qualidade de serviço de aplicações adaptáveis dinamicamente. Ambientes de Computação Ubíqua estão sempre mudando de forma rápida. As aplicações existentes em um tal ambiente devem se adaptar a estas mudanças para trazer o máximo de qualidade à experiência do usuário. Nosso sistema foi desenvolvido tendo em vista três requisitos fundamentais: escalabilidade, flexibilidade e completude.

Para atender às necessidades de sistemas altamente distribuídos e dinâmicos, como os espaços ativos, nossa solução deve ser escalável. Nesta nova classe de sistemas, centenas ou até mesmo milhares de computadores estão conectados através de canais de comportamento imprevisível. A infra-estrutura deve ser capaz de gerenciar uma grande quantidade de equipamentos e deve ser capaz de incluir todos eles no processo de gerenciamento de qualidade de serviço.

Flexibilidade também é um requisito importante para nossa arquitetura. Aplicações com propósitos distintos e utilizando técnicas de adaptação diferentes devem ser capazes de utilizar o serviço de gerenciamento de qualidade de serviço da forma que julgarem mais apropriada. O processo de integração da aplicação com a infra-estrutura não deve causar grandes mudanças arquiteturais em nenhum dos sistemas envolvidos. Trabalhos anteriores [VZL+98] e nossa própria experiência com o desenvolvimento de sistemas distribuídos sugerem que, para aumentar a quantidade de aplicações capazes de utilizar a infra-estrutura e para facilitar sua adaptação às mudanças, é necessário que as aplicações tenham conhecimento das alterações do ambiente que podem disparar processos de adaptação. Tendo controle completo do processo de adaptação, as aplicações podem decidir qual a melhor estratégia a ser utilizada. Esta decisão é totalmente dependente do propósito da aplicação e a infra-estrutura não pode assumir a responsabilidade de escolher o melhor método de adaptação em seu benefício.

Finalmente, nossa solução deve oferecer uma solução completa para o gerenciamento de qualidade de serviço. Nossa visão de uma solução completa é similar às apresentadas em [VZL+98, NXWL01]: as aplicações devem ser capazes de especificar requisitos de QoS, monitorar o nível de qualidade de serviço fornecido e adaptar-se às mudanças de necessidades de QoS. A infra-estrutura deve então fornecer mecanismos para especificação, garantia e monitoramento de qualidade de serviço, além de mecanismos de adaptação para as aplicações. Idealmente, esta solução também deveria fornecer mecanismos para garantia de QoS de ponta a ponta, envolvendo também a garantia de QoS no tráfego de rede. Esta área de pesquisa entretanto é muito extensa, e diversos trabalhos

de qualidade [GP99, CC97] já foram realizados sobre este assunto. Nós decidimos então focar nossa pesquisa no estudo dos problemas de garantia de QoS relacionados apenas aos nós do sistema, sem considerar as conexões entre os nós.

Além dos requisitos descritos nos parágrafos anteriores, existem também outros requisitos específicos de cada serviço fornecido pela infra-estrutura:

- A monitoração de QoS não deve sobrecarregar a rede, já que ela é o recurso que consome mais energia em um dispositivo móvel [SH00]. Além disso, o processo de monitoração não deve afetar o desempenho das aplicações consumindo os recursos que deveriam ser utilizados pela aplicação.

- A especificação de QoS deve ser separada do código-fonte da aplicação, possibilitando alterações da especificação de QoS em tempo de execução, sem interrupção do serviço.

- O sistema de garantia de QoS deve possuir um mecanismo de controle de admissão que informe às aplicações quando suas necessidades de QoS não podem ser atendidas pelo nó. Ele também deve propor contratos alternativos de QoS para análise pela aplicação.

- O sistema de adaptação de QoS deve fornecer meios para negociação e renegociação de parâmetros de QoS. As aplicações preparadas para trabalhar com vários níveis de QoS podem utilizar a negociação de QoS para determinar o melhor nível de QoS disponível no momento. Além disso, em caso de mudanças no ambiente, o mecanismo de renegociação pode ser utilizado para definir um novo nível de QoS que pode exigir mais ou menos recursos que o anterior.

- O mecanismo de negociação de QoS deve ser eficiente e flexível. As aplicações utilizarão este sistema de diversas formas para negociar seus requisitos de QoS com multíplos nós do ambiente distribuído. O sistema de negociação de QoS, portanto, não deve desperdiçar recursos tais como rede e tempo de processamento.

A segurança também é uma preocupação sempre que se utilizam agentes móveis. Neste trabalho que desenvolvemos, entretanto, o nível de segurança fornecido pelos agentes móveis é similar ao fornecido por outras tecnologias relacionadas, como RPC por exemplo. Este assunto já foi amplamente estudado e uma vasta discussão desse tópico está fora do escopo deste trabalho. Maiores detalhes a respeito de segurança com agentes móveis podem ser encontrados em [Vig98].

Para conseguirmos desenvolver uma arquitetura aderente aos requisitos descritos acima, decidimos utilizar sempre que possível sistemas já existentes, preenchendo as lacunas com contribuições originais. Na próxima seção nós apresentamos o modelo da arquitetura, no Capítulo 3 apresentamos as tecnologias utilizadas nesta pesquisa e, finalmente, no Capítulo 8 descrevemos nossas principais contribuições.

1.4 Modelo da Arquitetura

Uma infra-estrutura para gerenciamento de QoS em sistemas altamente dinâmicos e distribuídos encontra diversos problemas que não são solucionados apropriadamente através de modelos tradicionais cliente/servidor, como a negociação de múltiplos níveis de qualidade de serviço e a negociação com vários nós simultaneamente. Agentes móveis podem auxiliar a superar estes obstáculos de diversas formas, com ganhos em agilidade e flexibilidade. Agentes móveis podem encapsular

protocolos, omitindo detalhes de implementação da infra-estrutura para a aplicação; podem ser executados de forma assíncrona evitando longos períodos de atividade conectados e assim diminuir o tráfego de rede; e são compostos de código e dados, permitindo que as aplicações possam enviar agentes por todo ambiente distribuído para realizar ações em seu benefício [LA99].

Nossa arquitetura se utiliza de agentes móveis para executar diversas tarefas relativas à negociação e adaptação de QoS. O modelo da arquitetura pode ser melhor compreendido através da Figura 1.1. Um nó central de gerenciamento, que pode ser replicado, é responsável por concentrar os dados de monitoração dos dispositivos da rede. Usando o padrão *publisher/subscriber* [BMR+96], as aplicações recebem do nó central notificações de eventos com informações sobre o estado das entidades espalhadas por todo sistema distribuído (tais como a utilização de CPU do nó A, uso de memória do nó B, etc.). Através do sistema de monitoração, as aplicações podem ser informadas sobre o nível de QoS de todo o ambiente e podem detectar os melhores momentos para disparar o seu processo de adaptação. Alterações no nível de QoS podem ser causadas por um nó que se torne incapaz de atender aos requisitos de qualidade de serviço de uma aplicação, ou por um aumento na demanda por um serviço específico. Em ambos os casos, o sistema monitora os recursos, detecta estas modificações e informa as partes interessadas para que elas possam alterar seu comportamento e recuperar ou melhorar seus níveis de QoS.

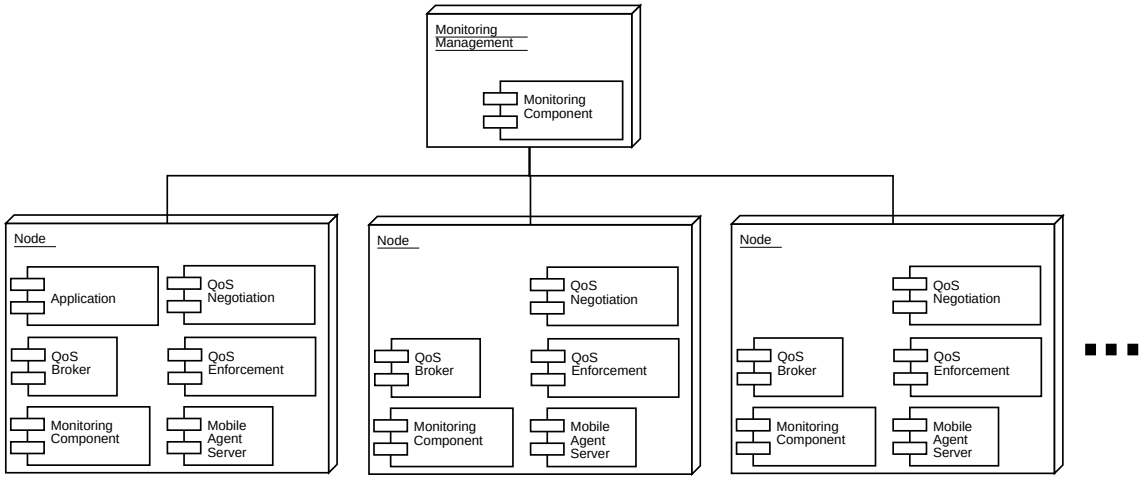

Figura 1.1: Modelo da Arquitetura

Se não for tomado o devido cuidado, o nó central de gerenciamento da monitoração pode se tornar tanto um gargalo quanto um ponto único de falha. Outros trabalhos atacaram estes problemas em sistemas similares, indicando como este nó poderia ser replicado com pequena perda de desempenho [KYH+01]. Além disso, experimentos já realizados com este sistema mostraram que cada nó é capaz de gerenciar facilmente até 100 dispositivos [MK02a]. Acreditamos que os problemas relacionados à escalabilidade e tolerância a falhas do sistema de monitoração possam ser superados. Entretanto, em nossa pesquisa não focamos as formas de resolução destes problemas. Neste trabalho procuramos nos concentrar na área de gerenciamento de qualidade de serviço utilizando agentes móveis.

Cada nó capaz de executar aplicações deve aceitar solicitações de reserva de recursos e deve receber agentes móveis. Eles também devem oferecer garantias de QoS aos serviços que estiverem hospedando. Entretanto, não são todos os nós do ambiente distribuído que devem respeitar estes

requisitos: equipamentos móveis com poucos recursos computacionais não precisam necessariamente hospedar serviços.

Também deve existir um *broker* para negociação de qualidade de serviço em cada nó do sistema. Ele é responsável por intermediar as requisições de recursos entre aplicações concorrentes, atuando como ponto central de negociação para todos os programas que necessitem de garantias de QoS em um dado nó. Desta forma ele pode realizar o controle de admissão, recusando contratos que o seu nó não possa honrar. Além disso, ele também pode impedir que impasses (*deadlocks*) ocorram em situações onde uma única aplicação requisite recursos de todos os nós do sistema, impossibilitando outras aplicações de utilizarem estes recursos.

O processo de negociação de requisitos de QoS é feito inteiramente através de agentes móveis. Quando uma aplicação necessita de uma negociação de QoS, um agente móvel lê o arquivo de especificação da aplicação e envia outros agentes pela rede, de acordo com a estratégia de negociação definida pela aplicação. Estes agentes realizam a negociação localmente em cada nó e enviam os resultados da negociação de volta para a aplicação, que por sua vez analisa os resultados recebidos e decide qual a melhor estratégia de adaptação que deve ser utilizada, de acordo com os seus requisitos operacionais.

Para obter o maior benefício possível da arquitetura, as aplicações também podem ser desenvolvidas utilizando o paradigma de agentes móveis. Isto aumenta a flexibilidade das aplicações, uma vez que elas podem utilizar as funções de clonagem e migração de agentes móveis para se adaptar às mudanças do ambiente. Estas duas técnicas, somadas aos demais mecanismos de adaptação já existentes, permitem o desenvolvimento de múltiplas formas de adaptação transparente para aplicações com necessidades de garantias de QoS. Aplicações já existentes também podem ser encapsuladas em agentes móveis e obterem os mesmos benefícios. Durante a nossa pesquisa, por exemplo, encapsulamos o servidor de nomes do JacORB[2] em um agente móvel com sucesso.

[2]http://www.jacorb.org

Capítulo 2

Trabalhos Relacionados

Neste capítulo descrevemos diversos sistemas relacionados à nossa pesquisa e os motivos pelos quais eles não foram utilizados em nosso trabalho. Começamos pelos sistemas de Computação Ubíqua, que foram a principal motivação de nossa pesquisa. Depois descrevemos outros sistemas interessantes para monitoração de recursos e garantia de qualidade de serviço, que são pontos fundamentais deste estudo.

2.1 Sistemas para Computação Ubíqua

A área de sistemas para Computação Ubíqua é ainda muito recente na computação. O desenvolvimento de sistemas que forneçam a infra-estrutura de software necessária ao funcionamento e ao desenvolvimento de aplicações para espaços ativos é um processo complexo: apenas a partir de 2001 começaram a surgir versões instáveis dos primeiros sistemas com este objetivo.

Um espaço ativo é um espaço físico qualquer (como um escritório, sala de aula, hospital, sala de reuniões ou até mesmo uma cidade) ampliado através da utilização de equipamentos com poder computacional integrados ao ambiente [KHR+00]. Computação Ubíqua é a área que estuda os problemas que surgem em um tal ambiente, relacionados às áreas de Interface com o Usuário, Redes e Sistemas Operacionais, por exemplo.

Nesta seção analisamos alguns dos sistemas para Computação Ubíqua em desenvolvimento hoje em dia, além de outros sistemas de computação distribuída que abordaram alguns dos problemas comuns a aplicações distribuídas e aplicações para espaços ativos. Estes últimos sistemas, apesar de mais estáveis, resolvem apenas uma parte dos diversos problemas encontrados em Computação Ubíqua e não podem ser considerados uma opção para a implementação de espaços ativos.

2.1.1 Gaia - um sistema operacional para espaços ativos

O Gaia[1] é um sistema desenvolvido pela Universidade de Illinois em Urbana-Champaign que procura criar um ambiente computacional em que não existam diferenças entre entidades reais (pessoas, objetos) e entidades virtuais (dados, estruturas). O principal objetivo deste projeto é o de criar um sistema operacional intermediário capaz de gerenciar os recursos de um espaço ativo, assim como os sistemas operacionais tradicionais gerenciam os recursos de um computador [RHR+01].

[1] http://choices.cs.uiuc.edu/gaia

Para atingir este objetivo, foi desenvolvido um sistema operacional, chamado GaiaOS, capaz de integrar e gerenciar todos os dispositivos existentes em um espaço ativo. A partir desta infra-estrutura, pretende-se desenvolver aplicações de computação ubíqua que não precisem conhecer detalhes do hardware do espaço ativo.

Arquitetura

O GaiaOS é um sistema operacional intermediário, que roda como *middleware* sobre os sistemas operacionais existentes, como Windows 2000, Windows CE e Solaris. Ele fornece diversos serviços que facilitam o desenvolvimento de aplicações para espaços ativos. O GaiaOS define um modelo de aplicação que permite alterar dinamicamente a quantidade, a qualidade e a localidade das entradas, saídas e componentes computacionais de uma aplicação [RC03].

A Figura 2.1 mostra a estrutura do Gaia. O núcleo (*kernel*) do Gaia envolve os seguintes componentes:

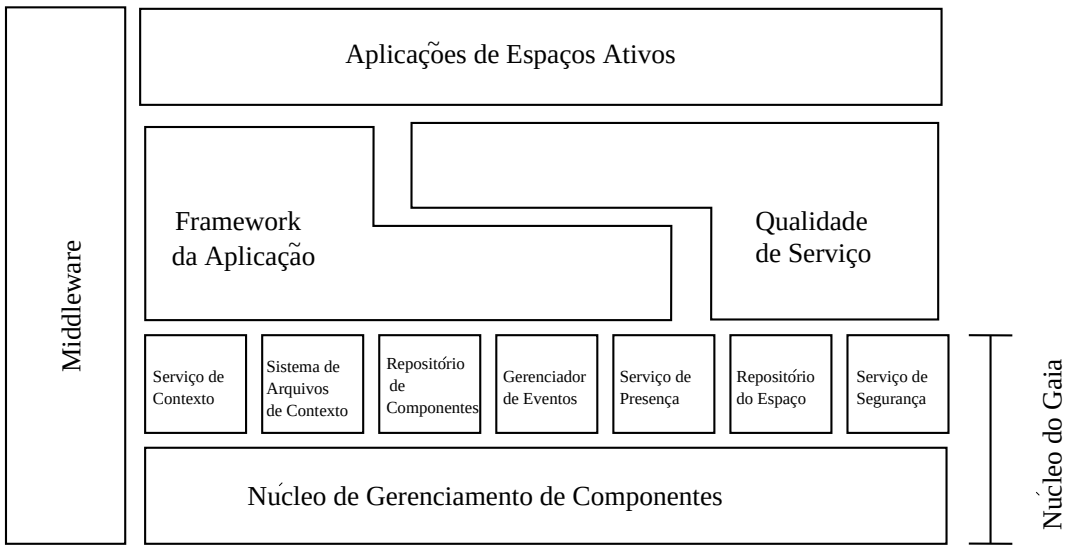

Figura 2.1: Estrutura do Gaia

- **Serviço de Contexto:** serviço responsável por fornecer às aplicações as informações relevantes de contexto. Estas informações podem melhorar a qualidade da interação entre usuários e computadores, uma vez que na interação entre humanos as informações de contexto são usadas naturalmente para simplificar a comunicação.
- **Sistema de Arquivos de Contexto:** o sistema de arquivos de contexto adiciona funções de organização dos dados e de transformação dos dados baseadas em informações contextuais. Os dados são organizados de forma diferente, dependendo da utilização do espaço ativo, para que as informações possam ser encontradas mais facilmente. Áreas de armazenamento pessoal seguem os usuários por onde eles estiverem.

- **Repositório de Componentes:** armazena todos os componentes de software existentes no sistema. Desta forma os nós do espaço ativo podem obter cópias dos componentes necessários para executar uma aplicação, ou atualizar os componentes que já possuem.
- **Gerenciador de Eventos:** fornece mecanismos que facilitam a criação e utilização de canais de eventos. Estes canais são a principal forma de comunicação entre entidades de um espaço ativo.
- **Serviço de Presença:** gerencia as informações relativas à entrada e saída de entidades do espaço ativo. Estas entidades podem ser pessoas, equipamentos, aplicações e serviços. As informações são anunciadas através de canais de eventos especializados, que podem ser utilizados pelas entidades do sistema para identificar mudanças no ambiente.
- **Repositório do Espaço:** armazena em um banco de dados as informações de todas as entidades presentes no espaço ativo em um determinado momento. Ele utiliza as informações anunciadas pelo Serviço de Presença para manter sua base atualizada. Aplicações podem, por exemplo, consultar o repositório para saber quais dispositivos de hardware estão disponíveis para uso.
- **Serviço de Segurança:** fornece mecanismos de controle de acesso às entidades do sistema, separando as aplicações dos sistemas de autenticação. Este serviço também assegura a privacidade dos usuários, possibilitando sua autenticação sem revelar sua real identidade às aplicações e serviços finais.
- **Núcleo de Gerenciamento de Componentes:** fornece meios para a manipulação de componentes, incluindo a criação, a destruição e a carga dinâmica de componentes.

Trabalho em andamento

A equipe do projeto continua trabalhando na implementação do núcleo do sistema operacional. No entanto, a estrutura de software já desenvolvida possibilita o desenvolvimento de algumas aplicações ubíquas. As aplicações descritas abaixo foram desenvolvidas para utilizarem os recursos existentes em um espaço ativo gerenciado pelo Gaia. Elas são executadas na sala de testes do Gaia, que possui sistema de som *surround*, 4 telas de plasma, projetor, telas de TV digital, computadores de mão, entre outros equipamentos.

- iCalendar - agenda
- Attendance - gerencia os participantes de uma tarefa
- MP3Player - toca músicas em diversos equipamentos de som
- mPPT - mostra diversas apresentações Power Point ao mesmo tempo, de forma sincronizada
- PDFViewer - exibe arquivos Adobe PDF em uma ou mais telas
- PPTViewer - exibe apresentações Power Point nas telas ou em computadores de mão
- TickerTape - mostra informações em todas as telas da sala
- FingerPrint - autentica usuários através da impressão digital

Estão sendo estudadas novas aplicações de Computação Ubíqua que utilizem o Gaia, além do impacto nos usuários das aplicações já desenvolvidas.

2.1.2 one.world

O one.world[2] é um sistema desenvolvido pela Universidade de Washington como parte do projeto Portolano, que procura resolver três dos principais problemas de um espaço ativo [GDL+01]. Os objetos são o primeiro problema, já que eles amarram fortemente código e dados em uma mesma abstração e por isso não escalam facilmente em sistemas grandes e distribuídos. O segundo problema é a disponibilidade intermitente e limitada dos serviços em um ambiente com as características de um espaço ativo. Finalmente, o terceiro problema é a heterogeneidade de equipamentos e plataformas de software de um tal ambiente.

Para evitar o primeiro problema, no one.world dados e código são mantidos separadamente e uma nova abstração é criada para agrupar os dois, mas ainda mantendo seu isolamento. A solução do segundo problema fica a cargo do desenvolvedor das aplicações, pois os pesquisadores do one.world acreditam não ser possível esconder as falhas do sistema das aplicações. Pelo contrário, todas mudanças devem ser explicitadas. Para lidar com o terceiro tipo de problema, eles propõem que seja criada uma API comum para desenvolvimento de aplicações e um código binário comum, que permita que o mesmo programa possa ser executado em ambientes diferentes (Windows, Unix, PalmOS) e em hardwares diferentes (Intel, Sparc, Motorola) sem necessidade de recompilação do programa ou criação de uma nova versão da aplicação [GDH+01].

Arquitetura

A arquitetura do one.world pode ser exemplificada pela Figura 2.2 [GDL+01]. Cada equipamento roda uma instância do núcleo do one.world. Estes equipamentos formam nós, que são totalmente independentes uns dos outros. As aplicações que são executadas em cada nó são formadas por Componentes e armazenam dados através de Tuplas. Os Ambientes fornecem estrutura e controle, servindo de contêiner para Componentes, Tuplas e outros Ambientes.

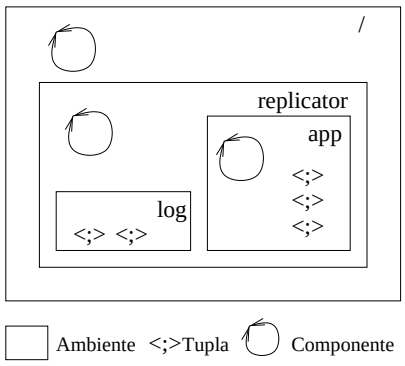

Figura 2.2: Estrutura do one.world

O acesso aos serviços, tanto locais quanto remotos, é feito através de *leases* [KJ04]. Os *leases*

[2]http://one.cs.washington.edu

limitam o tempo disponível para que as aplicações acessem os recursos e forçam as aplicações a renovarem seu interesse nos recursos solicitados. Esse mecanismo explicita as mudanças ocorridas no ambiente para as aplicações ao invés de escondê-las, que é um dos princípios do one.world.

Trabalho em andamento

Os desenvolvedores do one.world estão agora desenvolvendo aplicações que utilizam a estrutura de software criada e refinando a estrutura do one.world para torná-la mais adaptável à implementação de diversas aplicações.

A versão 0.7.1 do sistema foi lançada ao público em geral em 28/01/2002 e tanto o binário quanto o código-fonte podem ser obtidos na página do projeto na Internet.

2.1.3 Ninja

O Ninja[3] é um projeto desenvolvido pela Universidade de Berkeley na Califórnia que tem como objetivo criar um sistema capaz de oferecer serviços em rede de forma escalável e acessíveis a partir de equipamentos de capacidade de processamento reduzida [GWvB+00].

Ele aborda alguns dos problemas comuns aos espaços ativos, dentre eles como se adaptar a um ambiente com equipamentos de características diversas, como oferecer serviços escaláveis, tolerantes a falhas e acessíveis por milhares de clientes simultaneamente, como localizar um serviço no ambiente e como permitir acesso seguro a estes serviços.

Arquitetura

A arquitetura do Ninja é composta por quatro componentes: *bases, units, proxies* e *paths*.

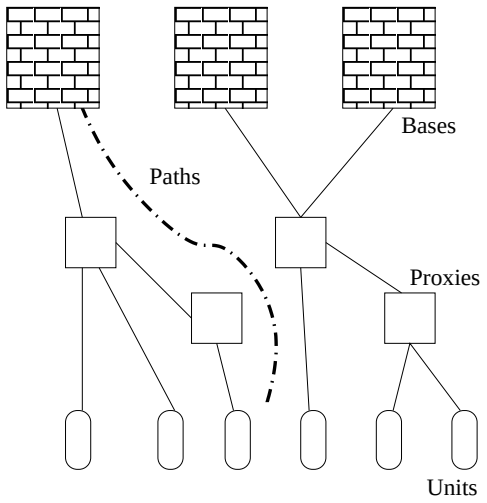

Figura 2.3: Estrutura do Ninja

[3]http://ninja.cs.berkeley.edu

- **Bases:** são aglomerados de diversas máquinas de uma arquitetura qualquer executando algum tipo de serviço que será disponibilizado às *units*.

- **Units:** são os aparelhos que utilizam os serviços do sistema. Estes aparelhos podem ser de qualquer arquitetura existente e usar qualquer tipo de protocolo, independentemente do serviço que eles queiram utilizar.

- **Proxies:** são os componentes intermediários responsáveis pela adaptação dos serviços às *units* que os utilizam. Os *proxies* podem converter tanto protocolos (como transformar uma conexão HTTP em uma conexão SSL, em casos onde seja necessária maior segurança ao acesso do serviço) quanto tipo de dados (*little endian* para *big endian* para computadores de arquiteturas diferentes, ou HTML para WML, para mostrar uma página da Web em um dispositivo com capacidade gráfica reduzida).

- **Paths:** são caminhos entre uma *unit* e uma *base*, passando por um ou mais *proxies*, que viabilizam a utilização do serviço pela *unit*.

Através desta arquitetura o Ninja consegue distribuir um serviço para máquinas de qualquer tipo, desde que se desenvolvam *proxies* para adaptar a arquitetura dessas máquinas à arquitetura do aglomerado em que o serviço está sendo executado. Ela também permite que a recuperação do sistema em caso de falhas de algum componente seja feita de forma simples, já que basta a alteração do *path* que estiver sendo utilizado para que a *unit* continue tendo acesso à *base*.

Trabalho em andamento

Depois de três anos de financiamento governamental, o projeto Ninja foi encerrado em 2001. No último ano foram realizadas pesquisas relacionadas à estrutura de dados clusterizada [GBH+00], ao gerenciamento de serviços em ambientes com alta carga e concorrência [WCB01] e à troca de mensagens entre os componentes do sistema, com a troca do Java RMI pelo NinjaRMI, uma re-implementação completa do Java RMI, com algumas melhorias em relação à implementação original.

2.1.4 Context Toolkit

O Context Toolkit[4] é um projeto desenvolvido pelo Professor Anind K. Dey na Universidade de Berkeley, Califórnia. A principal idéia do Context Toolkit é a de criar um conjunto de ferramentas que facilitem o desenvolvimento de aplicações sensíveis a contexto.

Como um dos pontos mais importantes de um espaço ativo é a coleta de informações contextuais e a utilização dessas informações pelas aplicações como forma de facilitar a interação entre o usuário e o sistema, o Context Toolkit acaba reunindo algumas características em comum com ambientes de Computação Ubíqua.

Arquitetura

A arquitetura do Context Toolkit pode ser melhor descrita através da Figura 2.4. O Toolkit é composto por três abstrações principais [DA00]:

[4] http://www.cs.berkeley.edu/~dey/context.html

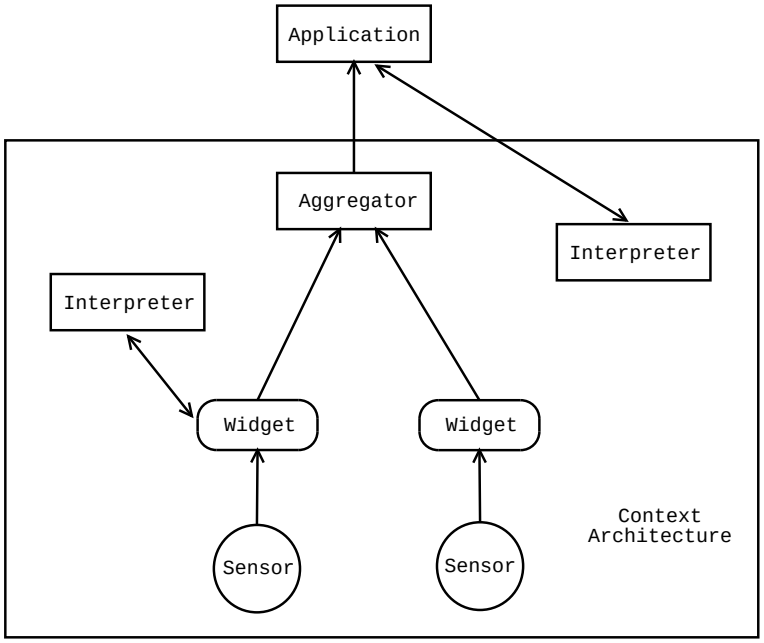

Figura 2.4: Estrutura do Context Toolkit

- **Widget:** utilizando o mesmo conceito dos *Widgets* de interfaces gráficas, foram criados os *Widgets* do Context Toolkit. Numa GUI, um *Widget* faz a mediação entre o usuário e a aplicação. No Context Toolkit, o *Widget* faz a mediação entre o usuário e o ambiente. Eles são responsáveis por encapsular informações de uma forma de contexto específica. Além disso os *Widgets* fornecem uma interface comum para as aplicações que utilizem contexto, independentemente dos sensores utilizados para captar a informação.

- **Aggregator:** os *Aggregators*, também chamados de Servidores de Contexto, são *Widgets* um pouco mais poderosos. Eles são responsáveis por coletar as informações de contexto que se referem a uma entidade do mundo real, como uma pessoa ou um lugar. O *Aggregator* recebe dados de todos os *Widgets* que forneçam informações relevantes a uma dada pessoa, por exemplo, e concentra essas informações num mesmo lugar.

- **Interpreters:** os Interpretadores de contexto são responsáveis apenas por transformar uma forma de contexto em outra. Pode ser uma conversão simples, como obter o endereço de e-mail a partir do nome de um usuário, ou mais complexa, como determinar se uma reunião está ocorrendo a partir da quantidade de pessoas numa sala, do nível de ruído e da direção em que cada uma estiver olhando.

O Context Toolkit foi escrito em Java, já que esta linguagem propicia uma certa transparência de arquitetura graças à existência de JVMs para diversas plataformas de computadores diferentes. Isso é importante nesse tipo de aplicação, que além de lidar com as diversas arquiteturas de computadores existentes num ambiente distribuído tradicional, ainda tem que lidar com os diversos tipos de sensores.

Trabalho em andamento

Depois de vários anos sendo desenvolvido pelo grupo de *Future Computing Enviroments* do Georgia Institute of Technology - GeorgiaTech, com a mudança do principal pesquisador para a Universidade de Berkeley em 2002, o Context Toolkit passou a ser desenvolvido pelo grupo de pesquisas em Computação Sensível a Contexto desta universidade.

Atualmente estão sendo pesquisados problemas relacionados à definição de contexto e à infra-estrutura necessária para aplicações sensíveis a contexto. Além disso estão sendo criadas aplicações que se utilizam do sistema já desenvolvido como forma de avaliá-lo. Uma das aplicações sendo desenvolvidas é uma cadeira de rodas sensível ao contexto, capaz de reconhecer informações como o local em que ela está sendo usada, a hora e a previsão do tempo.

2.1.5 Oxygen

O Oxygen[5] é um grande projeto sendo desenvolvido pelo Massachusetts Institute of Technology (MIT). Ele pretende investigar diversos assuntos relacionados à computação em um espaço ativo, que vão desde aspectos de interface com o usuário até redes, passando por Engenharia de Software e Sistemas Operacionais. As principais áreas de pesquisa do Oxygen estão melhor descritas abaixo:

- **Tecnologia de Equipamentos:** Pesquisa a área de equipamentos embutidos (chamados de E21) [SBGP04] como microfones, displays e sensores, utilizados para criar um espaço ativo com o qual seja possível interagir sem que o usuário se dê conta da presença de computadores, e a área de equipamentos de mão (chamados de H21) [Gut99] como *handhelds* que fornecem pontos de acesso móveis para o usuário de um espaço ativo.

- **Tecnologia de Redes:** Esta área pesquisa os esforços necessários para criar redes (chamadas de N21) [BM03, BBK02] capazes de conectar dinamicamente equipamentos móveis próximos a fim de criar grupos expontâneos de colaboração, onde trafeguem vários tipos de protocolos otimizados para diminuir o consumo de energia elétrica dos equipamentos. Estas redes também terão serviços de nome, localização e descoberta de recursos descentralizados e vai permitir que as informações sejam acessadas de forma segura.

- **Tecnologia de Software:** Pesquisa os problemas relacionados à criação de um ambiente de software para espaços ativos capaz de se adaptar facilmente a mudanças [SPP+03]. Algumas situações que ocorrem normalmente em um espaço ativo geram mudanças às quais o software deve se adaptar, como por exemplo um equipamento anônimo sendo customizado para um determinado usuário, uma alteração nas condições de operação existentes, a disponibilidade de novos softwares ou atualizações, etc.

- **Tecnologia de Percepção:** Pesquisa formas mais naturais de interação com um espaço ativo do que através de um teclado e um mouse. Dentre as tecnologias pesquisadas estão o reconhecimento de voz [Zue99] e a visão computacional [LLS+04].

- **Tecnologia de Uso:** Esta área se divide em três linhas de pesquisa que procuram descobrir formas de auxiliar o usuário a ter uma melhor experiência de utilização de um espaço ativo [BLHL01]. São pesquisadas Tecnologias de Automação, que permitem ao sistema aprender as preferências do usuário e adaptar o espaço ao seu gosto pessoal, Tecnologias de Colaboração,

[5] http://oxygen.lcs.mit.edu

que facilitem a criação de grupos de colaboração expontâneos e permitam a gravação de vídeo e áudio do resultado destes encontros e Tecnologias de Acesso ao Conhecimento, que pesquisa melhores formas de administrar e lidar com as informações existentes, personalizadas para cada usuário do ambiente.

Trabalho em andamento

O projeto Oxygen já começou a dar alguns resultados. Devido à grande abrangência da proposta original do Oxygen, os pesquisadores envolvidos resolveram encarar os problemas encontrados em espaços ativos independentemente, para depois juntar os resultados de cada área de pesquisa em um sistema operacional para Computação Ubíqua. Como resultado, hoje em dia este tal sistema ainda não está pronto, nem funcionando: existem apenas iniciativas independentes para áreas específicas. Alguns dos sistemas já desenvolvidos lidam com problemas como métodos de atualização de componentes [GWS01], mecanismos para adaptação de software e interface com o usuário. Estes sistemas entretanto ainda não se relacionam, nem foi desenvolvida uma aplicação que utilize os vários resultados de pesquisa obtidos para realizar uma tarefa específica. O resultado do projeto até agora não pode ser considerado um sistema de Computação Ubíqua, embora os esforços de pesquisa até o momento tenham dado origem a resultados de qualidade.

2.1.6 Jini

O Jini[6], desenvolvido pela Sun Microsystems, é um sistema que procura tornar mais fácil a tarefa de administrar uma rede. A tecnologia Jini provê mecanismos para que equipamentos, serviços e usuários se juntem ou saiam da rede de forma simples e natural [Sun99].

Para que a arquitetura Jini funcione, é necessário que pelo menos uma máquina do ambiente seja capaz de executar um ambiente Java completo. Equipamentos com menor poder computacional e memória podem ter acesso à tecnologia Jini através da arquitetura Surrogate[7].

Arquitetura

A arquitetura de um sistema Jini pode ser descrita através da Figura 2.5. As principais entidades desta arquitetura são:

- **Provedor de Serviços:** qualquer entidade (software, equipamento ou usuário) do sistema que deseje oferecer algum serviço aos outros membros da comunidade.
- **Cliente:** qualquer entidade (software, equipamento ou usuário) do sistema que necessite de algum serviço da comunidade.
- **Lookup Service:** serviço responsável por receber requisições de serviços dos clientes e identificar os provedores de serviço que podem satisfazer essas requisições.
- **Service Object:** objeto Java que contém a interface para uso do serviço, incluindo os métodos que usuários e aplicações executam para utilizá-lo.
- **Service Attributes:** atributos que descrevem o serviço oferecido, utilizados para identificar se o serviço satisfaz as necessidades do cliente.

[6] http://www.sun.com/jini
[7] http://surrogate.jini.org

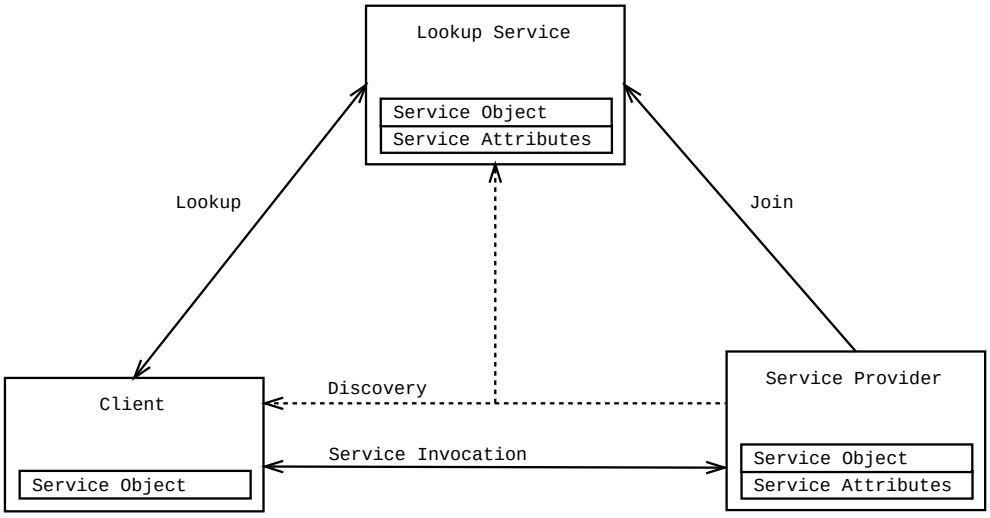

Figura 2.5: Estrutura do Jini

A tecnologia Jini funciona resumidamente da seguinte forma:

- Quando um novo provedor de serviços é adicionado à rede, ele faz um *broadcast* à procura do *Lookup Service* daquela comunidade. Este é o processo de *Discovery*.
- Depois de receber a identificação do *Lookup Service*, o provedor de serviços se registra, enviando o código do seu *Service Object* e seus *Service Attributes* ao *Lookup Service*. É a fase chamada de *Join*.
- O cliente consulta o *Lookup Service* sobre o serviço que ele necessita, passando o tipo do serviço mais alguns atributos opcionais. O *Lookup Service* identifica o serviço que se enquadra na consulta e envia o código do respectivo *Service Object* para o cliente, para que ele possa utilizá-lo. Esta é a fase de *Lookup* do Jini.
- De posse do *Service Object*, o cliente acessa diretamente o serviço através dos métodos definidos no objeto. O *Service Object* é responsável por contactar o serviço remoto utilizando o protocolo próprio do serviço em questão. Este *Service Object* é válido por um certo período de tempo, determinado por um *lease*. Passado esse período de tempo o cliente tem de renovar o *lease* para continuar tendo uma referência válida ao serviço.

Trabalho em andamento

A tecnologia Jini continua a ser desenvolvida, tanto pela Sun como pela sua extensa comunidade de usuários. Prova disso é o desenvolvimento da arquitetura Surrogate, que eliminou um dos grandes problemas da utilização do Jini em espaços ativos, que era a necessidade de máquinas com grande poder computacional, conectadas através de uma rede de bom desempenho. Para incentivar a participação da comunidade no desenvolvimento da tecnologia, a Sun decidiu liberar, em 2000, todo código-fonte produzido por ela relacionado ao Jini. Entretanto o objetivo do Jini continua sendo muito claro: resolver o problema de acesso a recursos e serviços em um ambiente distribuído.

Apesar deste ser um dos problemas que surgem na pesquisa em Computação Ubíqua, ele não é o único e o Jini sozinho não pode ser considerado um sistema de Computação Ubíqua.

2.2 Sistema de Garantias de Qualidade de Serviço – QuO

O projeto Quality Objects (QuO)[8] foi criado para superar as limitações dos sistemas CORBA tradicionais e fornecer aos objetos CORBA abstrações capazes de lidar com os aspectos de qualidade de serviço de um sistema [ZBS97]. O QuO é composto por um núcleo capaz de avaliar contratos e monitorar a qualidade de serviço dos objetos, um conjunto de linguagens chamado *Quality Description Languages* (QDL) para especificar aspectos relacionados à qualidade de serviço das aplicações e um entrelaçador (*weaver*) de códigos responsável por combinar as especificações em QDL, o código do núcleo do QuO e o código do cliente da aplicação a fim de produzir um único programa.

Em uma aplicação CORBA tradicional, o cliente executa um método de um objeto remoto através da sua interface, especificada utilizando-se a Interface Description Language (IDL) [OMG04a]. A chamada é processada por um ORB do lado do cliente, passada para o ORB do lado do objeto e então processada pelo objeto. Uma aplicação desenvolvida utilizando o QuO deve passar por alguns passos adicionais durante este processo. O QuO utiliza, além do cliente, do ORB e do objeto remoto, mais três componentes:

- Um representante local do objeto remoto (*delegate*), que possui uma interface idêntica à do objeto remoto mas é capaz de realizar uma avaliação do contrato de qualidade de serviço entre cada chamada remota e cada retorno dos métodos.
- Um contrato de QoS entre o cliente e o objeto, que especifica o nível de serviço esperado pelo cliente, o nível fornecido pelo objeto, as regiões de operação e as ações que podem ser tomadas quando o nível de qualidade de serviço é alterado.
- Objetos de condições do sistema, que fazem a mediação entre os diversos componentes, sendo usados para medir e controlar o nível de qualidade de serviço.

A Figura 2.6 mostra melhor o funcionamento de uma chamada de métodos utilizando o QuO.

Inicialmente, o cliente chama um método remoto (1) e esta chamada é passada para o *delegate*. Este componente requisita uma avaliação do contrato do cliente para verificar se é necessário realizar alguma adaptação. O contrato consulta os valores das condições do sistema e define a região de operação atual (2). Caso haja uma mudança de região, um processo de adaptação pode ser iniciado. O *delegate* então seleciona um comportamento baseado na região atual, envia a chamada ao objeto remoto e recebe o resultado (3). O contrato é reavaliado e a nova região de operação é determinada (4). Baseado nesta informação, o *delegate* seleciona o comportamento adequado e entrega o resultado ao cliente (5).

A QDL atualmente é um conjunto de três linguagens, usadas para definir diferentes aspectos relacionados à qualidade de serviço de uma aplicação. São elas:

CDL - a *Contract Description Language* é utilizada para descrever o contrato de QoS entre o cliente e o objeto, incluindo o nível de QoS que o cliente espera do objeto, o nível de QoS que o objeto espera fornecer, as regiões de QoS, as condições do sistema que devem ser monitoradas e os comportamentos que devem ser utilizados de acordo com as alterações do sistema.

[8] http://quo.bbn.com

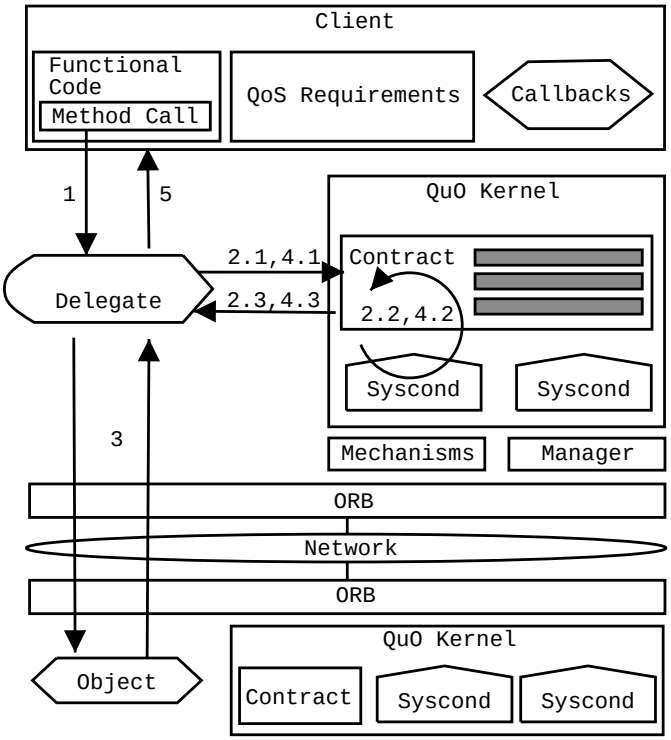

Figura 2.6: Estrutura do QuO

SDL - a *Structure Description Language* descreve a estrutura interna da implementação dos objetos remotos, tais como implementações alternativas e as opções de adaptação do *delegate* do objeto.

RDL - a *Resource Description Language* é utilizada para descrever os recursos existentes no sistema e o seu estado atual.

Destas três linguagens existentes, a mais importante para a nossa pesquisa é a CDL: ela é o componente fundamental do QuO e através dela são descritas todas as necessidades de QoS da aplicação, bem como as possíveis formas da aplicação se adaptar a mudanças. Um contrato do QuO, especificado através da CDL, é formado por:

- Regiões, que representam estados de QoS,
- Transições para cada região, que especificam a ação a ser tomada quando há mudanças de uma região para outra,
- Referências a objetos de condição do sistema, utilizados para medir e controlar a qualidade do serviço,
- Funções de *callback*, utilizadas para notificar o objeto ou o cliente da aplicação da mudança de estado de QoS.

Através das regiões, a aplicação pode negociar seu nível de qualidade de serviço com o nó e se adaptar às mudanças na disponibilidade de recursos. Entretanto, no QuO esta negociação ocorre apenas localmente. Quando a quantidade de recursos disponíveis de um nó diminui, a aplicação tem apenas a opção de reduzir o nível de QoS requerido. Na nossa arquitetura, procuramos eliminar esta limitação fazendo com que o processo de negociação inclua todo o sistema e não apenas o nó local. Desta forma, a aplicação pode também manter seu nível de QoS migrando para outros nós com mais recursos disponíveis.

Outra desvantagem é que a especificação de requisitos de QoS utilizada pelo QuO requer uma fase de compilação para geração de *stubs* tanto do lado do cliente quanto do lado da aplicação. Isto dificulta a alteração de valores e requisitos de QoS durante a execução da aplicação. Nossa arquitetura simplifica estas mudanças, utilizando um interpretador de requisitos de QoS. Com ele, a especificação pode ser alterada em tempo de execução sem interrupção da aplicação.

2.3 Sistema para Monitoração de Recursos – ReMoS

O ReMoS[9], desenvolvido pela Carnegie Mellon University, é uma biblioteca que permite às aplicações obterem informações relevantes sobre a rede, que podem ser utilizadas para que elas se adaptem às alterações detectadas [DGL+98]. O ReMoS é capaz de fornecer informações de fluxo de dados entre nós do ambiente e informações sobre a topologia das conexões da rede monitorada. Além disso, a interface do ReMoS pode devolver informações dinâmicas e estáticas sobre a rede. Uma informação dinâmica, por exemplo, seria a taxa de utilização de uma conexão, enquanto uma informação estática pode ser, por exemplo, a capacidade da conexão.

O usuário pode especificar durante qual janela de tempo as informações devem ser coletadas para depois serem passadas à aplicação. Além disso, o ReMoS também pode ser utilizado para saber qual será a disponibilidade de recursos em algum momento futuro. Para isso, são utilizadas técnicas de previsão baseadas nas informações históricas já existentes [LOG99].

O sistema possui algumas limitações, entretanto. O ReMoS não consegue lidar com pacotes *multicast* e, mais importante, a existência de rotas alternativas para um mesmo ponto é considerada apenas de forma parcial. Portanto, as informações coletadas em redes com alguma destas características podem ser distorcidas e não refletir o desempenho real das aplicações.

A API do ReMoS é dividida em três categorias:

Funções de Estado: através destas funções é possível detectar o estado corrente do fluxo de dados no caminho entre dois nós da rede. Além disso, o usuário pode utilizar estas funções para adicionar informações ao sistema. A existência de camadas adicionais de software entre a aplicação e a rede, por exemplo, podem gerar atrasos na comunicação entre dois nós. O ReMoS é incapaz de detectar este tipo de atraso. Entretanto, o programador pode informar estes valores utilizando funções existentes na API.

Funções de Encaixe: estas funções fornecem, ao programador, informações a respeito da disponibilidade de banda na rede para uma aplicação. Através destas funções, o programador pode verificar a possibilidade de alocar um novo fluxo de dados na rede e até realizar a reserva do recurso, caso o ReMoS esteja sendo executado em uma rede com suporte a QoS.

[9] http://www-2.cs.cmu.edu/ cmcl/remulac/remos.html

Funções de Topologia: as funções desta classe fornecem ao programador um grafo descrevendo a topologia entre nós escolhidos da rede. Para descrever esta topologia, as conexões são representadas como arestas do grafo, enquanto os nós podem representar dois tipos de equipamento: nós de rede representam *switches* e roteadores e nós computacionais representam os computadores.

2.4 Conclusão

Inicialmente, nosso objetivo era integrar os resultados da nossa pesquisa com o Gaia, pois entendemos que eles poderiam ser úteis no desenvolvimento de aplicações adaptáveis e poderiam ser utilizados tanto no desenvolvimento de um sistema para Computação Ubíqua, quanto no desenvolvimento de aplicações para espaços ativos.

No entanto, o Gaia ainda não possui uma distribuição estável e por este motivo o código-fonte não é liberado para o público. Pesquisamos outros diversos sistemas de Computação Ubíqua, mas nenhum deles atendia às nossas necessidades, seja pela falta de um código maduro com o qual pudéssemos integrar nosso sistema, seja pelo escopo reduzido do projeto que não justificaria os esforços de integração com nossa pesquisa.

Desta forma, continuamos a desenvolver nosso projeto paralelamente à pesquisa realizada em espaços ativos que é feita hoje em dia, com a perspectiva de integrar nossos resultados a algum destes trabalhos no futuro.

Na área de QoS, estudamos o QuO e em particular a sua linguagem para especificação de contratos de qualidade de serviço. Apesar de ser um projeto muito interessante, a sintaxe da linguagem definida para especificação de requisitos de QoS está muito ligada aos mecanismos de garantia específicos do QuO, não sendo geral o suficiente para utilizarmos em nosso projeto.

Durante este estudo, consideramos também a questão de garantia de qualidade de serviço em relação ao desempenho da rede. Problemas como conexões de rede sobrecarregadas podem ser um dos motivos que levem uma aplicação a tentar se adaptar para melhorar o tempo de resposta aos seus clientes. Por outro lado, se uma aplicação decidir migrar de servidor para melhorar seu desempenho, ela deve ter garantias de que exisitirão recursos de rede disponíveis para que os seus clientes sejam beneficiados pela migração.

Neste trabalho, restringimos nosso escopo à monitoração de valores de processamento dos nós envolvidos em nosso sistema. Não integramos o ReMoS à nossa pesquisa, nem desenvolvemos nenhum mecanismo para monitoração de informações de rede. Este, entretanto, é um item muito importante que pode ser adicionado futuramente ao sistema.

Capítulo 3

Tecnologias Utilizadas

Neste capítulo procuramos dar uma visão geral dos sistemas utilizados em nossa arquitetura. Inicialmente apresentamos os Aglets e destacamos as características que nos levaram a selecioná-lo como plataforma de desenvolvimento dos agentes móveis utilizados no sistema. Em seguida, detalhamos o funcionamento do DSRT, o sistema que utilizamos para garantia de requisitos de Qualidade de Serviço e a QML, a linguagem que utilizamos para especificação destes requisitos. Finalmente, descrevemos o funcionamento do Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos, utilizado para monitoração do uso de recursos dos nós do sistema.

3.1 Aglets - Ambiente para Desenvolvimento de Agentes Móveis

O Aglets é um sistema desenvolvido em Java para dar suporte ao desenvolvimento e à implementação de agentes móveis. Inicialmente o *Aglets Software Development Kit* (ASDK) foi desenvolvido pela IBM[1], mas hoje o ASDK é um projeto de código aberto[2].

O ASDK, além de conter todas as classes necessárias para facilitar o desenvolvimento de agentes móveis utilizando Java, também possui uma aplicação chamada Tahiti. O Tahiti permite que um usuário seja capaz de enviar, receber e gerenciar os agentes do sistema através de uma interface gráfica, o que facilita a implementação de sistemas com agentes móveis.

A escolha da linguagem Java para desenvolvimento de um sistema de agentes móveis traz benefícios e prejuízos. Por um lado, a linguagem Java oferece transparência de plataforma, execução segura de programas de terceiros, carregamento dinâmico de classes, programação *multithread*, seriação de objetos e meios que permitem a inspeção do sistema em tempo de execução (API de reflexão). Por outro lado, a linguagem Java não permite que se controle a quantidade de recursos utilizados por uma aplicação, não permite controle nem proteção às referências públicas e não possibilita a preservação implícita do estado de execução e sua posterior retomada (isto deve ser feito explicitamente pelo desenvolvedor da aplicação, quando necessário).

3.1.1 Estrutura da API de um Aglet

Segue abaixo a descrição dos principais componentes que fazem parte da API para desenvolvimento de agentes móveis [LO98]:

[1] http://www.trl.ibm.com/aglets
[2] http://aglets.sourceforge.net

Aglet: é um objeto Java móvel que visita pontos da rede habilitados para recepção de Aglets. Ele é autônomo, pois tem sua própria *thread* de execução e reativo, pois é capaz de responder a mensagens.

Proxy: o Proxy é um representante do Aglet, que impede o acesso direto de outros agentes ao Aglet, como forma de proteção. Além disso, o Proxy e o Aglet podem estar em pontos diferentes da rede, permitindo transparência de localização do agente. Nesse caso o Proxy recebe todas as requisições e as encaminha para a localização atual do Aglet.

Context: o *Context* é o ambiente de execução de um Aglet. Cada nó da rede pode ter vários *Contexts* sendo executados simultaneamente. Cada *Context* possui configurações específicas de segurança e ambiente para os Aglets que são executados sob sua supervisão.

Identifier: é a identificação do Aglet. Esta identificação é globalmente única e não se altera durante o ciclo de vida do Aglet.

Operações realizadas por um Aglet

Várias operações básicas ao funcionamento de um Aglet foram implementadas no ASDK. Um Aglet é capaz de realizar as seguintes operações:

Criação: a criação de um Aglet ocorre dentro de um *Context*. O Aglet recebe um identificador, é adicionado ao *Context* e inicializado.

Clonagem: a Clonagem produz uma cópia quase idêntica do Aglet clonado. O novo Aglet tem um novo identificador e sua execução é reiniciada. O estado de execução não é clonado.

Envio: retira o Aglet do seu *Context* de origem e adiciona o Aglet ao *Context* de destino, onde ele é reiniciado. O estado de execução não é enviado.

Retorno: quando um Aglet é solicitado para retornar ele é retirado do *Context* em que está e é adicionado ao *Context* de onde partiu o pedido de retorno.

Ativação e Desativação: a desativação interrompe temporariamente a execução de um Aglet e armazena seu estado em disco. A ativação continua a execução do Aglet no mesmo *Context* em que ele foi interrompido.

Destruição: quando um Aglet é destruído ele interrompe a sua execução e se apaga do *Context* em que está.

Comunicação entre Aglets

O ASDK fornece mecanismos para comunicação entre os agentes móveis do sistema. Existem três abstrações responsáveis por viabilizar a troca de mensagens entre Aglets:

Messages: uma *message* é um objeto passado de um Aglet para outro. Existem mecanismos para passagem de mensagens tanto de forma síncrona quanto assíncrona.

Future Reply: o objeto *future reply* é utilizado numa troca de mensagens assíncrona para permitir que o Aglet que enviou a mensagem possa receber uma reposta no futuro.

Reply Set: um *reply set* é um conjunto de *future replies*, usado para obter as respostas conforme elas apareçam, sem ter de esperar que todas fiquem prontas para recebê-las.

3.1.2 Segurança

O ASDK possui um sistema de segurança bastante completo, baseado em permissões e entidades, que possibilita que tanto o servidor que hospeda aglets quanto os próprios aglets se protejam de diversos tipos de ataques [KLO97]. Além disso a configuração é bastante flexível, permitindo vários níveis de controle de acesso. As principais permissões que podem ser configuradas são:

Permissão de Acesso a Arquivos: pode-se controlar o acesso dos Aglets a um arquivo ou diretório, permitindo-se acesso apenas para leitura ou para leitura e escrita.

Permissão de Acesso à Rede: o acesso à rede também é sujeito a controle. Pode-se controlar o acesso do Aglet a alguns servidores da rede e a certas portas de comunicação.

Permissão de Acesso ao Sistema de Janelas: pode se dar ou não permissão a que um aglet abra janelas.

Permissão de Acesso ao Contexto: pode se controlar o acesso de um aglet aos serviços oferecidos pelo *Context*, incluindo os métodos para criar, clonar, enviar, retrair, ativar e desativar aglets.

Permissão de Acesso ao Aglet: os métodos oferecidos por um aglet também podem ter seu acesso controlado. Isso impede que um aglet qualquer possa acessar os métodos de um outro aglet.

3.1.3 Exemplo de agente utilizando o ASDK

A Figura 3.1 mostra um exemplo de agente móvel utilizando as bibliotecas do ASDK. Neste exemplo é criado um agente que é enviado a uma máquina remota para capturar o conteúdo de um diretório e retorna à origem sozinho para mostrar a listagem adquirida.

Este aglet possui 4 variáveis de instância: `back`, que assume o valor `true` imediatamente antes do aglet retornar à sua origem, `dir` que contém o caminho do diretório que será listado pelo aglet, `list` que contém a listagem do diretório propriamente dita e `origin` que contém o endereço de origem do aglet.

Ele implementa o método `onCreation` que instala um `MobilityListener` para lidar com os eventos de mudança de local de um aglet. Quando o aglet chega no seu destino, a variável `back` possui o valor `false` e o aglet obtém a listagem do diretório solicitado. Quando a listagem é completada, `back` recebe o valor `true` e o aglet é mandado de volta ao seu endereço de origem. O aglet então volta para o seu ponto de partida e mostra a listagem adquirida.

3.2 DSRT - Sistema de Garantia de Tempo Real Flexível

O DSRT - *Dynamic Soft Real Time Scheduler*[3] é um projeto desenvolvido pela Universidade de Illinois em Urbana-Champaign que inicialmente procurava criar um Servidor de Processamento de tempo-real flexível capaz de garantir justiça no agendamento de tarefas e ainda assim priorizar as tarefas de tempo real agendadas sem alteração do núcleo do sistema operacional [LCN98]. Atualmente, o projeto é voltado para garantias de Qualidade de Serviço a aplicações sensíveis a variações temporais e possui além do Servidor de Processamento, um Servidor de Memória [NCN99], um

[3] http://cairo.cs.uiuc.edu/software/DSRT-2/dsrt-2.html

```
1  public class ListingAglet extends Aglet {
2      boolean back = false;
3      File dir = new File("/tmp/PUBLIC");
4      String[] list;
5      URL origin = null;
6
7      public void onCreation(Object o) {
8        addMobilityListener(
9           new MobilityAdapter() {
10            // Método chamado sempre que o agente chega a um novo nó
11            public void onArrival(MobilityEvent me) {
12              // Se o agente voltou, mostra a listagem do diretório
13              if(back) {
14                for(int i=0; i<list.length;)
15                  System.out.println(i + ": " + list[i++]);
16                dispose();
17              }
18              // Pega a listagem do diretório e manda o agente de volta
19              else {
20                try{
21                  list = dir.list();
22                  back = true;
23                  dispatch(origin);
24                }
25                catch(Exception e) {
26                  dispose();
27                }
28              }
29            }
30          }
31        );
32        // Descobre o endereço de origem e envia o agente para outro nó
33        origin = getAgletContext().getHostingURL();
34        try {
35          dispatch(new URL("atp://killi.genmagic.com"));
36        }
37        catch(Exception e) {
38          System.out.println("Failed to dispatch Aglet");
39        }
40     }
41 }
```

Figura 3.1: Código exemplo de um Aglet

Negociador de Recursos (*Broker*), possibilidade de renegociação de requisitos, entre outras características interessantes. A especificação de requisitos é feita no próprio código da aplicação, através de chamadas a funções da biblioteca do DSRT.

3.2.1 Servidor de Processamento

Através do servidor de processamento do DSRT, é possível que as aplicações tenham algum controle sobre a utilização do processador, que é um recurso compartilhado. O administrador do sistema pode especificar as necessidades de recursos da aplicação através de uma interface gráfica, ou as próprias aplicações podem fazer sua reserva de recursos utilizando as APIs fornecidas pelo DSRT.

As aplicações são classificadas em processos de *Tempo Real* e processos de *Tempo Compartilhado*, que não possuem requisitos de tempo real. Os processos de Tempo Real ainda podem ser classificados de acordo com sua *Classe de Processamento*, dependendo do padrão de utilização do processador da aplicação.

O Negociador de Processamento recebe as solicitações dos clientes e verifica se o novo processo pode ou não ser aceito, isto é, se existe capacidade de processamento suficiente disponível na máquina para acomodar a demanda de processamento do novo processo. Caso exista, o processo é transferido para a fila de processos de tempo real e é calculado um outro escalonamento dos processos existentes para que o novo processo possa ser atendido conforme sua reserva de recursos. Como o processo de negociação e escalonamento dos processos pode ser demorado, o Negociador funciona como um processo separado, de tempo compartilhado. Isto evita que a negociação de requisitos interfira no escalonamento dos processos.

Os processos de tempo real são agendados pelo escalonador do DSRT, enquanto os demais processos são agendados pelo próprio escalonador do sistema operacional. O Negociador mantém uma porcentagem do processamento livre para ser utilizada pelos processos de tempo compartilhado. Esta porcentagem pode ser ajustada pelo administrador do sistema para um melhor desempenho. A estrutura de agendamento pode ser vista na Tabela 3.1

Tipo	Prioridade	Processo
Tempo Real	mais alta	Escalonador do DSRT
	2^a mais alta	Processo de tempo real em execução
Tempo Compartilhado	qualquer um	Qualquer processo
Tempo Real	mais baixa	Processos de tempo real em espera

Tabela 3.1: Estrutura de Escalonamento

O Escalonador de Processos é executado na maior prioridade possível enquanto o processo de tempo real em execução tem a segunda maior prioridade. Os demais processos de tempo compartilhado são executados na sua prioridade normal e os outros processos de tempo real são executados na prioridade mais baixa existente. O Escalonador de processos é responsável por periodicamente mover os processos de tempo real da fila de espera (processos com prioridade de execução baixa) para a fila de execução (processo com a segunda maior prioridade). Utilizando este mecanismo, é possível priorizar processos de tempo real sem necessidade de alteração do núcleo do sistema operacional. O Escalonador de Processos é um processo leve, que não penaliza o processador e o código de escalonamento de processos é preso na memória física, para evitar a ocorrência de faltas de página que afetariam o desempenho do sistema.

A tarefa de descobrir a quantidade de processamento necessária para uma aplicação não é trivial. Este valor depende de diversas variáveis, entre elas o tipo de processador da máquina, quantidade de memória real e virtual, sistema operacional, entre outras. Para ajudar na realização desta tarefa, o DSRT fornece meios para que sejam feitos *Testes de Perfil* da aplicação. Através do teste de perfil, a aplicação pode ter uma idéia precisa da quantidade de processamento necessária para atender suas necessidades de qualidade de serviço em um computador específico. Durante a fase de teste de perfil, algumas iterações do programa são executadas sem reserva de processamento, para analisar o comportamento do processo. Depois destas iterações, o DSRT sugere um perfil de processamento a ser utilizado pelo processo e uma quantidade de processamento a ser reservada. Utilizando perfis de processamento, aplicações que não utilizam o processador em períodos constantes de tempo também podem ter suas reservas de recursos garantidas pelo sistema [CN99].

Atualmente, o DSRT aceita as seguintes classes de processamento:

Periodic Constant Processing Time (PCPT): Um processo se enquadra nesta classe se seu tempo de processamento não ultrapassa um determinado valor a cada período. Se a solicitação é aceita, o DSRT garante o tempo de processamento de pico a cada período. A aplicação precisa chamar a função `yield()` da API para sinalizar o fim do seu período de processamento.

Periodic Constrained Constant Processing Time (PCCPT): Mesmo que o anterior, mas não exige que o processo chame a função `yield()` para liberar o processador. Ideal para executar programas que não foram alterados para utilizar as APIs do DSRT.

Periodic Variable Processing Time (PVPT): Usado para processos que não ultrapassam um pico de processamento a cada período e que os excessos acima do seu tempo médio de processamento não ultrapassam um certo valor de tolerância. Neste caso o DSRT garante o tempo médio de processamento a cada período e executa os excessos que acontecerem com prioridade máxima em relação aos excessos de outros processos.

Aperiodic Constant Processing Utilization (ACPU): Um processo se encaixa nesta classe se não consome mais processamento (em porcentagem do total) que um dado valor de pico. Como não há periodicidade, o processo tem que declarar qual o seu próximo prazo para que o DSRT possa agendá-lo.

Event: Uma reserva para um evento é válida para apenas um período. A utilização de processamento da aplicação não pode ultrapassar um determinado valor de pico neste período.

3.2.2 Servidor de Memória

O desempenho de um processo depende não só da quantidade de processamento utilizada por ele, mas também da disponibilidade de memória física a ser usada pela aplicação. A falta de memória física pode implicar na ocorrência constante de faltas de página e a aplicação pode acabar gastando mais tempo realizando a paginação do que processando – fenômeno conhecido como *thrashing* [SGG04] – afetando o seu desempenho. Devido a isso, o DSRT também possui um Servidor de Memória. Através dele os processos de tempo real podem reservar a quantidade de memória física necessária antes ou durante sua execução.

O Servidor de Memória também possui um Negociador de Memória, responsável por receber as requisições de memória e, com base na quantidade disponível, aceitar ou rejeitar a solicitação. Quando um processo faz sua solicitação, o Negociador verifica se o pedido pode ser ou não aprovado.

Caso seja aprovado, o Negociador diminui a quantidade de memória disponível para reservas e separa uma área de memória compartilhada do tamanho solicitado. Um identificador da área de memória reservada é passado então para o processo solicitante que passa a utilizar esta área de memória como parte de seu espaço de endereçamento.

Não foram implementados mecanismos de teste que permitissem à aplicação identificar durante sua execução qual a quantidade de memória física necessária para reserva. O motivo para isso é que geralmente é fácil estimar a quantidade de memória a ser utilizada por uma aplicação, o que não é possível fazer em relação ao seu tempo de processamento.

O Servidor de Memória possui algumas limitações. A reserva de memória pode ser feita apenas para os segmentos de texto e dados, mas não para a pilha de execução. Implementar reserva para a pilha de execução implicaria em alterações no núcleo do sistema operacional, o que não é foco do DSRT. Outra limitação é que os usuários não podem alterar as alocações de memória feitas pelas bibliotecas de sistema, que são responsáveis por grande parte do consumo de memória das aplicações.

3.3 QML - Linguagem para Especificação de Requisitos de Qualidade de Serviço

A QML (QoS Modeling Language) [FK98], desenvolvida nos laboratórios da Hewlett-Packard, é uma linguagem criada para definir as propriedades de qualidade de serviço desejáveis de uma aplicação. Diversos sistemas para garantia de qualidade de serviço, como o QuO [LSZB98] e o DSRT [NCN99] utilizam mecanismos para especificação de requisitos em tempo de compilação. Esta estratégia não é a ideal, pois diminui a flexibilidade do desenvolvedor na definição dos requisitos de qualidade de serviço da aplicação.

Com a QML, é possível especificar requisitos de qualidade de serviço independentemente da forma como eles serão implementados. Além disso, as especificações de requisitos feitas através da QML podem ser lidas pelas aplicações em tempo de execução, fazendo-se a negociação e reserva de recursos a partir destas especificações.

3.3.1 Estrutura da Linguagem

A especificação de requisitos de qualidade de serviço em QML é feita utilizando-se três abstrações diferentes: tipo de contrato, contrato e perfil.

- **Tipo de contrato:** define as dimensões de um determinado parâmetro de QoS. O desempenho de uma aplicação, por exemplo, pode ser determinado através dos seus valores de atraso e de vazão de respostas a requisições.
- **Contrato:** especifica os valores desejáveis das dimensões definidas pelo tipo de contrato utilizado.
- **Perfil:** define os contratos que devem ser utilizados por uma aplicação particular. Pode-se definir contratos para toda a aplicação e refinamentos para cada um dos métodos de sua interface.

Além de permitir a definição de restrições simples (como `falhas < 5`), a QML também possibilita a especificação de dados mais complexos, utilizando Aspectos. Um aspecto é uma especificação

estatística, que pode ser utilizada de quatro formas diferentes: porcentagem, média, variância e freqüência. Isso permite a criação de restrições do tipo `percentile 60 < 9`, que significa que 60% dos valores de um determinado parâmetro devem ser menores que 9. Combinando aspectos e restrições simples, é possível criar especificações bastante complexas de requisitos.

A QML também possui mecanismos de herança, chamados de Refinamentos. Um contrato já existente pode ser utilizado por um novo contrato que deseje refinar a especificação de um ou mais parâmetros existentes. Contratos refinados devem ser sempre mais restritivos que os contratos originais e facilitam a definição de contratos similares em uma aplicação.

Finalmente, foi desenvolvida uma extensão da UML para a QML, que permite a especificação gráfica dos requisitos de qualidade de serviço diretamente no diagrama de classes da aplicação, durante a fase de projeto.

3.3.2 Exemplo

O código da Figura 3.2 mostra uma interface para um serviço de taxas de câmbio. O método `latest` definido na interface informa a taxa de conversão entre duas moedas, enquanto o método `analysis` realiza uma análise e devolve uma previsão de tendência para uma determinada moeda.

```
interface RateService{
    Rates latest(in Currency c1, in Currency c2)
        raises(InvalidC);
    Forecast analysis(in Currency c)
        raises(Failed);
};
```

Figura 3.2: Interface do serviço de taxas de câmbio

Na Figura 3.3 temos uma especificação de requisitos de QoS para o serviço da Figura 3.2. Nela são especificados dois tipos de contrato: `Reliability` em função do número de falhas, tempo de reparo (TTR) e disponibilidade do serviço e `Performance` em função do atraso e da vazão do serviço. O termo `increasing` significa que valores maiores são garantias mais fortes que valores menores, enquanto termo `decreasing` indica o contrário.

Também é especificado o contrato `systemReliability` que define as restrições nas dimensões do tipo de contrato `Reliability`. Este contrato pode então ser usado no perfil de uma operação, por exemplo. Finalmente, é definido o perfil de QoS da interface da Figura 3.2. Ele utiliza o contrato `systemReliability` como requisito padrão para todas as suas operações. Além disso, ele define os requisitos de desempenho de seus dois métodos separadamente, com valores diferentes. Assim, foi possível definir os requisitos de confiabilidade e desempenho de cada método do serviço de taxas de câmbio.

Em nosso projeto, as aplicações de referência que utilizam o DSRT para reservar recursos e garantir um nível de QoS, especificam seus requisitos através de QML. Isto porque durante a implantação de um sistema real, pode ser necessário ajustar os valores de alguns requisitos de QoS em função das características particulares do ambiente em que o sistema está sendo implementado. Se forem utilizadas técnicas de especificação de requisitos em tempo de compilação, o sistema teria de ser recompilado para que as alterações tivessem efeito, o que tornaria o processo de ajuste lento e trabalhoso.

```
type Reliability = contract {
  numberOfFailures: decreasing numeric no/year;
  TTR: decreasing numeric sec;
  availability: increasing numeric;
};

type Performance = contract {
  delay: decreasing numeric msec;
  throughput: increasing numeric mb/sec;
};

systemReliability = Reliability contract {
  numberOfFailures < 10 no/year;
  TTR {
        percentile 100 < 2000;
        mean < 500;
        variance < 0.3;
      };
  availability > 0.8;
};

rateServiceProfile for RateService = profile {
  require systemReliability;
  from latest require Performance contract {
    delay {
        percentile 50 < 10 msec;
        percentile 100 < 40 msec;
        mean < 15 msec;
        };
  };

  from analysis require Performance contract {
    delay < 4000 msec;
  };
};
```

Figura 3.3: Especificação de QoS do serviço de taxas de câmbio

Além disso, a definição dos requisitos de qualidade de serviço pode variar ao longo do tempo. Sistemas podem passar por períodos críticos, que exijam melhor tempo de resposta do que o normal. O QML permite que isso seja feito de forma prática e elegante.

3.3.3 QML *versus* XML

Durante a fase de pesquisa de nosso trabalho consideramos também a hipótese de utilizar a *Extensible Markup Language (XML)* para especificação de requisitos das aplicações. Depois de realizarmos alguns testes, verificamos que o formato QML é muito mais claro, legível e flexível. Como o XML não é um formato desenvolvido para esta finalidade, a especificação de requisitos em XML necessitaria de um grande número de *tags* para definição dos comportamentos que gostaríamos que fossem possíveis.

Em [JN02], Jim e Nahrstedt fazem uma classificação e uma comparação das linguagens para especificação de QoS existentes. Neste trabalho elas analisam a HQML [GNY+02], uma linguagem para especificação de qualidade de serviços baseada em XML. As autoras concluem que a QML tem vantagens em termos de reusabilidade em relação à HQML, além de ser a linguagem com melhores características em geral.

Ainda assim, se for necessário converter as especificações de QML para XML, é possível desenvolver trivialmente um programa que realize esta tarefa. O arquivo XML resultante desta conversão seria mais complexo e menos legível que o arquivo QML que o originou, mas ainda assim poderia ser utilizado pelo nosso sistema para especificar requisitos para o DSRT, desde que fosse desenvolvido o software necessário. Desta forma o DSRT poderia utilizar as especificações em XML para fazer a reserva de requisitos em favor da aplicação.

3.4 Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos

O objetivo do Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos, desenvolvido no Instituto de Matemática e Estatística da Universidade de São Paulo por Francisco José da Silva e Silva em sua pesquisa de doutorado, é criar uma estrutura de software para adaptação dinâmica, que possa ser utilizada para instrumentar aplicações distribuídas adaptáveis dinamicamente [dSeS03, SEK03, dSeSEK01].

O arcabouço introduz um modelo para adaptação dinâmica de sistemas. Este modelo é constituído por três módulos principais: o módulo de Monitoração, o módulo de Detecção de Eventos e o módulo de Reconfiguração Dinâmica.

Utilizamos o arcabouço em nossa pesquisa para monitorar os recursos dos nós do sistema distribuído. O arcabouço também é utilizado para informar as aplicações quando alguma alteração ocorreu na disponibilidade de recursos. Desta forma, as aplicações podem julgar se é necessário se adaptar à alteração e a partir desta notificação iniciar o processo de adaptação se for este o caso.

3.4.1 Monitoração

O módulo de Monitoração é responsável por acompanhar a utilização de recursos dos diversos servidores distribuídos pelo sistema. Existem dois pacotes que fazem parte deste módulo:

Monitor de Recursos: responsável por monitorar a utilização de recursos como processamento, memória, disco e rede nos diversos pontos do sistema distribuído. Atualmente este pacote está preparado para monitorar apenas a utilização de memória e processamento dos nós do sistema distribuído, mas pretende-se implementar no futuro a monitoração de outros parâmetros como utilização de disco e de recursos de rede.

Interceptador: monitora as interações entre objetos CORBA do sistema. Através desta monitoração, pode-se extrair informações importantes para a adaptação dinâmica do ambiente, como por exemplo o tempo de duração da execução de um método, ou a identidade das máquinas que mais acessam um dado objeto.

3.4.2 Detecção de Eventos

O módulo de Detecção de Eventos é responsável por definir quais situações dos parâmetros monitorados gerarão eventos para as aplicações interessadas em acompanhar a utilização de um dado recurso. Ele também é formado por dois pacotes principais:

Avaliador de Eventos: analisa os dados coletados pelo módulo de monitoração e determina a ocorrência ou não de um evento para a aplicação, baseado nas faixas de valores monitoradas pela aplicação para que seja realizada uma adaptação dinâmica.

Composição e Notificação: realiza a composição de eventos, isto é, a geração de um evento baseado na ocorrência de outros dois ou mais eventos e a notificação de fato da ocorrência do evento no sistema às aplicações interessadas.

3.4.3 Reconfiguração Dinâmica

Finalmente, o módulo de Reconfiguração Dinâmica é responsável por decidir se deve ou não ser tomada uma ação de reconfiguração e por realizar esta ação. Este módulo é constituído por três pacotes:

Tomada de Decisão: recebe os eventos gerados pelo sistema e decide as ações necessárias para adaptação aos eventos ocorridos.

Reconfigurador Dinâmico: realiza as ações de reconfiguração de acordo com as decisões tomadas pelo pacote de Tomada de Decisão.

Gerente de Dependências: auxilia o Reconfigurador Dinâmico a realizar as ações necessárias, gerenciando as dependências das aplicações envolvidas no processo de reconfiguração, tornando este processo mais consistente e seguro.

3.5 Conclusão

Como já descrevemos anteriormente, nosso sistema pretende facilitar o desenvolvimento de aplicações capazes de se adaptarem dinamicamente migrando para nós disponíveis da rede. Para que isto seja possível, três pontos são fundamentais: que o processo de adaptação possa ser realizado de forma simples e elegante pela aplicação, que os efeitos do processo de adaptação sejam duradouros e que a aplicação possua informações suficientes do sistema distribuído para tomar decisões acertadas de adaptação.

Os agentes móveis, pelas características já citadas, são uma maneira interessante de desenvolver aplicações capazes de se adaptarem facilmente às alterações dinâmicas do ambiente. Para realizarmos este trabalho, decidimos que o melhor sistema de agentes móveis a ser utilizado é o Aglets. Vários fatores pesaram para esta decisão. O mecanismo de segurança existente hoje em dia nos Aglets não é o mais completo existente, mas é suficiente para as aplicações que gostaríamos

de desenvolver. O sistema hoje é mantido por uma comunidade e licenciado de acordo com a IBM Public License, aprovada pela comunidade de código aberto. Através desta licença poderíamos ter acesso ao código fonte do sistema e alterá-lo caso fosse necessário. Outra vantagem é que o sistema está evoluindo constantemente e existem diversos recursos disponíveis, como documentação extensa e listas de discussão, capazes de nos auxiliar na utilização dos Aglets durante o desenvolvimento de nossas aplicações.

Outra vantagem dos Aglets é que eles utilizam a máquina virtual original da linguagem Java, sem alterações. Com isso o sistema possui uma maior estabilidade do que os sistemas que se utilizam de máquinas virtuais adaptadas, já que a máquina virtual original do Java é extensivamente testada e utilizada em diversas outras aplicações tanto acadêmicas quanto comerciais.

O DSRT oferece os mecanismos de reserva de recursos necessários para o protótipo que desenvolvemos. Utilizando o DSRT foi possível garantir às aplicações os recursos de que elas necessitam, tornando mais duradouros os benefícios conquistados através do processo de adaptação dinâmica. Para evitar que a especificação destes requisitos fosse fixada no código da aplicação, utilizamos um sistema para especificação de requisitos independente do código. Este sistema é baseado em uma linguagem de especificação de requisitos de alto-nível, capaz de descrever as necessidades da aplicação independentemente do sistema utilizado para implementar a reserva dos requisitos especificados. Desta forma, caso o sistema responsável por implementar as reservas de requisitos seja alterado de alguma forma, a especificação não precisa ser modificada para se adaptar a estas alterações, liberando desta tarefa o desenvolvedor da aplicação.

Entendemos que a QML é adequada para descrever os requisitos de qualidade de serviço necessários para efetuar a reserva de recursos em nosso sistema. Na verdade, a QML permite descrever uma variedade de condições que não podem ser implementadas através do DSRT. Utilizamos, portanto, um subconjunto da QML, contendo apenas as especificações de recursos que possam ser reservados pela nossa estrutura.

Para monitoração de recursos do ambiente distribuído utilizamos o Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos. Apesar deste sistema não monitorar todas as informações existentes em um ambiente distribuído, as informações disponibilizadas pelo arcabouço são suficientes para demonstrar o funcionamento do nosso sistema. Sabemos que quanto mais parâmetros possam ser monitorados pelas aplicações, melhor é o resultado das ações de reconfiguração tomadas por elas como conseqüência. Entretanto, a relação entre monitoração e desempenho também deve ser avaliada, uma vez que este processo é custoso em termos de recursos e deve ser realizado com o mínimo impacto possível para a estrutura já existente. Desta forma, concluímos que o Arcabouço fornece as informações suficientes sem interferir excessivamente no desempenho do sistema distribuído.

Capítulo 4

Negociação de Requisitos de Qualidade de Serviço (*QoS*)

Os sistemas de middleware tradicionais para aplicações distribuídas, como o CORBA [OMG04b] por exemplo, fornecem diversos serviços para facilitar o desenvolvimento deste tipo de aplicação. Serviços de eventos, de nomes e de localização, para citar alguns, são normalmente necessários a estas aplicações e são largamente utilizados por desenvolvedores de sistemas distribuídos.

Entretanto, diversas classes de aplicações altamente distribuídas possuem um requisito adicional, que normalmente não é contemplado pelos sistemas de middleware existentes. Aplicações multimídia distribuídas (como serviços de distribuição de vídeo) e aplicações de computação ubíqua necessitam também que o middleware forneça mecanismos que possam garantir a *qualidade de serviço* das aplicações. Para estes sistemas, é fundamental que a saída gerada respeite alguns limites de qualidade. Caso a saída não esteja dentro destes limites, o funcionamento da aplicação é comprometido.

Considere por exemplo o caso de um servidor de fluxos de vídeo. Os fluxos de vídeo distribuídos por este servidor devem respeitar algumas métricas de qualidade (número de quadros por segundo, atraso entre quadros, variação do atraso, etc.) a fim de que o resultado recebido pelo cliente seja um vídeo com as mesmas características do arquivo original. Caso o resultado gerado pelo servidor não tenha a qualidade desejada, o vídeo recebido pelo cliente pode ter atrasos entre o som e a imagem, quadros podem não ser mostrados ou a imagem pode congelar. Estes resultados são indesejáveis e cabe à aplicação evitar que estas situações ocorram.

Um outro caso são as aplicações executadas em ambientes de computação ubíqua. Neste tipo de ambiente, o objetivo é que o usuário não perceba a presença de computadores no ambiente e que interaja com o sistema computacional de forma natural, como outras tecnologias já difundidas em nosso dia-a-dia: interruptores, torneiras, etc. Neste tipo de aplicação, não podem haver atrasos entre a ação do usuário do ambiente e o resultado gerado pela ação. O usuário espera que, ao pressionar um botão, as luzes da sala se acendam, por exemplo. Neste momento, outras atividades do sistema não podem atrasar a resposta da ação do usuário, pois isto comprometeria a utilização do ambiente.

Para fornecer este tipo de garantia às aplicações, foram desenvolvidas desde extensões aos serviços oferecidos pelos sistemas de middleware tradicionais, como o QuO [ZBS97] - que é uma extensão do CORBA - até sistemas específicos dedicados a garantir a qualidade de serviço das aplicações, como o DSRT [NXWL01]. Entretanto, em todos estes trabalhos, o foco foi criar um

sistema que permitisse às aplicações especificar requisitos de QoS, monitorar os níveis de QoS oferecidos, obter garantias de QoS e se adaptar às mudanças nos níveis de QoS ofertados pelo sistema.

Estes serviços, todavia, não são suficientes para aplicações altamente distribuídas. Para estas aplicações, os problemas relacionados à *negociação* de requisitos de QoS passam a ter importância fundamental. Nestes sistemas, o processo de negociação pode envolver centenas de nós. As aplicações precisam que o middleware ofereça mecanismos que as auxiliem a realizar a negociação de requisitos de QoS da forma mais eficiente possível.

Para solucionar este problema, desenvolvemos um mecanismo de negociação de requisitos de qualidade de serviço baseado em agentes móveis capaz de liberar a aplicação do processo de negociação. Este mecanismo faz parte de uma solução completa de gerenciamento de QoS desenvolvida com base no DSRT. Nas próximas seções caracterizamos cada um dos componentes desta solução. Na Seção 4.1 descrevemos o mecanismo de definição de requisitos, na Seção 4.2 introduzimos o *broker* de recursos e na Seção 4.3 detalhamos o processo de negociação de requisitos de QoS em nossa arquitetura. Na Seção 4.3.1 analisamos as duas estratégias de procura desenvolvidas e finalmente na Seção 4.4 mostramos como podemos integrar o mecanismo de negociação de requisitos a aplicações preparadas para lidar com diversos níveis de QoS.

4.1 Definição de Requisitos

Uma das necessidades da nossa arquitetura, apresentadas no Capítulo 3, era que a especificação de requisitos de QoS fosse feita separadamente do código-fonte da aplicação. O objetivo era permitir que fossem feitas alterações nas especificações de requisitos de qualidade de serviço, sem a necessidade de recompilação do código da aplicação. A razão para isso é que, dependendo do cenário de utilização da aplicação, a sua necessidade de recursos pode variar. Uma aplicação que esteja em ambiente de testes tem necessidades diferentes de recursos que uma aplicação em produção. É indesejável que mudanças na especificação de requisitos da aplicação impliquem em recompilação do seu código-fonte.

Nossa arquitetura para gerenciamento de níveis de qualidade de serviço foi baseada no DSRT, apresentado na Seção 3.2. Conforme descrevemos naquela seção, o mecanismo de reserva de recursos do DSRT funciona através de uma API, que deve ser utilizada pela aplicação para testar seu perfil de QoS, iniciar uma reserva de requisitos, liberar os recursos alocados, etc. As chamadas aos métodos da API são feitas através do código-fonte da aplicação. O resultado é que mudanças nas quantidades de recursos alocados tornam necessária a alteração das chamadas da API. Isto pode exigir do desenvolvedor um esforço adicional para adaptar as chamadas da API às mudanças dos requisitos, ou até mesmo a recompilação do programa.

Para evitar este tipo de problema, nossa arquitetura separa os conceitos de *código-fonte* e *especificação de requisitos*. A aplicação define suas necessidades de requisitos em um arquivo à parte, incluindo os diversos níveis de QoS que ela pode utilizar. Este arquivo é utilizado pelo sistema de negociação para realizar as reservas de requisitos em favor da aplicação. O arquivo é interpretado em tempo de execução pelo sistema de negociação, o que permite que a especificação de requisitos seja alterada durante a execução da aplicação sem a necessidade de interrupção da execução do programa.

Utilizamos a QML, apresentada na Seção 3.3, como a linguagem de especificação de requisitos de QoS. Através da QML podem ser definidos tipos de contratos, contratos e perfis das aplicações. A Figura 4.1 mostra a definição em QML dos tipos de contratos de reserva de recursos aceitos pelo

DSRT. Para cada tipo de contrato são definidas as respectivas dimensões e as unidades utilizadas por cada dimensão.

Inicialmente, é definido o nome do tipo de contrato. Neste caso, utilizamos os mesmos nomes das classes de processamento definidas pelo DSRT. Depois disso, uma lista de dimensões é detalhada, identificando diversas propriedades das dimensões utilizadas. Primeiro, no caso de dimensões numéricas, é definido se a dimensão é *increasing* ou *decreasing*. Se a dimensão é *increasing*, significa que valores maiores são garantias mais fortes. Se ela é *decreasing*, significa que valores menores são garantias mais fortes. Depois disso, é definido o tipo do valor da dimensão, se ela é numérica, *string*, etc. Finalmente, é definida a unidade da dimensão, quando aplicável.

```
type CPU_RT_PCPT = contract {
    period: decreasing numeric msec;
    peakProcessing_per_period: increasing numeric msec;
};

type CPU_RT_PCCPT = contract {
    period: decreasing numeric msec;
    peakProcessing_per_period: increasing numeric msec;
};

type CPU_RT_PVPT = contract {
    period: decreasing numeric msec;
    peakProcessing_per_period: increasing numeric msec;
    sustainableProcessing_per_period: increasing numeric msec;
    burstTolerance: increasing numeric msec;
};

type CPU_RT_ACPU = contract {
    peakProcessingUtil: decreasing numeric percentage;
};

type CPU_RT_EVENT = contract {
    period: decreasing numeric msec;
    peakProcessing_in_period: increasing numeric msec;
};
```

Figura 4.1: Definição em QML dos tipos de contrato aceitos pelo DSRT

Os tipos de contrato mostrados na Figura 4.1 são utilizados posteriormente pelas aplicações para que elas definam suas necessidades de requisitos. Estes tipos de contrato foram definidos de acordo com os contratos aceitos pelo DSRT. Uma descrição mais detalhada de cada tipo de contrato pode ser encontrada na Seção 3.2.

A Figura 4.2 mostra um contrato e um perfil de requisitos de QoS de uma aplicação. Na definição do contrato, primeiro dá-se um nome ao contrato criado (neste caso `processamentoBaixo`), definindo-se o tipo de contrato que será utilizado (`CPU_RT_PCPT`). Depois disso são atribuídos valores a cada dimensão especificada pelo tipo de contrato.

Depois que o contrato é definido, deve ser criado um perfil, que associa uma aplicação a um ou mais contratos. Neste exemplo, está definido que o contrato `perfilBaixaCarga` da aplicação `mpeg_player` requer o contrato `processamentoBaixo`.

```
processamentoBaixo = CPU_RT_PCPT contract {
    period = 80;
    peakProcessing_per_period = 45;
};

perfilBaixaCarga for mpeg_player = profile {
    require processamentoBaixo;
};
```

Figura 4.2: Definição em QML do contrato e perfil de QoS de uma aplicação

Apesar de utilizarmos o DSRT como base da nossa arquitetura, queríamos que ela fosse flexível o suficiente para podermos utilizá-la com outros sistemas de garantia de requisitos de qualidade de serviço. Caso seja utilizado um outro sistema, do ponto de vista da especificação de requisitos de QoS, é necessário apenas definir em QML os tipos de contratos aceitos pelo novo sistema e definir os requisitos de QoS da aplicação nos termos dos novos tipos de contratos disponíveis.

Como estes arquivos são interpretados em tempo de execução pelo sistema de negociação de requisitos, desenvolvemos um interpretador de QML, capaz de identificar os tipos de contratos, contratos e perfis definidos em um arquivo de especificações QML. Para descrever a gramática utilizada e criar um interpretador da gramática, utilizamos as ferramentas JFlex[1] e BYacc/J[2]. O código do interpretador está disponível para uso e faz parte do pacote que pode ser obtido em `http://gsd.ime.usp.br/software/QoSNegotiation`.

4.1.1 Implementação

Como o DSRT não possuia uma forma de especificação de requisitos desvinculada do código-fonte da aplicação, decidimos adicionar a ele um mecanismo capaz de fornecer esta funcionalidade. Depois de analisar algumas opções de linguagens para especificação de requisitos de qualidade de serviço, decidimos utilizar a QML, pois ela era mais flexível, expressiva e simples que as demais linguagens.

Entretanto, o DSRT não permite que as especificações de requisitos de QoS sejam feitas utilizando-se a sintaxe do QML. Para resolver este problema, implementamos um *parser* da linguagem QML, capaz de transformar as definições existentes em um arquivo de requisitos de QoS nos parâmetros utilizados pelas funções da API do DSRT. O objetivo do parser era permitir que o negociador do sistema, antes de iniciar o processo de negociação, pudesse ler o arquivo com as definições de requisitos da aplicação e a partir dele construir as chamadas de funções da API do DSRT que seriam usadas durante a negociação.

Para implementar o *parser*, usamos dois programas muito utilizados para criar aplicações deste tipo: o JFlex e o BYacc/J. Através do JFlex definimos os *tokens* existentes na linguagem QML e no BYacc/J descrevemos a gramática utilizada por esta linguagem. A Figura 4.3 mostra a definição

[1] http://jflex.de
[2] http://byaccj.sourceforge.net

dos *tokens* identificados em um arquivo QML. Definimos que toda linha iniciada por '//' é um comentário, que não é considerado durante o processo de interpretação do arquivo.

```
DOUBLE = [0-9]+ "." [0-9]+
LONG = [0-9]+
NL  = \n | \r | \r\n
CHAR = [A-Za-z]
STRING = ({LONG}|{DOUBLE}|{CHAR}|\_)+
COMMENT = "//".*{NL}
```

Figura 4.3: Definição dos tokens no JFlex

A definição da gramática do QML é um pouco mais extensa, pois é necessário descrever como são formados os tipos de contrato, os contratos e os perfis de requisitos de qualidade de serviço das aplicações. Na Figura 4.4 mostramos um fragmento destas definições. Inicialmente, definimos que uma especificação de requisitos pode ser um arquivo vazio, ou pode ser composto de tipos de contrato, contratos e perfis. Depois disso mostramos como é composto um perfil: ele possui uma definição do perfil e uma lista de requisitos.

```
especification:
| especification contract_type
| especification contract
| especification profile;

profile: profile_definition "{" requirements "}" ";" ;

profile_definition: STRING FOR STRING "=" PROFILE ;

requirements: REQUIRE STRING ";" ;
```

Figura 4.4: Definição da gramática no BYacc/J

A definição do perfil é composta pelo nome do perfil (a primeira *string*), a palavra reservada `for`, o nome da aplicação (a segunda *string*), o operador `=` e a palavra reservada `profile`. Já a lista de requisitos é composta da palavra reservada `require` e o nome do contrato que é requisito da aplicação para aquele perfil.

O programa resultante é capaz de receber um arquivo de especificação de requisitos do DSRT escrito em QML e gerar a descrição da reserva de processamento. Esta descrição deve ser utilizada como o primeiro parâmetro da função `reserve(CpuReserve* cr, int pid=0)`, responsável por realizar uma reserva de requisitos junto ao DSRT.

Entendemos que este interpretador de especificações QML para o DSRT pode ser utilizado em outras pesquisas e em outros projetos. Ele foi desenvolvido de forma independente dos demais programas deste trabalho e possui uma função bem definida: permitir que aplicações definam seus requisitos de QoS para o DSRT em um arquivo específico, separado do código-fonte da aplicação. Outros projetos que tenham esta necessidade podem utilizar o *parser* para que isto seja viável.

4.2 *Broker* de Recursos

Na arquitetura que propusemos, as diversas aplicações existentes no ambiente podem negociar seus requisitos de QoS simultaneamente com vários nós do sistema. Esta situação levanta uma série de questões relacionadas à concorrência dos processos. O que fazer caso duas aplicações reservem os mesmos recursos de um nó? Como evitar que uma aplicação reserve recursos de todos os nós do sistema sem utilizá-los?

Estas questões nos levaram a identificar a necessidade de um coordenador do processo de negociação de requisitos. Este coordenador seria responsável por realizar o controle de admissão das reservas, mediar a negociação para impedir que ocorram problemas de concorrência entre reservas e evitar escassez de recursos destinados às aplicações do sistema. Inicialmente consideramos a hipótese de criarmos um processo centralizado para coordenar as requisições de reservas de requisitos das aplicações. No entanto, seria custoso manter atualizadas, de forma centralizada, as informações sobre reservas de cada nó, além de introduzir um ponto único de falha na arquitetura.

Decidimos então distribuir o processo de coordenação das reservas pelos nós do sistema. Desta forma, cada nó possui um *broker* independente, responsável por coordenar as solicitações de reserva de recursos daquele nó. Não há comunicação entre os *brokers* do sistema. Os *brokers* recebem as solicitações dos agentes de negociação e as repassam para o DSRT conforme necessário. A Figura 4.5 mostra como é realizado este processo.

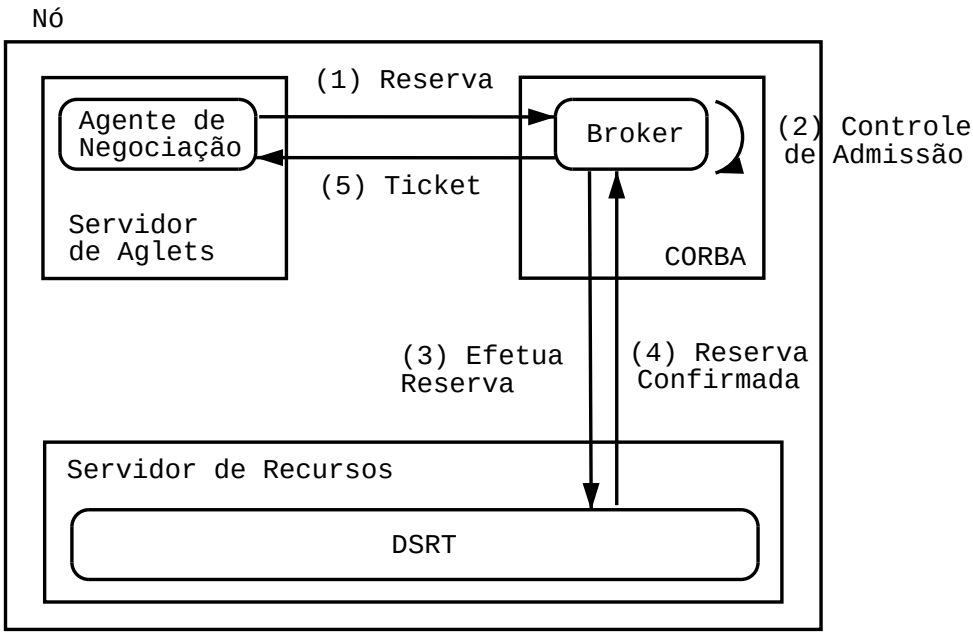

Figura 4.5: Comunicação entre o agente de negociação, *broker* de recursos e o DSRT

O agente de negociação (que será descrito mais detalhadamente na Seção 4.3) solicita ao *broker* uma reserva de recursos (1). O *broker* verifica se esta reserva pode ser atendida com os recursos disponíveis no nó; esta verificação é chamada de controle de admissão (2). Caso a reserva passe pelo controle de admissão, o *broker* efetua a reserva junto ao DSRT (3). Após a confirmação da reserva pelo DSRT (4), o *broker* gera um *ticket*, que é enviado ao agente de negociação (5). Se o agente

desejar utilizar esta reserva posteriormente, ele deve apresentar este *ticket*, que está associado a um contrato reservado junto ao DSRT. O *ticket* evita que ocorram problemas de concorrência entre solicitações de reservas simultâneas em um mesmo nó: se o agente de negociação recebe o *ticket* para um determinado contrato, os recursos solicitados estão garantidos para este contrato pelo DSRT. Outras solicitações para os mesmos recursos serão negadas pelo *broker*.

Caso a reserva não passe pelo controle de admissão, o agente de negociação pode pedir ao *broker* uma sugestão de reserva. Neste caso, o *broker* verifica qual a melhor reserva que pode ser feita com os recursos disponíveis no nó. Se o agente de negociação desejar utilizar a sugestão feita pelo *broker*, ele deve solicitar uma nova reserva, desta vez com os valores sugeridos. Por outro lado, se a aplicação estiver preparada para operar com diversos níveis de QoS, o agente pode tentar negociar um contrato de requisitos de QoS alternativo que permita à aplicação operar com menores garantias de recursos, ao invés de pedir uma sugestão de novo contrato. Desta forma, temos um processo de negociação ativo-ativo: tanto o agente de negociação pode sugerir novos contratos de reserva de requisitos durante a negociação, quanto o *broker* pode sugerir os níveis máximos de contratos de reserva de requisitos que ele pode cumprir.

Finalmente, para evitar que aplicações reservem recursos de diversos nós do sistema sem utilizá-los, impedindo que outras aplicações possam utilizar estes recursos, decidimos estabelecer um prazo de validade para as reservas realizadas. A validade das reservas é controlada utilizando-se o conceito de *leases* [KJ04]. Os *tickets* recebidos pelos agentes de negociação são, na verdade, *leases* que são válidos durante um determinado período de tempo (normalmente alguns poucos segundos). Após este período de tempo, caso o *ticket* não tenha sido utilizado pela aplicação, os recursos que lhe haviam sido reservados são liberados automaticamente. Se uma aplicação desejar utilizar os recursos de um *ticket* expirado, esta requisição é negada pelo *broker* e a aplicação tem que negociar seu contrato novamente para obter um novo *ticket*.

4.2.1 Aspectos Importantes dos *Leases*

O mecanismo *leases* é uma forma simples, eficiente e elegante de controlar o tempo de vida das referências a recursos de um sistema. Este mecanismo traz diversos benefícios, principalmente quando utilizado em sistemas distribuídos. Através da associação de *leases* aos recursos do sistema, é possível democratizar o acesso a recursos por parte das aplicações, evitar o uso desnecessário de processamento e armazenamento de referências a recursos que não são mais válidos e facilitar a adaptação do sistema a mudanças.

Entretanto, um fator crítico na implementação deste mecanismo é o tempo de duração dos *leases*. Um *lease* com validade muito curta acarreta um gasto desnecessário de processamento para sua renovação e controle. Já um *lease* com validade muito longa pode trazer os mesmos problemas de acesso a recursos que um sistema tradicional, como o acúmulo de referências inválidas e a dificuldade de acesso aos recursos.

Em nosso sistema, os *leases* foram implementados com algumas particularidades. Após a negociação de requisitos com cada nó, o contrato negociado é associado a um *lease*. Não é possível renovar o *lease*. Caso ele expire, é necessário renegociar o contrato com o nó para obter uma nova validade. A duração do *lease* é um valor definido globalmente. Com base nos testes que realizamos, definimos o valor do *lease* em 5 segundos, tempo suficiente para negociar requisitos com 8 nós utilizando a estratégia de negociação paralela ou 4 nós utilizando a estratégia linear.

É possível que em uma aplicação real de espaços ativos este valor de duração de um *lease* não seja adequado. Por exemplo, durante uma negociação, os contratos já obtidos podem perder a

validade antes de serem utilizados. Pode-se solucionar este problema de diversas formas diferentes. A princípio, pode ser feita uma sintonia manual deste valor de acordo com as características da aplicação e do espaço ativo sendo utilizado.

Uma outra solução mais geral é a implementação de um mecanismo de renovação dos *leases*. Quando estiver próximo de expirar, o *lease* notifica a aplicação, que pode revisar os valores dos contratos já negociados e avaliar a necessidade de continuar com a negociação ou utilizar um dos contratos existentes. Caso a aplicação deseje continuar com a negociação, os *leases* para os contratos já obtidos podem ser renovados por um novo período de validade. Para evitar a descaracterização do mecanismo *leases*, podemos estabelecer que cada *lease* pode ser renovado por no máximo 3 vezes.

Desta maneira, podemos definir a duração do *lease* com um valor que permita a negociação com 8 nós em condições normais, por exemplo. Os casos em que este valor não for suficiente podem ser contornados através da renovação dos *leases*. Com a renovação, a aplicação pode obter um prazo máximo de acesso ao recurso até 4 vezes maior que a duração de um *lease* típico. Isto permite que as aplicações adaptem a validade dos contratos às suas necessidades específicas, até um prazo máximo que não prejudique o funcionamento do ambiente como um todo.

4.2.2 Implementação

Para implementar o *broker* conforme a descrição da seção 4.2, tentamos inicialmente utilizar o *broker* já existente no DSRT [KN00]. Ele possui funções de controle de admissão e métodos para reserva futura de requisitos, sendo possível até agendar o início de uma reserva. Entretanto, o *broker* original do DSRT não faz distinção entre pedidos de negociação de recursos e pedidos de reservas de recursos. Quando uma aplicação faz um pedido de negociação de recursos, o *broker* original do DSRT realiza efetivamente a reserva. Desta forma, se uma aplicação decidisse não utilizar os recursos negociados em diversos nós, ela teria que contactar cada nó explicitamente e liberar os recursos alocados em cada um deles. Isto causaria um gasto desnecessário de recursos de rede, o que poderia afetar o desempenho de todo ambiente.

Devido a este fato, resolvemos desenvolver uma camada adicional de software baseada em CORBA para realizar as funções de *broker* do DSRT. Ela também realiza controle de admissão e ainda implementa o mecanismo *leasing* [KJ04] para os *tickets* de reserva gerados. Com isso, caso uma aplicação não deseje utilizar os contratos negociados com alguns nós do sistema, não é necessária nenhuma nova ação por parte da aplicação. Assim que a validade dos *tickets* expira-se, o próprio *broker* se encarrega de liberar os recursos que haviam sido destinados ao contrato do *ticket*.

A Figura 4.6 mostra como se dá a negociação de contratos de QoS entre o agente de negociação e o *broker*. Ao chegar ao nó com o qual deseja negociar requisitos de QoS, o agente de negociação obtém uma referência ao *broker* de recursos daquele nó (linha 3). Depois disso (linha 9) o agente solicita ao *broker* uma reserva de recursos conforme especificado no contrato da aplicação. Caso a reserva tenha sido obtida com sucesso, o *broker* devolve um *ticket* com valor positivo e a negociação é encerrada.

Se o *broker* verificar que o nó não possui recursos para atender ao contrato solicitado pelo agente, o agente pode pedir uma sugestão de reserva de recursos ao *broker* (linha 21). De posse do contrato sugerido, o agente de negociação pode solicitar uma nova reserva utilizando este contrato e a reserva pode ou não ser aceita pelo *broker*. Ao invés de solicitar uma sugestão de contrato do *broker*, o agente poderia também tentar realizar a reserva de recursos de um contrato alternativo.

Mesmo um contrato sugerido pelo *broker* pode não ser aceito no momento de efetuar uma reserva. Caso um segundo agente de negociação solicite uma reserva de recursos entre o momento

```
...
// Pega referência para o broker local
Broker bk = getBrokerReference();

// Pega referência para o contrato que será negociado
String[] reserva = contrato;

// Solicita a reserva de recursos para o contrato
reservationTicket = bk.reserve(reserva, agentPid.shortValue());

// Verifica se a reserva foi feita com sucesso
if(reservationTicket!=-1)
{
    // Reserva feita com sucesso!
}

// Caso não tenha conseguido a reserva, pede uma sugestão
// baseada no contrato
else
{
    String [] sug = bk.suggest(reserva);

    // Tenta usar a sugestão
    reservationTicket = bk.reserve(sug, agentPid.shortValue());

    // Verifica se a sugestão foi reservada com sucesso
    if(reservationTicket!=-1)
    {
        // Sugestão utilizada com sucesso
    }

    else
    {
        // Sugestão não pode ser utilizada
    }
}
...
```

Figura 4.6: Interação do Agente de Negociação com o *broker*

da sugestão do contrato e o da reserva efetiva dos recursos pelo primeiro agente, se esta reserva for aceita pelo *broker*, o contrato sugerido pode deixar de ser factível. Neste caso, os recursos estariam disponíveis no momento da sugestão ao primeiro agente, mas no momento da reserva eles já estariam alocados ao segundo agente. Isto implicaria em iniciar o processo de negociação

novamente a fim de obter uma nova sugestão de contrato.

Inicialmente, a sugestão de contrato era feita da seguinte forma: o *broker* tentava reduzir sequencialmente os valores de requisitos especificados pela aplicação até que uma reserva fosse aceita pelo DSRT. Nos testes que realizamos, entretanto, este método se mostrou extremamente ineficiente, impactando no processo de negociação.

Resolvemos então utilizar um outro método para encontrar uma sugestão de contrato. A partir do contrato requisitado pela aplicação, realizamos uma busca binária para encontrar o maior contrato possível aceito pelo nó. Utilizando esta estratégia, conseguimos melhorar o desempenho da negociação e a sugestão de contratos deixou de impactar no tempo total deste processo.

4.3 Processo de Negociação

Nosso principal objetivo é liberar as aplicações do processo de negociação de requisitos, oferecendo auxílio do middleware na realização desta tarefa. Para isso, desenvolvemos um sistema de negociação baseado em agentes móveis capaz de realizar a negociação com participação mínima da aplicação. Este sistema é responsável por lidar com os detalhes do processo de negociação, transmitindo para a aplicação apenas as informações que interessassem a ela.

O mecanismo de negociação é composto por duas entidades: o **Negociador**, responsável por coordenar o processo de negociação do lado da aplicação e os **Agentes de Negociação**, responsáveis por visitar os nós do sistema distribuído e realizar o processo de negociação de requisitos localmente com cada *broker*.

O negociador espera um pedido de negociação por parte da aplicação. Quando a aplicação solicita uma negociação de requisitos, o negociador lê o arquivo com a especificação de contratos de QoS da aplicação e cria os agentes de negociação necessários, de acordo com a estratégia de procura definida pela aplicação (as estratégias de procura disponíveis serão detalhadas na Seção 4.3.1). Depois de enviar os agentes de negociação aos nós, o negociador reúne as respostas das negociações enviadas pelos agentes de negociação até que o processo de negociação seja interrompido. Ele pode ser interrompido de duas formas: quando todos os agentes tiverem enviado as respostas dos seus processos de negociação ou quando a aplicação requisitar os resultados obtidos até o momento.

Os agentes de negociação por sua vez interagem localmente com o *broker* de cada nó em busca do melhor contrato de requisitos de QoS para a aplicação. Assim que a negociação é encerrada, o Agente de Negociação envia os seus resultados para o negociador e termina sua execução.

4.3.1 Estratégias de Procura

As estratégias de procura definem como os agentes de negociação percorrem o ambiente para negociar contratos de garantia de QoS. Desenvolvemos duas estratégias de procura: a estratégia linear e a estratégia paralela. A Figura 4.7 mostra o diagrama UML das estratégias de negociação desenvolvidas ao longo desta pesquisa.

Na estratégia paralela, diferentes agentes de negociação são enviados ao mesmo tempo para todos os nós do sistema distribuído incluídos no processo de negociação, realizando a negociação simultaneamente em todos eles. Já na estratégia linear, o mesmo agente percorre sequencialmente cada nó do sistema e realiza a negociação em cada nó individualmente.

A estratégia paralela obtém os resultados da negociação mais rapidamente, em troca de um maior consumo de recursos. A estratégia linear, por outro lado, consome menos recursos já que apenas um agente móvel é criado para executar a negociação, mas demora mais para obter os

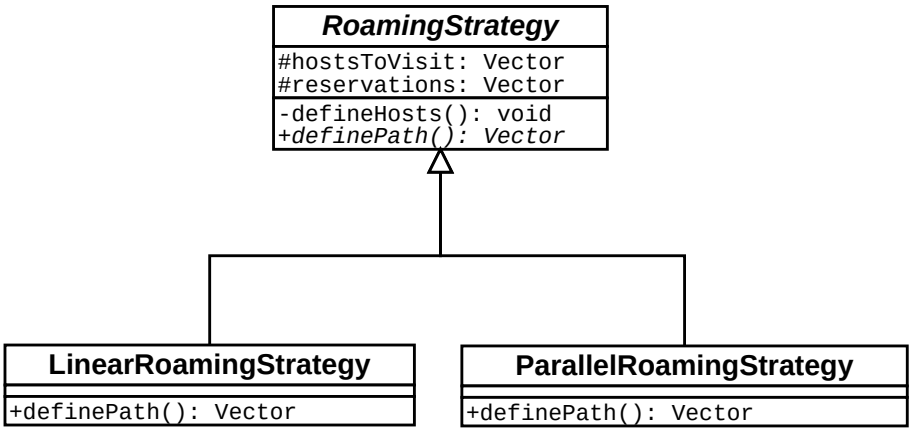

Figura 4.7: UML das estratégias de procura

resultados. Mesmo assim, a estratégia linear pode ser útil em diversos casos. Um Agente de Negociação que utilize a estratégia de negociação linear pode interromper sua busca assim que um contrato ótimo tenha sido negociado. Desta forma, o processo seria um pouco mais rápido e ainda teria a vantagem de consumir poucos recursos da rede em comparação com a estratégia paralela.

É possível imaginar uma estratégia de negociação mista, onde cada Agente de Negociação seria responsável por visitar um número limitado de nós, possivelmente sugeridos por um monitor central da rede como, por exemplo, o *Global Resource Manager* do sistema 2K [KYH+01, MK02b] ou do sistema InteGrade [GKG+04]. Isso diminuiria o consumo de recursos de rede se comparado com a estratégia paralela e traria resultados mais rápidos do que a estratégia linear. Outra estratégia que pode ser imaginada é uma estratégia *peer-to-peer*. Neste caso, o Agente de Negociação deixaria seu nó de origem sabendo apenas o seu próximo destino. Dependendo dos resultados obtidos nesta primeira negociação, o agente definiria o seu destino seguinte e assim sucessivamente.

Este mecanismo de negociação poderia ser extendido para permitir a negociação de requisitos de múltiplas aplicações. O desenho atual da arquitetura requer que cada aplicação inicie um processo de negociação separado. Se duas ou mais aplicações de um mesmo nó iniciam processos de negociação simultâneos, elas disparam múltiplos processos de negociação, o que é desnecessário. Neste caso, um mesmo Agente de Negociação pode ser responsável por negociar os contratos de todas aplicações de um mesmo nó, reduzindo a carga da rede e aumentando a eficiência do processo de negociação. Além disso, os resultados obtidos durante a negociação de uma aplicação poderiam eliminar a necessidade de renegociação das demais aplicações.

4.4 Integração com as Aplicações

As aplicações que se utilizarem do sistema de negociação oferecido pela nossa estrutura deverão possuir dois pontos de integração. As aplicações devem primeiramente iniciar o processo de negociação e depois analisar os resultados obtidos. Os detalhes do processo de negociação podem ser totalmente ignorados pela aplicação.

Toda interação entre a aplicação e o mecanismo de negociação se dá entre o código da aplicação e o agente negociador. Conforme afirmamos na Seção 4.3, o negociador é o agente responsável por coordenar o processo de negociação de requisitos da aplicação. A Figura 4.8 mostra as mensagens

trocadas entre os diversos componentes do sistema durante um processo de negociação de requisitos de QoS.

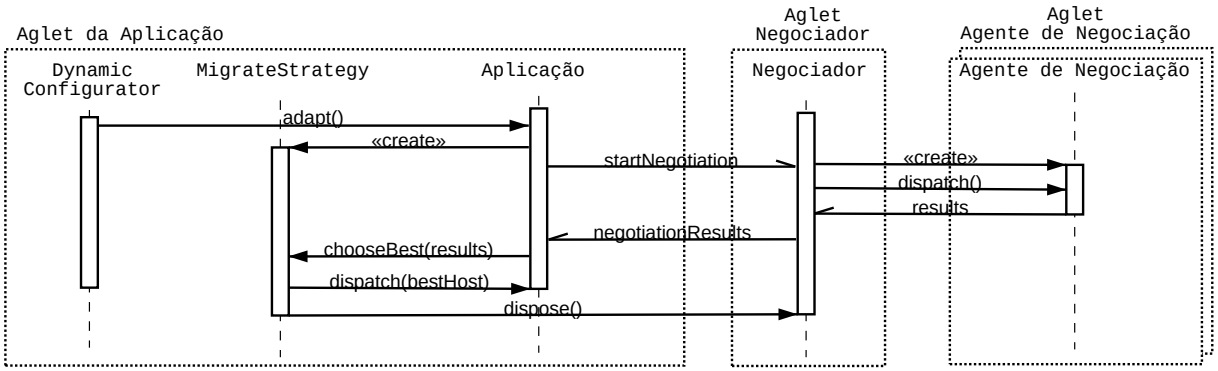

Figura 4.8: Mensagens trocadas durante uma negociação

Inicialmente, o sistema de monitoração notifica a aplicação sobre uma alteração na disponibilidade dos recursos ou em variáveis do ambiente. O processo de configuração dinâmica da aplicação avalia se é necessário que a aplicação se adapte ao ocorrido. Caso isso seja necessário, este processo envia uma mensagem à aplicação propriamente dita avisando que uma adaptação deve ser realizada, como podemos ver na Figura 4.8. A aplicação então cria um objeto com sua estratégia de adaptação - uma estratégia de migração através do objeto `MigrateStrategy` - que é responsável por receber os resultados da negociação, selecionar o melhor resultado de acordo com os requisitos funcionais da aplicação e utilizar este resultado para adaptar a aplicação de acordo com a estratégia escolhida. Depois disso, a aplicação solicita ao negociador que inicie um processo de negociação de requisitos. Ele, por sua vez, cria e envia os agentes de negociação pela rede de acordo com a estratégia de procura selecionada pela aplicação. Estes agentes de negociação realizam a negociação localmente em cada nó e enviam os resultados ao negociador, que reúne todos os resultados recebidos e os envia à aplicação. Esta por sua vez transmite os resultados à estratégia de adaptação, que analisa os resultados, seleciona a melhor oferta e, no caso descrito na Figura 4.8, migra a aplicação para o novo nó.

Se a aplicação desejasse abreviar o processo de negociação, ela poderia esperar um período de tempo e enviar ao negociador uma mensagem `getResult`. Isto faria com que o negociador reunisse as respostas obtidas até aquele momento e as enviasse à aplicação, mesmo se elas não incluíssem os resultados da negociação em todos os nós do ambiente.

Capítulo 5

Mecanismo de Adaptação

Satyanarayanan [Sat96] define três maneiras de uma aplicação se adaptar a mudanças do ambiente, conforme mostrado na Figura 5.1. Em um extremo se encontra o que foi denominado adaptação *laissez-faire*. Nesta abordagem, não é necessário nenhum auxílio do sistema, o processo de adaptação é responsabilidade total da aplicação. Esta técnica possui algumas desvantagens: não há uma entidade responsável por organizar as requisições de recursos e impor limites às aplicações. Ela também torna mais difícil o desenvolvimento de aplicações adaptáveis, uma vez que cada aplicação deve implementar todos os mecanismos necessários para que a aplicação seja adaptável dinamicamente.

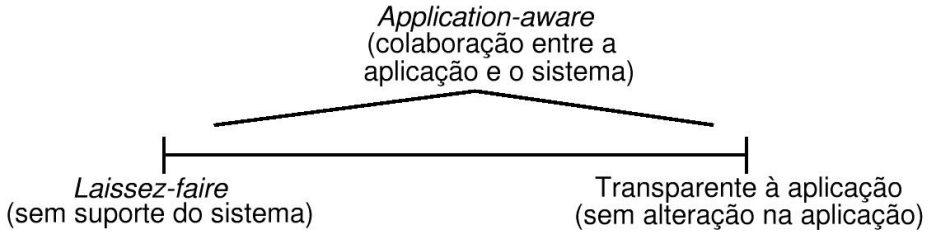

Figura 5.1: Possíveis estratégias de adaptação

No outro extremo, se encontra a abordagem de adaptação transparente à aplicação, que atribui toda a responsabilidade pela adaptação ao sistema, o que possibilita que mesmo aplicações desenvolvidas sem o conceito de adaptação sejam adaptáveis dinamicamente. A maior desvantagem desta técnica é que, em alguns casos, as decisões de adaptação tomadas pelo sistema podem ser inapropriadas ou improdutivas para a aplicação.

Finalmente, entre estes dois extremos, encontram-se diversas possiblidades que são chamadas de *application-aware*. Esta técnica se baseia em uma colaboração entre a aplicação e o sistema, o que permite que a aplicação decida a melhor estratégia de adaptação a ser utilizada enquanto possibilita que o sistema tenha controle sobre os recursos utilizados e garanta a alocação dos recursos necessários. Em nosso sistema, conforme afirmamos no Capítulo 1, alterações no ambiente que podem disparar processos de adaptação devem ser informadas às aplicações para que elas possam decidir qual a melhor estratégia de adaptação a utilizar baseada nos seus requisitos operacionais. As aplicações que se utilizam da nossa arquitetura, portanto, devem ser desenvolvidas utilizando a técnica *application-aware* de adaptação.

5.1 Monitoração

Para viabilizar a tomada de decisão de adaptação por parte da aplicação, é necessário que as condições do ambiente sejam monitoradas e que a aplicação seja notificada em caso de mudanças importantes. O Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos, descrito na Seção 3.4, é utilizado em nossa arquitetura com esta finalidade. Este arcabouço deve ser configurado com os nós existentes no sistema distribuído e com as métricas de uso de recursos que serão utilizadas na monitoração. Se, por exemplo, as aplicações do sistema devem ser notificadas caso o processamento de um nó atinja o valor de 90%, deve ser criado um evento no arcabouço retratando esta situação. Esta parametrização pode ser feita até mesmo em tempo de execução.

É possível fazer uma configuração inicial do arcabouço geral o suficiente para que qualquer aplicação seja executada no ambiente sem necessidade de reconfiguração. Por exemplo, pode-se definir que a cada 5% de alteração no uso de recursos um novo evento é gerado pelo arcabouço. Isso permite que a aplicação possa detectar um valor de uso de recursos que afete seu funcionamento. Se um novo nó é adicionado ao ambiente, ele pode se cadastrar no arcabouço e utilizar os mesmos parâmetros de monitoração.

As definições dos nós do ambiente, dos recursos de cada nó e das métricas monitoradas são armazenadas em um banco de dados no nó central de gerenciamento do sistema de monitoração. Este nó pode ser replicado para atender a requisitos de tolerância a falhas e redundância. As métricas configuradas e as condições necessárias para geração de eventos são comunicadas a cada nó monitorado, que realiza localmente a avaliação destas configurações e comunica o nó central caso ocorra algum evento. Este evento então é enviado para um canal específico, monitorado por todas as aplicações que possuem interesse neste tipo de ocorrência. O processo é melhor exemplificado através da Figura 5.2.

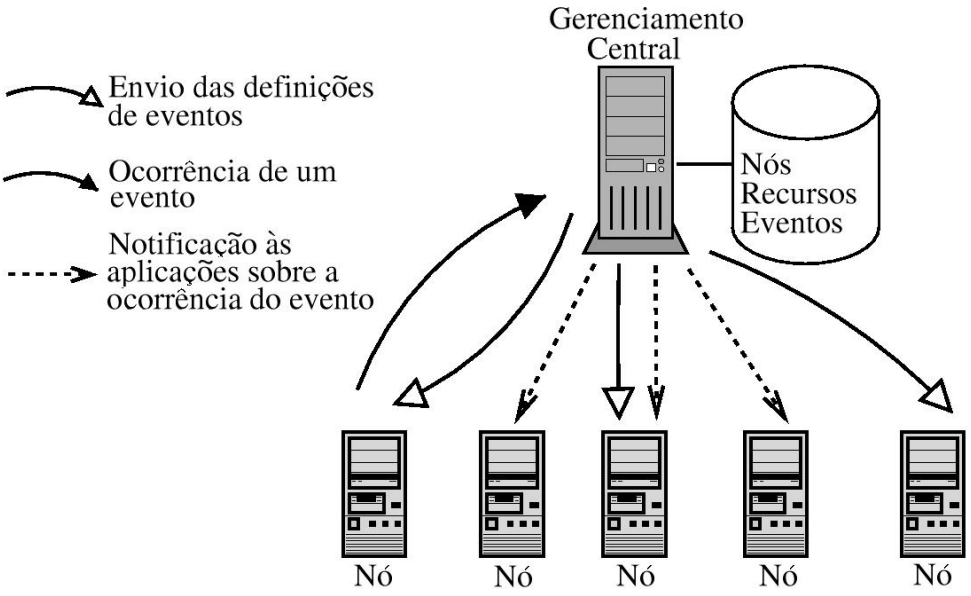

Figura 5.2: Diagrama de funcionamento do Arcabouço

O Arcabouço utiliza o mecanismo de eventos de CORBA [OMG04c] para propagar as alterações ocorridas na utilização de recursos do ambiente. Para ser notificada sobre a ocorrência de um evento,

a aplicação precisa apenas obter uma referência para o canal de eventos apropriado e aguardar até que o evento de seu interesse ocorra. O fragmento de código da Figura 5.3 mostra como isso funciona na prática.

```
1  ...
2
3  // Obtém referência para o canal de eventos apropriado
4  try {
5    NamingContextExt namingContext =
6      NamingContextExtHelper.narrow(
7        orb.resolve_initial_references("NameService"));
8    EventChannel eventChannel =
9      EventChannelHelper.narrow(namingContext.resolve(
10       namingContext.to_name("eventchannel.resource")));
11 }
12 catch (Exception e)
13 { e.printStackTrace();  }
14
15 // Obtém o objeto do canal de eventos
16 consumerAdmin  = eventChannel.for_consumers();
17 proxyPullSuplier = consumerAdmin.obtain_pull_supplier();
18
19 try
20 {
21   Any event = null;
22   EventNotification notification;
23
24   // Espera a ocorrência do evento de interesse (CPU_HIGH)
25   // e ignora os demais
26   do
27   {
28     event = proxyPullSuplier.pull();
29     notification = EventNotificationHelper.extract(event);
30   }while(!notification.eid.equalsIgnoreCase("CPU_HIGH"));
31 }
32 catch (Disconnected e)
33 { e.printStackTrace();  }
34
35 ...
```

Figura 5.3: Integração com o Arcabouço

Nas linhas de 5 a 10 a aplicação obtém referências para os serviços CORBA utilizados pelo Arcabouço: o serviço de nomes e o serviço de eventos. Nas linhas 16 e 17 a aplicação consegue as referências para o canal de eventos de recursos, utilizado pelo Arcabouço para anunciar os eventos

relacionados aos recursos do sistema. Finalmente, nas linhas de 26 a 31 a aplicação aguarda a ocorrência de algum evento, até que o evento de interesse da aplicação ocorra. Neste caso, a aplicação está esperando por um evento que indique alto consumo de processamento (*CPU_HIGH*).

Desta forma, a aplicação pode ser notificada sobre mudanças na utilização de recursos do ambiente que podem afetar diretamente a qualidade do serviço prestado pela aplicação. A notificação do evento fornece um gancho, através do qual a aplicação pode disparar o seu processo de adaptação.

5.2 Adaptação

Neste trabalho, focalizamos duas técnicas de adaptação: a clonagem e a migração da aplicação. O desenvolvimento de aplicações utilizando o paradigma de agentes móveis tornam naturais a utilização destas duas técnicas, uma vez que agentes móveis possuem mecanismos nativos que facilitam a transferência de código entre nós do sistema.

Logicamente, a utilização de clonagem e migração de código como forma de adaptação de aplicações traz consigo o problema de atualizar as referências utilizadas pelos clientes. Se a aplicação passa a ser executada em um novo nó, ou se uma nova instância da aplicação começa a ser executada, é necessário que os clientes da aplicação sejam avisados desta mudança. Nesta pesquisa não abordamos este problema. Entendemos que existem diversas técnicas disponíveis para que as aplicações solucionem esta situação e cada aplicação pode optar por uma forma específica de lidar com esta questão. Em nossos experimentos com o refletor de áudio, por exemplo, criamos uma lista de execução (*playlist*) contendo todos os nós do sistema, que era tocada repetidamente. Quando um dos nós deixava de transmitir o fluxo de áudio, a aplicação cliente automaticamente tentava se conectar ao próximo nó da lista e assim sucessivamente, até encontrar um novo nó transmitindo o fluxo de dados.

Através da técnica de migração, as aplicações podem mudar de nó e escolherem um outro nó do sistema que possua mais recursos disponíveis do que o nó onde estão sendo executadas. A técnica de clonagem, por outro lado, permite que as aplicações se adaptem de forma elegante a picos de utilização, criando um clone a fim de dar vazão ao aumento de requisições por parte dos clientes. Em nosso sistema, criamos uma classe abstrata chamada *AdaptationStrategy*, implementada através de duas classes: *MigratingStrategy* e *CloningStrategy*. O diagrama UML da relação entre estas classes está na Figura 5.4.

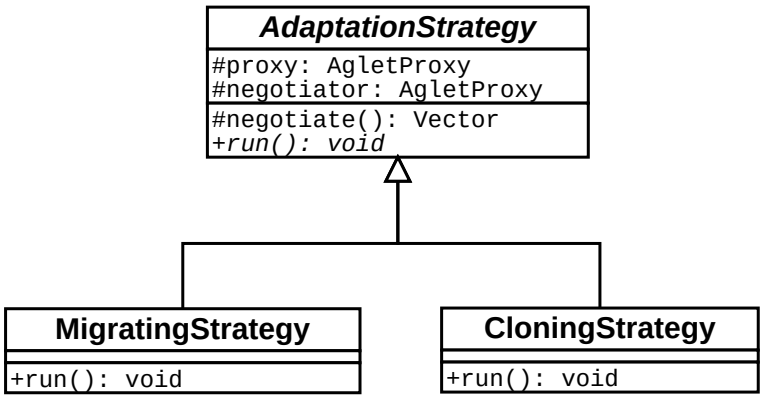

Figura 5.4: Diagrama UML das estratégias de adaptação

A classe *AdaptationStrategy* define todos os métodos comuns a qualquer estratégia de adaptação. Ela implementa métodos para definição da melhor oferta obtida pela aplicação durante a negociação e para início da adaptação. Cada estratégia derivada da classe *AdaptationStrategy* deve implementar o seu próprio método *run()*. O código da Figura 5.5 mostra a implementação do método *run()* da estratégia de migração.

```
1  public void run()
2  {
3    URL address=null;
4    NegotiationResult winner=null;
5
6    // Calcula a melhor oferta recebida
7    winner = bestOffer(offers);
8
9    // Se existe uma oferta melhor, envia a aplicação para o novo nó
10   if(winner!=null)
11   {
12     address = winner.getHost();
13     try
14     {
15       proxy.dispatch(address);
16     }
17
18     catch (IOException e)
19     {
20       e.printStackTrace();
21     }
22     catch (AgletException e)
23     {
24       e.printStackTrace();
25     }
26   }
27   else
28     System.out.println("Negotiations failed, impossible to migrate");
29 }
```

Figura 5.5: Implementação da estratégia de migração

Na linha 7 a estratégia define a melhor oferta de garantia de QoS dentre as recebidas pela aplicação durante o processo de negociação. Caso seja encontrada uma oferta, a estratégia obtém o endereço do nó que fez a oferta na linha 12 e na linha 15 ela envia a aplicação para o novo nó. A estratégia de clonagem tem implementação similar, mas utiliza o método *clone(address)*, que é fornecido pela classe *Aglet*, ao invés do *dispatch(address)*. Outras estratégias de adaptação poderiam ser criadas, de acordo com a necessidade específica de cada aplicação, herdando-se os métodos necessários da classe *AdaptationStrategy*.

5.3 Integração com as Aplicações

A integração da aplicação com o processo de adaptação é muito fácil de ser feita. O código da Figura 5.6 mostra como isso funciona. Este fragmento foi retirado do trecho do código da aplicação que recebe as mensagens enviadas pelo processo de negociação de contratos de QoS. Quando a aplicação recebe uma mensagem com os resultados da negociação, ela retira desta mensagem um vetor com os contratos negociados em cada nó visitado durante o processo de negociação. Os contratos negociados são então passados para a estratégia de adaptação, que definirá o melhor contrato recebido e executará a ação de adaptação, que pode ser migrar a aplicação para um novo nó ou criar uma cópia da aplicação em um novo nó, dependendo da estratégia de adaptação que foi utilizada.

```
1  ...
2
3  if(msg.sameKind("negotiationResults"))
4  {
5    // Recebe os resultados da negociação
6    Vector results = (Vector)msg.getArg();
7
8    // Inicia o processo de adaptação com os resultados recebidos
9    adapt.doAdaptation(results);
10
11   return(true);
12 }
13
14 ...
```

Figura 5.6: Integração da aplicação com a estratégia de migração

Não é necessário que a aplicação esteja encapsulada em um agente móvel para se beneficiar dos mecanismos de adaptação fornecidos pela nossa arquitetura. Entretanto, a utilização do paradigma de agentes móveis permite que as aplicações disponham de formas mais ricas de adaptação, facilitando a implementação de estratégias de migração e clonagem. Em nossos testes, utilizamos uma aplicação encapsulada em um agente móvel pois entendemos que desta forma poderíamos enriquecer a estratégia de adaptação da aplicação.

Neste capítulo procuramos mostrar os mecanismos oferecidos pela nossa arquitetura para facilitar o desenvolvimento de aplicações adaptáveis dinamicamente. Mostramos também como as aplicações podem se beneficiar destes mecanismos para implementarem diferentes estratégias de adaptação que permitam que elas mantenham um nível satisfatório de qualidade de serviço para os seus clientes. Nossa arquitetura fornece mecanismos flexíveis, que podem ser utilizados pelas aplicações de acordo com suas necessidades particulares. Através dos fragmentos de códigos apresentados, procuramos também demonstrar como é simples integrar aplicações à nossa arquitetura proposta.

Aplicações multimídia distribuídas podem se beneficiar de nossa arquitetura de diversas formas. Uma aplicação que tenha como objetivo distribuir vídeos para milhares de clientes, por exemplo,

pode monitorar o estado dos recursos do nó onde ela se encontra a fim de identificar situações que afetem a qualidade do vídeo transmitido aos seus clientes. Caso a aplicação sofra perda de desempenho relacionado a algum problema com o servidor que a hospeda, ela pode procurar outro nó do sistema para manter a qualidade do vídeo enviado aos clientes. Caso haja um pico de demanda pelo vídeo transmitido, a aplicação pode procurar um nó que possua mais recursos disponíveis e que desta forma possa fornecer um melhor contrato de QoS para atender à variação na demanda. A aplicação pode até criar uma cópia dela mesma, hospedada em outro nó, para dividir a carga conseqüente do aumento de demanda.

Aplicações de computação ubíqua também podem se aproveitar da arquitetura de várias formas. Quando um usuário entrar em uma sala preparada com diversos dispositivos computacionais, uma aplicação que está sendo executada em seu computador de mão pode migrar para um outro nó da sala a fim de melhorar a qualidade da saída ao usuário. Além disso, ambientes de computação ubíqua podem ser usados ora por apenas uma pessoa, ora por várias, como por exemplo em uma reunião. As aplicações deste tipo de ambiente devem se adaptar às variações de demanda pelos seus serviços. Uma das formas possíveis de adaptação é a criação de vários clones da aplicação durante a reunião, que podem deixar de ser executados tão logo a reunião termine.

Capítulo 6

Aplicações

Procuramos testar nossa arquitetura com aplicações que necessitassem de garantias de qualidade de serviço e que, com isso, pudessem obter melhores resultados através da nossa arquitetura. A área de multimídia é rica em aplicações com este perfil. Este tipo de aplicação lida geralmente com imagens e sons. Falhas no processamento são imediatamente percebidas pela nossa visão ou audição. Estas aplicações, portanto, possuem necessidades estritas de recursos para evitar a ocorrência de falhas. Nossas duas aplicações são originárias desta área e serão descritas nas seções a seguir.

6.1 Refletor de Áudio/Vídeo

A transmissão de conteúdo multimídia na Internet normalmente é um processo caro em termos de rede e de processamento. Os servidores multimídia devem capturar som e imagem, converter o conteúdo para um formato apropriado de transmissão utilizando algoritmos de compressão e enviar aos clientes uma grande quantidade de dados. À medida que o número de clientes deste tipo de aplicação aumenta, a estratégia cliente/servidor se torna ineficiente.

A arquitetura típica escalável para serviços de envio de fluxos multimídia em tempo real na Internet utiliza refletores multimídia para diminuir a carga nos servidores do sistema gerada pela transmissão dos dados [KCN01]. Esta arquitetura está retratada na Figura 6.1.

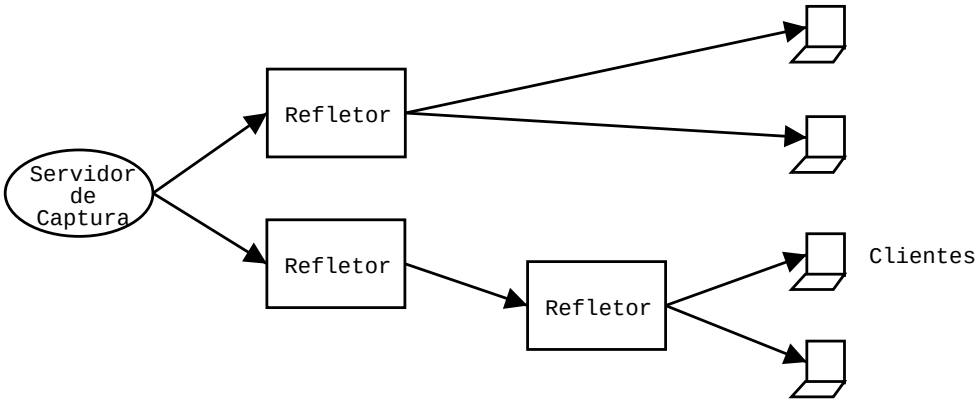

Figura 6.1: Arquitetura típica de envio de conteúdo multimídia pela Internet

Os clientes da aplicação se conectam aos refletores ao invés de se conectarem diretamente aos servidores de conteúdo multimídia. Estes refletores são processos leves em termos de processamento, uma vez que eles apenas se conectam ao servidor e repassam os dados recebidos aos clientes da aplicação ou a outros refletores. Esta arquitetura alivia a carga nos servidores de conteúdo multimídia e melhora o desempenho do sistema como um todo.

Para validar o funcionamento da nossa arquitetura, nós desenvolvemos um refletor multimídia baseado no paradigma de agentes móveis. Este refletor, chamado `ReflectorAglet`, é capaz de especificar seus requisitos de QoS, detectar indisponibilidade de processamento e de usar o sistema de negociação de requisitos de QoS para encontrar um nó do sistema capaz de suprir suas necessidades de recursos.

Os refletores monitoram o processamento do seu nó para dectarem possíveis faltas de recursos. Caso o refletor detecte uma sobrecarga no seu nó, ele inicia um processo de negociação no ambiente a fim de encontrar outro nó capaz de fornecer os requisitos de QoS necessários. O refletor então migra para este novo nó, onde passa a utilizar o novo contrato de QoS negociado.

6.2 Implementação do Refletor de Áudio/Vídeo

O arquivo de especificação de requisitos do refletor foi colocado em um servidor Web, para que o sistema de negociação pudesse ter fácil acesso de leitura. A especificação de requisitos do nosso refletor está descrita na Figura 6.2.

```
type CPU_RT_PCPT = contract {
    period: decreasing numeric msec;
    peakProcessing_per_period: increasing numeric msec;
};

processamentoIdeal = CPU_RT_PCPT contract {
    period = 40;
    peakProcessing_per_period = 30;
};

processamentoMinimo = CPU_RT_PCPT contract {
    period = 100;
    peakProcessing_per_period = 60;
};

idealProfile for reflectorAglet = profile {
    require processamentoIdeal;
};

minimoProfile for reflectorAglet = profile {
    require processamentoMinimo;
};
```

Figura 6.2: Especificação de requisitos do Refletor Multimídia

Ela especifica dois tipos de contrato de requisitos de QoS para o refletor multimídia: um contrato ideal, que demanda 30 ms de processamento a cada período de 40 ms e um contrato mínimo, que demanda 60 ms de processamento a cada período de 100 ms. A aplicação é responsável por decidir qual contrato atende suas necessidades de recursos em um dado instante. O refletor, por exemplo, poderia começar sua execução com o contrato mínimo e, conforme a quantidade de clientes da aplicação aumentasse, renegociar o seu contrato para passar a utilizar o contrato ideal.

O refletor é constituído por uma *thread* principal e diversas outras *threads* responsáveis por funções específicas do programa. Uma *thread* é responsável por se conectar ao servidor multimídia, enquanto uma nova *thread* é criada para lidar com cada conexão dos clientes. Além destas, uma outra *thread* é utilizada para receber notificações sobre escassez de recursos do ambiente. A *thread* principal tem a função de coordenar a execução das demais *threads*, além de tomar decisões que afetem globalmente o refletor, como por exemplo qual contrato de requisitos de QoS deve ser utilizado, quando deve ser iniciado um processo de negociação, como a aplicação deve se adaptar a alterações na disponibilidade de recursos, entre outras.

Na versão atual da nossa arquitetura, cada aplicação deve criar um agente negociador para coordenar o seu processo de negociação de requisitos. Futuramente, isto pode ser refinado de forma que seja criado apenas um agente de negociação em cada nó do sistema, independentemente do número de aplicações sendo executadas naquele nó. Devido a esta limitação da nossa arquitetura, quando o refletor é iniciado, ele deve também criar um agente negociador. A Figura 6.3 mostra o código executado pelo refletor no momento da sua criação.

Retiramos do código apenas o trecho onde é criado o *MobilityListener* e são definidos os procedimentos para cada evento de movimentação de interesse do refletor multimídia. Quando o refletor chega a um novo nó, por exemplo, devem ser criadas novamente as *threads* de reconfiguração dinâmica, negociação e de tratamento das conexões com os clientes. O código executado pelo agente ao chegar em um novo nó está na Figura 6.4.

A *thread* de reconfiguração dinâmica se conecta ao canal de eventos do Arcabouço para Adaptação Dinâmica de Sistemas Distribuídos, apresentado na Seção 3.4. Por este canal são enviados todos os eventos de utilização de recursos cadastrados no Arcabouço e identificados pelo sistema. Outras aplicações podem implementar neste trecho do código uma lógica mais complexa para verificar se é preciso que a aplicação se adapte, de acordo com os eventos recebidos. No caso da nossa aplicação, a *thread* de reconfiguração dinâmica apenas verifica se é um evento que indica alto consumo de processamento e notifica a aplicação que ela deve ser reconfigurada. O método *run* da *thread* de reconfiguração dinâmica utiliza o método *checkEvent* (apresentado na Figura 6.5) para receber eventos do arcabouço e identificar eventos de alto consumo de processamento. Após a identificação do evento, o método *run* chama outro método, definido pela aplicação, com as ações de reconfiguração necessárias.

Este método utiliza um objeto da classe *AdaptationStrategy*, que possui a lógica de definição da melhor oferta dentre os resultados obtidos pelo processo de negociação e do método de adaptação que será utilizado pela aplicação (se ela irá migrar para um novo nó ou se irá criar um clone e enviá-lo a outro nó, por exemplo).

Tanto a lógica para definição da melhor oferta recebida quanto o melhor método de adaptação a ser utilizado são decisões específicas de cada aplicação. A estratégia de adaptação deve então ser criada pelo desenvolvedor da aplicação. No caso do refletor, criamos as estratégias de migração e de clonagem. Ambas podem ser úteis para a aplicação, dependendo da situação a que ela for submetida: a migração pode ser utilizada para que o refletor fique mais próximo dos clientes, diminuindo a latência de rede na transmissão dos dados, enquanto a clonagem pode servir para

```
1  public void onCreation(Object init)
2  {
3    ...
4
5    // Cria um listener para ser notificado de
6    // eventos de movimentação do agente
7    addMobilityListener(mobListen);
8
9    // Abre uma janela para que o usuário digite o endereço do
10   // servidor multimídia e a porta em que o refletor responderá
11   dialog = new ReflectorAgletDialog();
12   dialog.show();
13   dialog.waitResults();
14
15   // Cria a thread que aguarda conexões
16   this.serverAddress=dialog.getServerAddress();
17   this.localPort=dialog.getLocalPort();
18   createServer(serverAddress, localPort);
19
20   // Cria a thread de reconfiguração dinâmica
21   createDynConfiguration();
22
23   // Cria o aglet para negociação de QoS
24   createNegotiator();
25 }
```

Figura 6.3: Criação do refletor multimídia

```
1  public void onArrival(MobilityEvent event)
2  {
3    createServer(serverAddress, localPort);
4    createDynConfiguration();
5    createNegotiator();
6  }
```

Figura 6.4: Chegada do refletor multimídia a um novo nó

dividir a carga de clientes entre o refletor e o seu clone. O trecho de código do refletor que inicia o processo de adaptação é apresentado na Figura 6.6.

A resposta do processo de negociação chega ao refletor utilizando o mecanismo de troca de mensagens dos aglets, apresentado na Seção 3.1. O refletor então transmite este resultado ao objeto de estratégia de adaptação, que deve escolher a melhor oferta e realizar a adaptação da aplicação. No caso do nosso refletor, a estratégia de adaptação escolhida foi a migração da aplicação para

```
1  private void checkEvent(ProxyPullSupplier pps)
2  {
3    Any ev = null;
4    EventNotification event;
5
6    try
7    {
8      do
9      {
10       // Recebe o evento
11       ev = pps.pull();
12       // Extrai as informações do evento
13       event = EventNotificationHelper.extract(ev);
14     }while(!event.eid.equalsIgnoreCase("CPU_HIGH"));
15   }
16   catch (Disconnected e)
17   {
18     e.printStackTrace();
19   }
20 }
```

Figura 6.5: Verificação dos eventos detectados pelo Arcabouço

```
1  public void adaptationStrategy()
2  {
3    // Cria o objeto de estratégia de adaptação que utiliza migração
4    AgletProxy reflectorProxy =
5        getAgletContext().getAgletProxy(getAgletID());
6    adapt = new MigratingStrategy(reflectorProxy);
7
8    // Inicia uma negociação de requisitos
9    try
10   {
11     negotiator.sendMessage(new Message("startNegotiation"));
12   }
13   catch (Exception e)
14   {
15     e.printStackTrace();
16   }
17 }
```

Figura 6.6: Início do processo de adaptação

um novo nó. A estratégia de adaptação utiliza o *proxy* do aglet da aplicação para realizar a transferência, como podemos ver na Figura 6.7.

```
1  public void run()
2  {
3    URL address=null;
4    NegotiationResult winner=null;
5
6    // Seleciona o melhor contrato
7    winner = bestOffer(offers);
8
9    if(winner!=null)
10   {
11     address = winner.getHost();
12
13     try
14     {
15       // Envia o refletor para o nó com a melhor oferta
16       proxy.dispatch(address);
17     }
18     catch (Exception e)
19     {
20       e.printStackTrace();
21     }
22   }
23 }
```

Figura 6.7: Estratégia de adaptação - Migração

6.3 Multiplexador de Áudio

Nossa principal preocupação ao desenvolver uma aplicação para a nossa arquitetura era que a aplicação pudesse ser simples o suficiente para que mantivéssemos o foco da nossa pesquisa no desenvolvimento da arquitetura, e não no desenvolvimento da aplicação. Ao mesmo tempo, procurávamos uma aplicação que pudesse se beneficiar dos serviços oferecidos pela nossa arquitetura, ou seja, que necessitasse de garantia de qualidade de serviço para desempenhar bem o seu objetivo.

O refletor multimídia, apesar de ser uma aplicação simples de ser desenvolvida e de requerer garantias de QoS, é um processo muito leve de ser executado. Normalmente, para um número pequeno de clientes, o consumo de processamento do nó não é preocupante. É claro que, à medida que o número de clientes aumenta, a necessidade de processamento do refletor também aumenta, tornando-se um recurso crítico para a aplicação. Mas isso acontece apenas quando o refletor está sujeito a uma demanda muito grande de clientes [KCN01], o que é difícil até de simular em laboratório.

Uma outra aplicação multimídia que poderia se beneficiar dos serviços oferecidos pela nossa arquitetura é um multiplexador (*mixer*) de áudio. Um multiplexador é um componente capaz de reunir diversos sinais de áudio distintos em um único sinal. O multiplexador possui diversas aplicações práticas e pode ser usado, por exemplo, para otimizar o tráfego de rede necessário para realizar conferências multimídia com vários participantes. A Figura 6.8 mostra os fluxos de dados necessários em uma conferência que utilize apenas refletores de áudio.

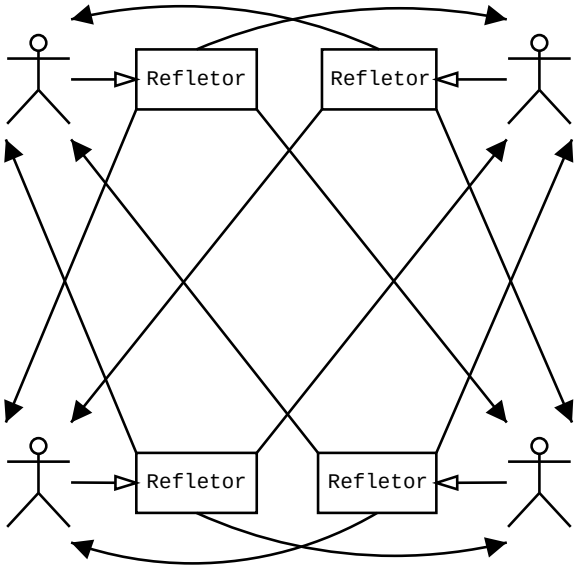

Figura 6.8: Conferência utilizando apenas refletores

Como podemos ver na figura, utilizando refletores, cada participante da conferência precisa enviar apenas um fluxo de dados, mas ainda assim precisa receber $n - 1$ fluxos, onde n é o número de participantes da conferência. Esta arquitetura não é eficiente, já que os fluxos recebidos por um participante podem interferir no fluxo de dados enviado por ele, gerando atrasos e degradando o desempenho da solução globalmente. Além disso, ela também não é escalável, pois quanto mais participantes tiver a conferência, mais fluxos serão recebidos por cada participante, impactando ainda mais no desempenho da aplicação.

A Figura 6.9 mostra como a utilização de um multiplexador pode otimizar o consumo de rede e o desempenho da aplicação. Nesta arquitetura, cada nó envia um fluxo de dados e recebe outro, independente da quantidade de participantes. Cabe ao multiplexador combinar os fluxos de dados gerados por cada participante em um único sinal que possua o som de cada um deles. Este fluxo mixado é então transmitido a todos os clientes. Esta arquitetura é mais escalável que a anterior, já que a quantidade de dados recebida por cada nó não depende mais do número de participantes da conferência.

A operação de mixagem dos sinais é relativamente simples. Basta somar os sinais de áudios recebidos de cada participante e dividir o resultado pelo número de participantes. Mesmo assim, o multiplexador é uma aplicação que faz uso intensivo de processamento. Os fluxos de dados originais são normalmente enviados em formato codificado (como por exemplo o MP3), que comprimem as informações e diminuem o tráfego de rede necessário para envio do dado. O multiplexador deve então descomprimir o fluxo de áudio enviado por cada participante para obter o áudio em formato

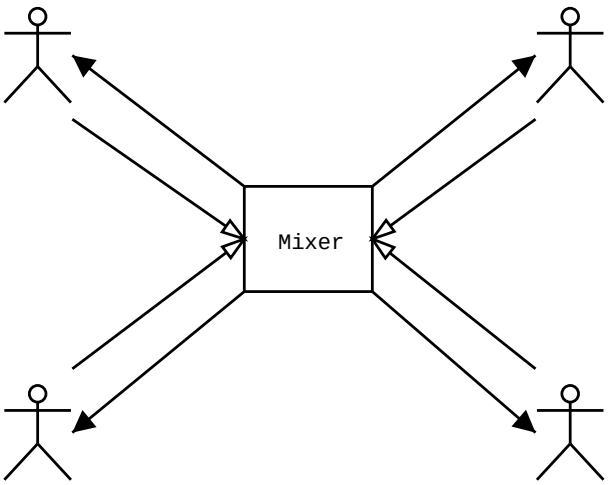

Figura 6.9: Conferência utilizando o mixer de áudio

puro (*raw*), mixar o sinal puro e codificar o sinal resultante novamente, para envio aos clientes.

Os processos de codificação e decodificação dos sinais são extremamente custosos em termos de CPU, geralmente consumindo todo processamento disponível no nó. Este tipo de aplicação pode se beneficiar melhor dos serviços disponibilizados pela nossa arquitetura, já que para o multiplexador o processamento é um recurso sensível.

Desenvolvemos o multiplexador de áudio baseado no código do refletor multimídia, uma vez que as aplicações são semelhantes. O refletor possuía apenas uma conexão de origem de dados e várias de destino. Já o multiplexador pode ter diversas origens de dados e diversos destinos. Nosso protótipo foi desenvolvido para mixar apenas dois canais de áudio. Além disso, a principal diferença entre as duas aplicações é relacionada ao processamento de sinal. O refletor não processava o sinal recebido, apenas o enviava aos clientes da mesma forma que o recebia. O multiplexador, por sua vez, processa os sinais recebidos e envia aos clientes o sinal processado.

A biblioteca padrão para manipulação de arquivos de áudio do Java, a Java Sound API, não oferece mais métodos para manipulação de arquivos MP3 por questões relacionadas aos direitos autorais da tecnologia MP3. Utilizamos então a biblioteca Tritonus[1], que é uma implementação de código aberto da Java Sound, com funções para facilitar a utilização de arquivos MP3. Ainda assim, a manipulação de arquivos de áudio em Java pode gerar diversas condições de erro e como resultado o código-fonte fica poluído e de difícil leitura. Por este motivo não incluímos fragmentos do código do multiplexador de áudio neste documento. O código do multiplexador de áudio pode ser obtido na página do projeto na Internet[2].

Em nossos testes, utilizando um computador Pentium IV 1.5GHz com 512MB de memória RAM, a aplicação não obteve o desempenho necessário para realizar seu objetivo. O tempo necessário para processar cada byte de dado causava paradas perceptíveis no áudio e até mesmo a desconexão da aplicação cliente. Ainda há espaço para otimizações no código do multiplexador, mas dificilmente a aplicação terá o desempenho necessário para mixar dois canais de áudio em MP3 em tempo real,

[1] http://tritonus.org

[2] http://gsd.ime.usp.br/software/QoSNegotiation

sem paradas perceptíveis, utilizando a linguagem Java em um computador deste porte.

Capítulo 7

Experimentos

Para validarmos a nossa arquitetura, realizamos diversos testes com o refletor de áudio que desenvolvemos, descrito na Seção 6.1. Nossos testes procuraram validar o impacto da negociação de requisitos em relação ao processo completo de migração da aplicação, principalmente em termos de escalabilidade, uma vez que pretendemos utilizar nossa arquitetura em ambientes de computação ubíqua, que são tipicamente constituídos por uma grande quantidade de nós.

7.1 Descrição do Ambiente de Testes

Para realizarmos nossos testes, utilizamos 10 computadores conectados por uma rede Fast Ethernet de 100Mbps. Todos os computadores possuíam processadores AMD Athlon XP 1700 e 512MB de memória RAM. Nosso ambiente de testes está descrito na Figura 7.1. Para realizar nossos testes, utilizamos o Darwin Streaming Server [1] como servidor de fluxos de arquivos MPEG Layer 3 (MP3). Os refletores que nós desenvolvemos se conectam ao Darwin Streaming Server e repassam o tráfego de dados aos clientes da aplicação. Estes clientes utilizam o XMMS[2] para se conectarem aos refletores e tocarem o fluxo de áudio transmitido. Tanto o servidor de fluxos quanto a estação cliente estavam conectados à rede de testes através da Internet.

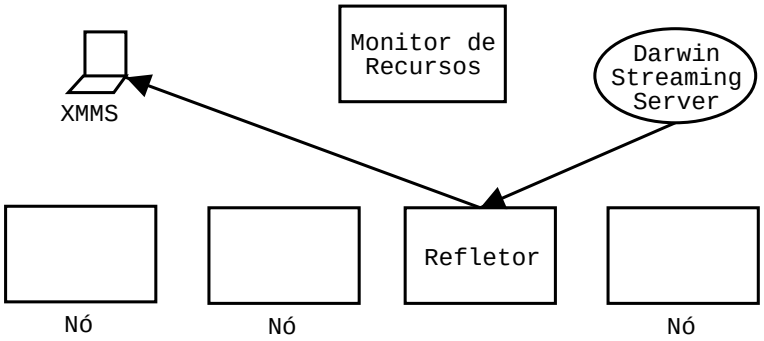

Figura 7.1: Ambiente utilizado para testes

Um dos computadores foi utilizado como nó central de gerenciamento do Arcabouço descrito

[1] http://developer.apple.com/darwin/projects/streaming
[2] http://www.xmms.org

na Seção 3.4. Este nó executava um servidor MySQL e armazenava o repositório de entidades do sistema. Para que o processamento utilizado pelo repositório não distorcesse os resultados dos testes, este nó não foi incluído no processo de negociação. Caso ele fosse incluído, a negociação neste nó seria mais lenta e limitaria superiormente os tempos de negociação. Por este motivo, os agentes de negociação visitavam apenas os 9 computadores restantes.

7.2 Experimentos Realizados

Foram realizados testes considerando-se três cenários possíveis em um ambiente real.

Cenário 1: - O servidor de aglets de um nó acabou de ser iniciado e nenhum agente foi executado pelo servidor.

Cenário 2: - O servidor de aglets foi iniciado e algum aglet visitou o nó. O agente de negociação, entretanto, ainda não visitou o nó.

Cenário 3: - O servidor de aglets foi iniciado e, em algum momento passado, um agente de negociação visitou o nó.

As situações descritas em cada cenário afetam o tempo de carga dos agentes e das classes necessárias para negociação de contratos de QoS. O primeiro cenário é o mais pessimista, já que nem as classes do agente de negociação, nem as classes para execução de aglets estão em memória. No segundo cenário, as classes do agente de negociação não estão em memória, mas as classes do servidor de aglet - necessárias para a execução de agentes móveis - estão. Finalmente, no terceiro cenário, tanto as classes do agente de negociação quanto as classes do servidor de aglets já foram carregadas para a memória.

Estes três cenários foram testados com 1, 2, 4 e 8 nós participando do processo de negociação. Cada cenário foi avaliado com a estratégia de procura paralela e com a estratégia de procura linear. Cada teste foi repetido 11 vezes, pois baseado no desvio padrão dos testes preliminares, com esta amostra obteríamos um intervalo de confiança de 95% estreito o suficiente [Jai91].

Foi medido o tempo total da negociação, incluindo-se o tempo necessário para que a aplicação migrasse de nó. O tempo de migração foi medido da seguinte forma: no nó original, foi criado um *socket* e assim que a aplicação migrava para um novo nó, sua primeira ação era se conectar a este *socket*.

O processo de negociação foi dividido em quatro etapas principais. São elas:

Criação de agentes para negociação: Compreende o tempo utilizado para criação de todos agentes necessários para a negociação.

Negociação com os nós: Tempo gasto na negociação com cada nó, incluindo-se os tempos de migração dos agentes de negociação.

Seleção da melhor oferta: Tempo utilizado pela estratégia de adaptação da aplicação para selecionar a melhor oferta dentre todas recebidas.

Migração da aplicação: Tempo necessário para que a aplicação deixasse o nó de origem e chegasse ao novo nó.

7.3 Resultados dos Experimentos

A Figura 7.2 mostra o tempo total de negociação para cada um dos 3 cenários utilizando a estratégia de busca linear.

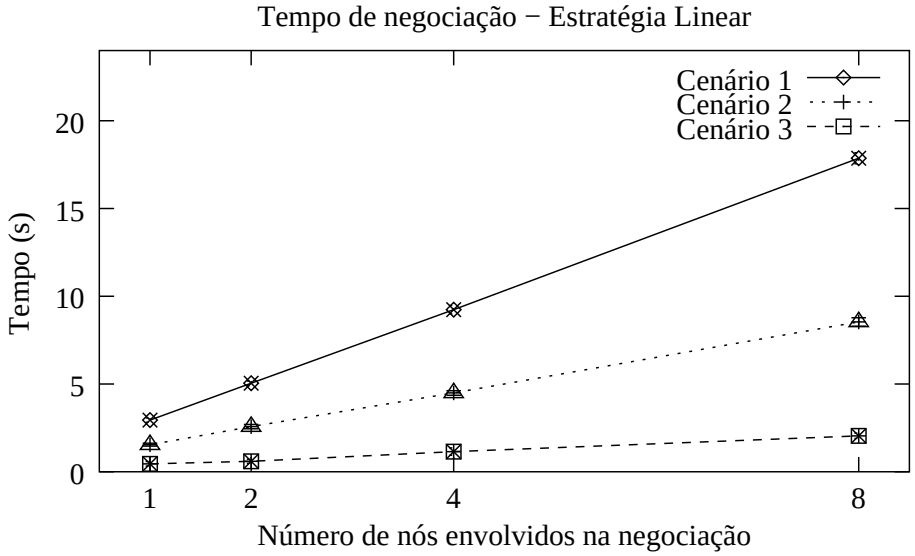

Figura 7.2: Tempo de negociação da estratégia linear em cada cenário

Neste gráfico podemos observar que houve um crescimento linear do tempo de negociação em função da quantidade de nós. Isto era esperado, já que na estratégia linear o mesmo agente de negociação visita cada um dos nós envolvidos no processo de negociação. Como conseqüência, os tempos de negociação em cada agente são somados.

Além disso, no caso da negociação linear o tempo total do cenário 1 chegou a ser oito vezes mais lento que o do cenário 3 e o seu crescimento foi mais acentuado. Isto é justificável já que, no caso da estratégia linear, a sobrecarga causada pelo carregamento dos códigos do agente e do servidor de aglets é multiplicada pelo número de nós. Se considerarmos o cenário 3, em que esta sobrecarga é menor, o crescimento foi bem menos acentuado.

As Figuras 7.3, 7.4 e 7.5 mostram o porcentual de participação de cada uma das etapas em relação ao tempo total de negociação na estratégia de procura linear. Os resultados estão divididos em quatro etapas: criação de agentes, negociação com os nós, seleção do melhor resultado e migração da aplicação.

Nos três cenários, conforme o número de nós envolvidos na negociação aumenta, o tempo gasto com a negociação com cada nó passa a dominar o tempo total. Na negociação linear, o tempo para criação de agentes de negociação é constante pois independentemente do número de nós, sempre é criado apenas um agente. Os tempos relativos à escolha da melhor oferta e à migração da aplicação são inexpressivos quando comparados ao tempo utilizado pela negociação.

A Figura 7.6 mostra uma comparação entre o tempo de negociação da estratégia paralela em cada um dos três cenários avaliados. Neste caso, podemos observar que a negociação no cenário 1 chega a ser até cinco vezes mais lenta do que no cenário 3. Estes números entretanto não são preocupantes, uma vez que, em um ambiente real, a situação descrita pelo cenário 1 ocorrerá raramente. Em um tal ambiente, será muito mais comum que a negociação de requisitos envolva

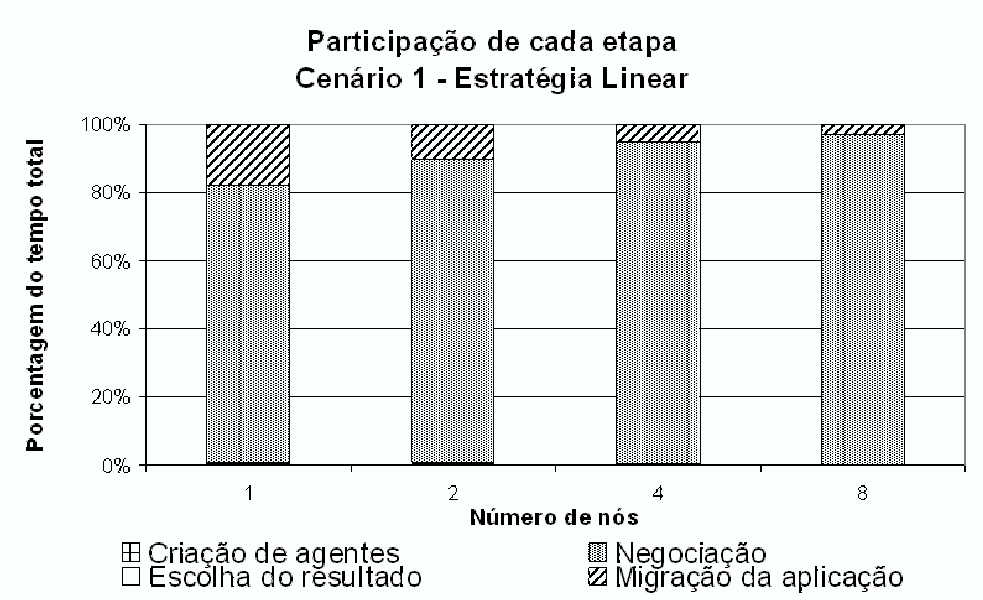

Figura 7.3: Total por etapa da estratégia linear - Cenário 1

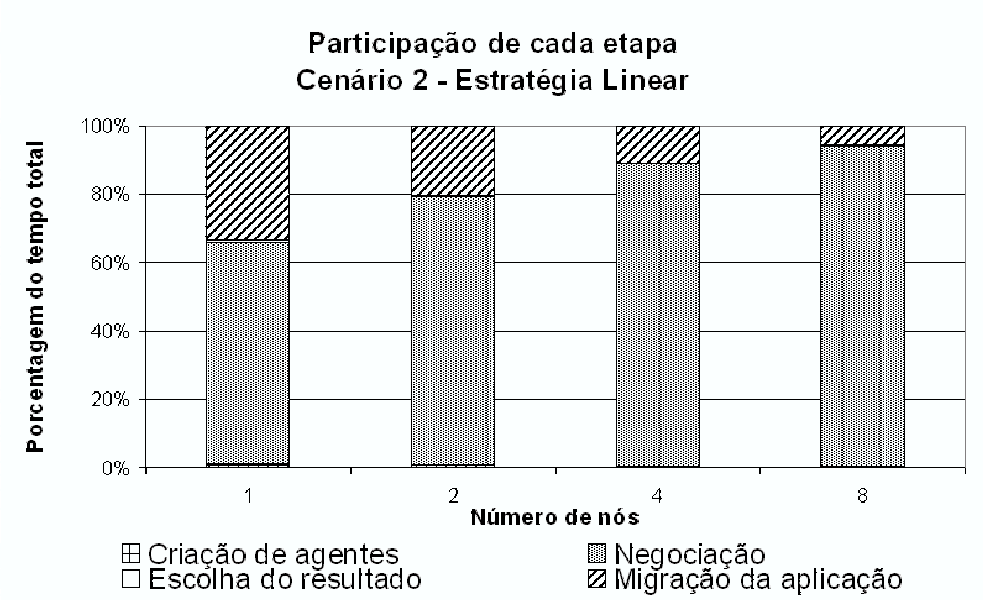

Figura 7.4: Total por etapa da estratégia linear - Cenário 2

nós que em algum momento já passaram por algum processo de negociação no passado. Os valores de tempo de negociação em um ambiente real, portanto, devem se aproximar dos valores do cenário 3.

Além disso, percebemos que há um aumento do tempo de negociação em função da quantidade de nós envolvidos. Em uma estratégia de negociação paralela, em tese, isto não deveria ocorrer:

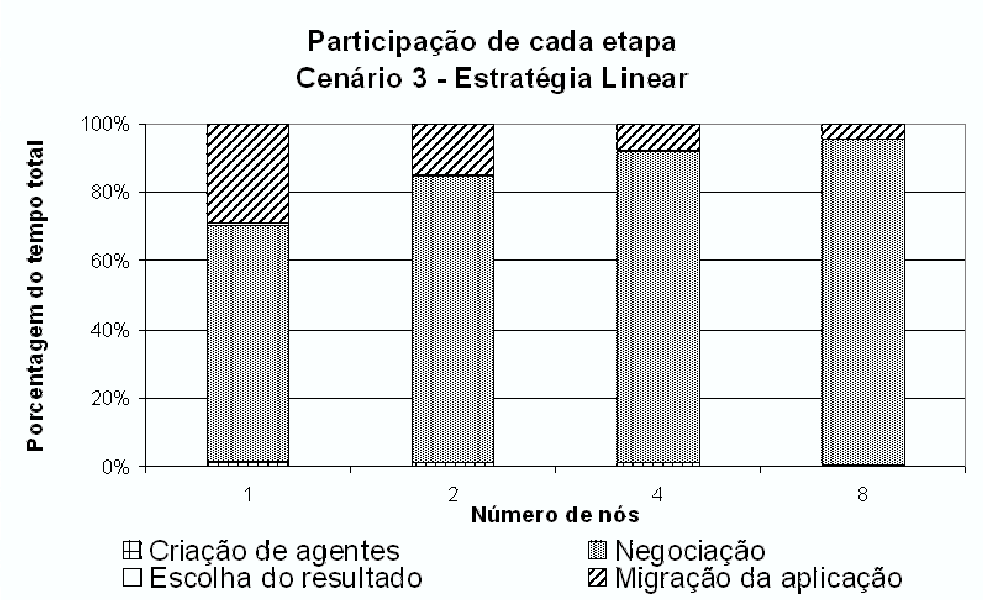

Figura 7.5: Total por etapa da estratégia linear - Cenário 3

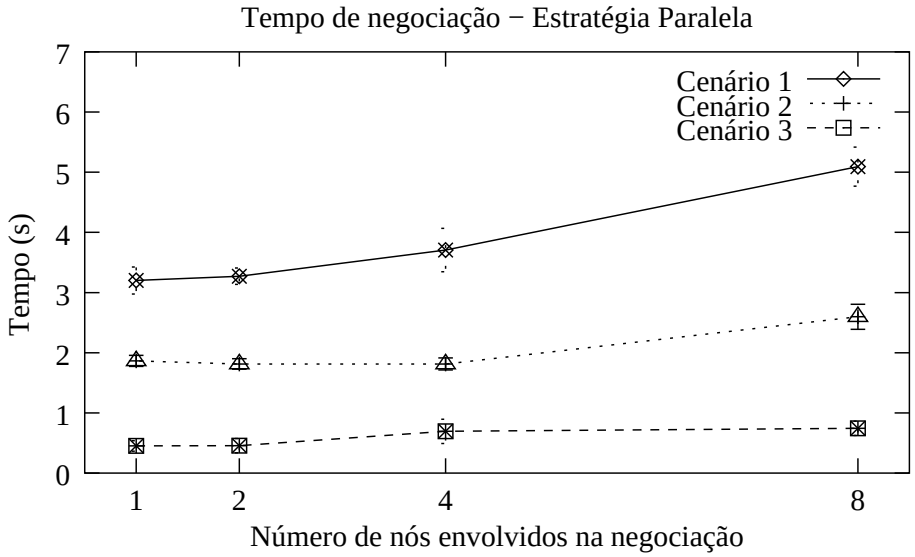

Figura 7.6: Tempo de negociação da estratégia paralela em cada cenário

como os agentes negociam com todos os nós simultaneamente, a negociação deveria demorar o mesmo tempo independente da quantidade de nós envolvidos. Todavia, os agentes são enviados para os nós um a um. Com isso, o tempo que cada nó leva para carregar o agente de negociação acaba impactando no tempo total. No caso do cenário 3, onde o tempo necessário para carregar o agente é bem menor, o tempo total de negociação se mantém quase o mesmo para 2, 4 e 8 nós.

Este problema poderia ser minimizado caso o processo de criação de agentes para negociação

fosse paralelizado. Atualmente, a migração está sendo feita de forma seqüencial. O agente de negociação é criado, enviado ao seu nó de destino e, só depois de recebido no nó destino, o agente seguinte é criado. Isto pode gerar atrasos desnecessários caso um dos nós visitados demore para receber o agente. Se o processo de envio dos vários agentes for realizado utilizando *non-blocking* TCP/IP, este atraso pode ser muito menor.

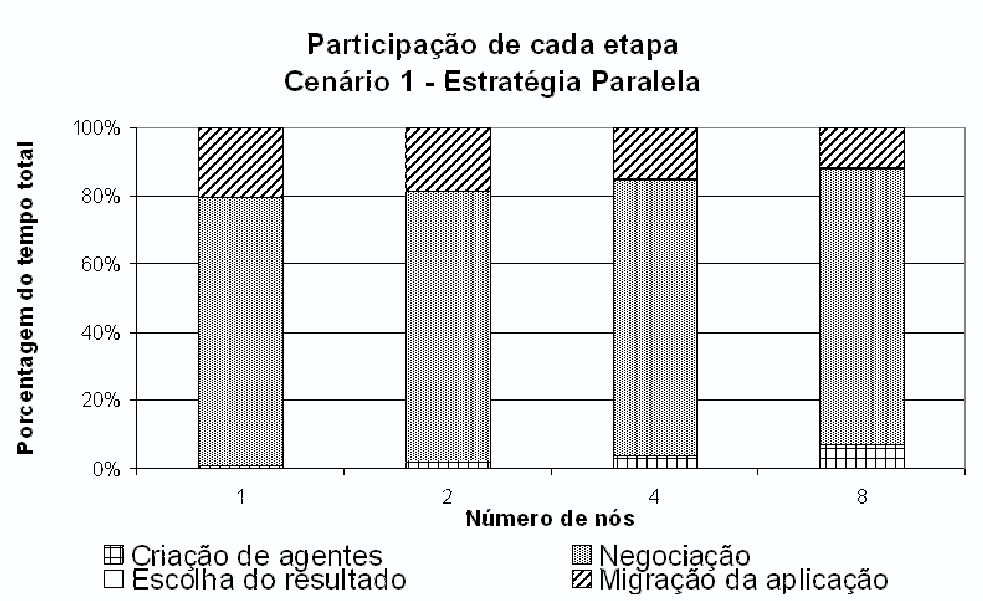

Figura 7.7: Total por etapa da estratégia paralela - Cenário 1

As Figuras 7.7, 7.8 e 7.9 mostram a participação porcentual de cada etapa do processo de negociação em relação ao tempo total de negociação. No cenário 1, o processo de negociação é dominado pelo tempo de migração da aplicação e pelo tempo de negociação com cada nó. Como no cenário 1 nenhuma classe foi carregada para a memória, as atividades que exigem que os nós carreguem agentes são excessivamente lentas, uma vez que o interpretador Java deve carregar uma grande quantidade de código para memória.

No cenário 2 o mesmo perfil se repete, apesar de a negociação com 8 nós mostrar uma participação maior do tempo de criação de agentes para negociação. No cenário 3 esta característica se consolida e, apesar do processo de negociação ainda ter a maior participação em relação ao total, a migração da aplicação e a criação dos agentes de negociação passam também a serem críticos para o desempenho do processo de negociação. A tendência que observamos é que conforme a quantidade de nós aumenta, o tempo de criação dos agentes passa a ser crítico para o desempenho da arquitetura. Como mencionamos anteriormente, a paralelização do processo de criação dos agentes de negociação pode diminuir o tempo utilizado por esta etapa.

Comparando-se com os resultados da estratégia linear, na estratégia paralela a etapa de negociação não representa a maior parte do tempo total de negociação. Em contrapartida, o tempo necessário para criar agentes passa a ser um gargalo de desempenho, o que não acontecia na estratégia linear.

As Figuras 7.10, 7.11 e 7.12 mostram comparações entre os tempos utilizados na negociação com a estratégia paralela e linear.

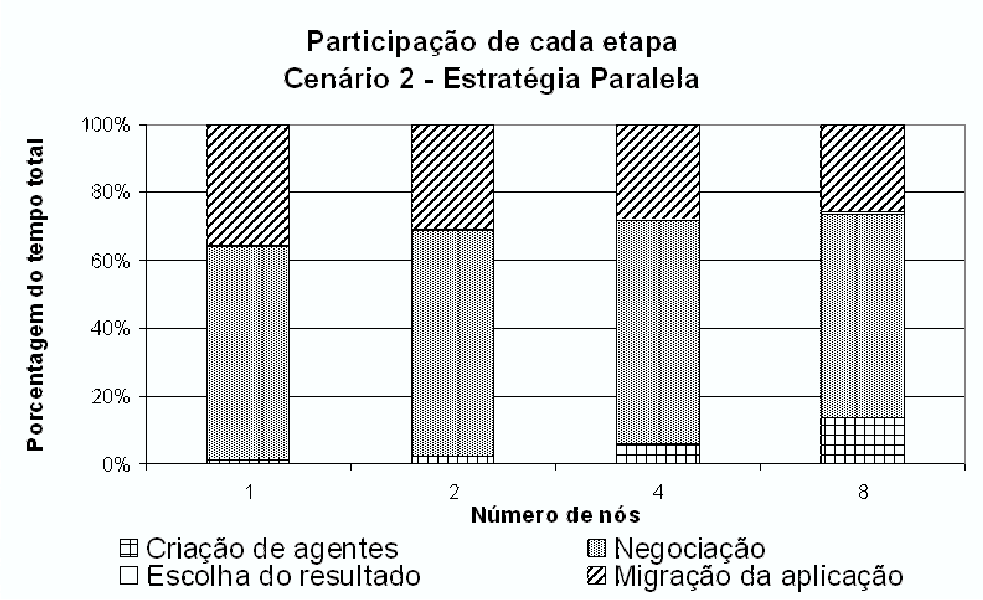

Figura 7.8: Total por etapa da estratégia paralela - Cenário 2

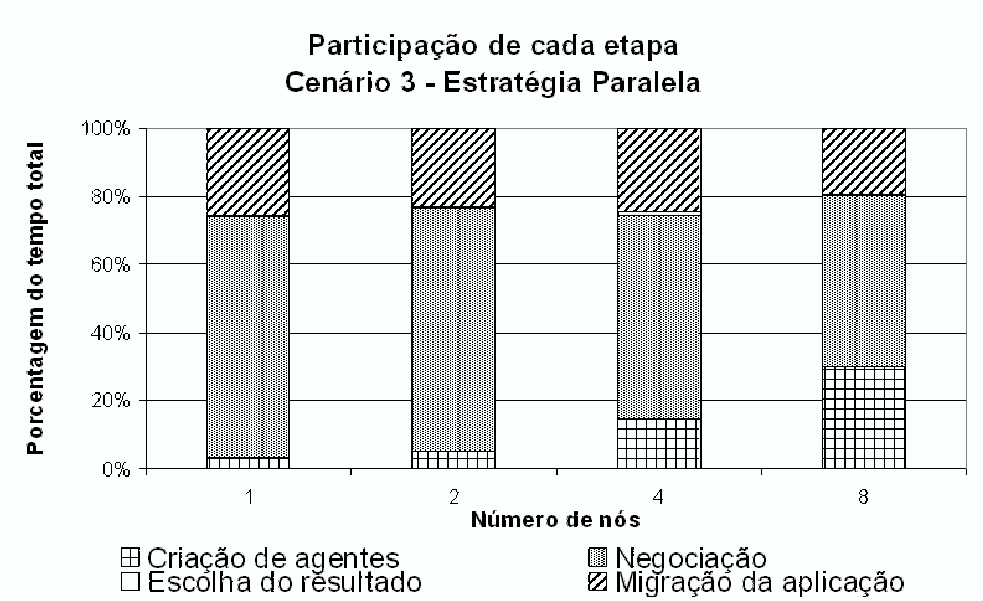

Figura 7.9: Total por etapa da estratégia paralela - Cenário 3

Podemos perceber nos três cenários que a curva de crescimento de tempo é mais acentuada quando utilizamos a estratégia linear. As comparações entre o cenário 1 e o cenário 2 são bem semelhantes e mostram um leve aumento no tempo de negociação da estratégia paralela, enquanto o aumento do tempo de negociação da estratégia linear é constante.

Já o gráfico de comparação do cenário 3 mostra uma curva um pouco diferente. Enquanto o

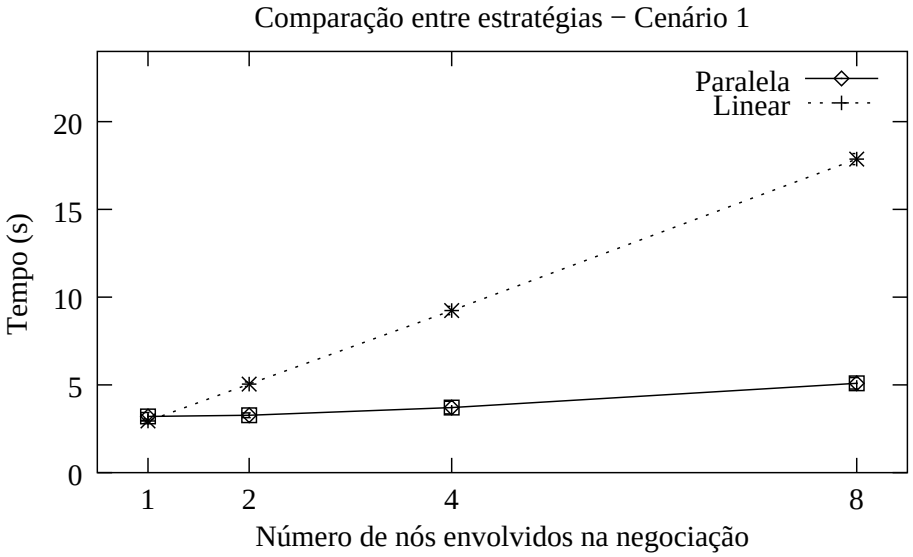

Figura 7.10: Comparação entre estratégia paralela e linear - Cenário 1

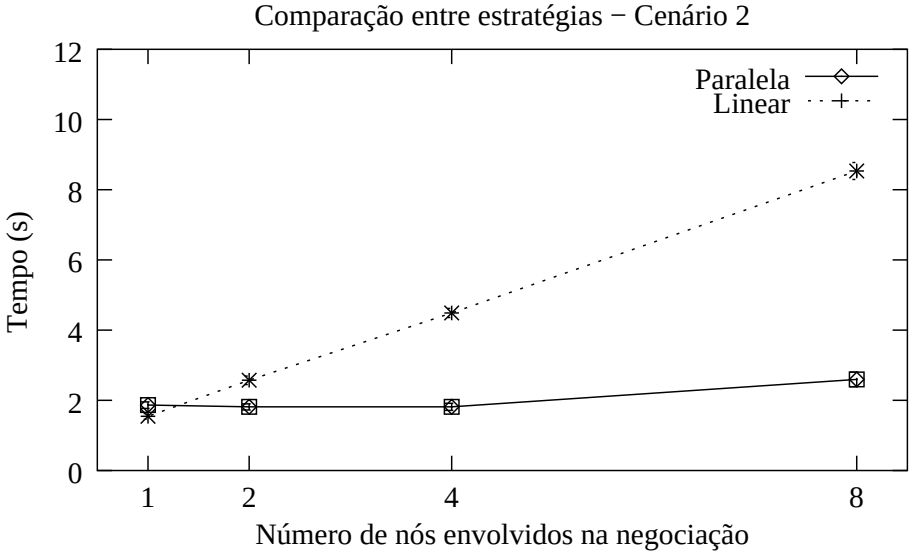

Figura 7.11: Comparação entre estratégia paralela e linear - Cenário 2

tempo de negociação da estratégia linear continua crescendo de forma constante de acordo com o aumento do número de nós, o tempo de negociação da estratégia paralela se mantém praticamente o mesmo com o aumento de nós. Como este cenário é o que esperamos ser mais comum em um ambiente real, na prática é possível aumentar a quantidade de nós envolvidos na negociação de requisitos sem um grande aumento no tempo total utilizado para a negociação, utilizando-se a estratégia paralela.

Obviamente, a estratégia paralela causa uma sobrecarga maior no ambiente em comparação com

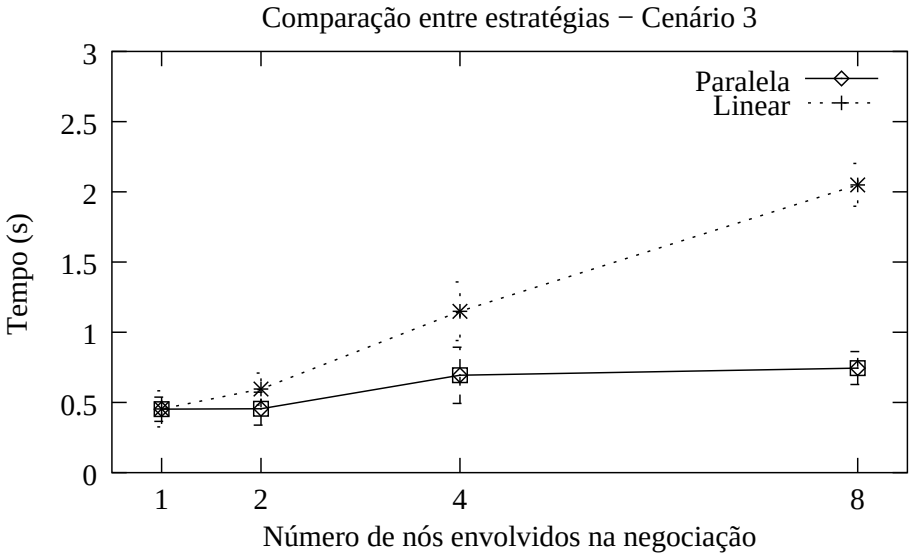

Figura 7.12: Comparação entre estratégia paralela e linear - Cenário 3

a estratégia linear e a aplicação deve decidir entre receber os resultados da negociação o mais rápido possível ou utilizando a menor quantidade de recursos. Eventualmente poderíamos acrescentar à arquitetura uma estratégia de negociação mista que pudesse definir, em função do número de nós visitados durante a negociação, quantos agentes deveriam ser criados e quantos nós deveriam ser visitados por cada agente. Isto possibilitaria um balanço entre o tempo utilizado pela negociação (que é menor na estratégia paralela) e o tempo utilizado para criação dos agentes de negociação (que é menor na estratégia linear).

Capítulo 8

Principais Contribuições

Grande parte dos resultados obtidos ao longo desta pesquisa foram influenciados por uma decisão que tomamos no início do processo de implementação de nossa arquitetura. Nesta etapa, decidimos usar a maior quantidade possível de trabalhos já existentes nas áreas abrangidas pela nossa pesquisa, utilizando nossa arquitetura para identificar problemas com estas tecnologias e para complementá-las quando necessário.

Nesta seção apresentamos nossas contribuições originais. Elas incluem, além da arquitetura propriamente dita, duas ferramentass criadas para complementar as tecnologias utilizadas em nossa pesquisa. Estas contribuições foram descritas em dois artigos que escrevemos durante este trabalho.

8.1 Arquitetura de Negociação

A arquitetura de negociação é o foco principal do nosso trabalho. A área de negociação de requisitos de QoS não tem despertado grande interesse da comunidade científica e poucos trabalhos se dedicaram a estudar este problema. Ainda assim, entendemos que o problema da negociação de QoS é importante, principalmente para sistemas altamente distribuídos como algumas aplicações executadas na Internet ou as aplicações desenvolvidas para ambientes de Computação Ubíqua. Nosso trabalho visa contribuir com o estudo nesta área e diminuir a escassez de pesquisas neste assunto.

Além disso, de acordo com nosso conhecimento, esta é a primeira vez que se propõe o uso de agentes móveis para negociação de QoS. Esta é também uma contribuição à pesquisa em agentes móveis, pois apesar de a tecnologia ter sido introduzida há quase dez anos e ter diversas virtudes em relação a outras estratégias de desenvolvimento de aplicações distribuídas, ela ainda é pouco usada. Nosso trabalho contribui apresentando mais uma aplicação para a qual agentes móveis podem trazer melhorias de desempenho e flexibilidade em relação a outros paradigmas.

8.2 *Parser* QML para o DSRT

Para desvincularmos a especificação de requisitos do código fonte, desenvolvemos um *parser* QML para o DSRT. Ele permite que as aplicações definam suas necessidades de requisitos em um arquivo à parte, utilizando a linguagem QML. Este arquivo é então interpretado em tempo de execução e os agentes de negociação utilizam as informações nele contidas para negociar contratos de utilização

de recursos em favor da aplicação. O *parser*, descrito na Seção 4.1.1, foi desenvolvido em Java utilizando-se o JFlex e o BYacc/J.

8.3 *Broker* CORBA para o DSRT

O *broker* é um componente muito importante para qualquer sistema de reserva de recursos para garantia de QoS. Ele é responsável por realizar o controle de admissão de reservas, além de outros serviços necessários ao gerenciamento das solicitações de reserva de recursos.

Apesar do DSRT já possuir um *broker*, ele não atendia às nossas necessidades neste projeto. O *broker* original do DSRT não diferencia os conceitos de solicitações de reserva e de reservas efetivas de recursos. Desta forma as aplicações não conseguiriam testar uma configuração de reserva de recursos que poderia ou não ser efetuada depois, dependendo dos resultados do processo de negociação de requisitos. Toda solicitação feita ao *broker* original do DSRT, se aprovada pelo controle de admissão, é efetivada junto ao DSRT. Este comportamento não é o que procurávamos.

Além disso, o *broker* original do DSRT não é um processo separado do núcleo do DSRT. Se uma aplicação deseja se beneficiar das funcionalidades do *broker*, ela deve utilizar uma versão alternativa do DSRT, na qual o *broker* age como uma camada de software acima da API original do DSRT. O *broker* realiza as validações necessárias nas solicitações das aplicações e depois passa estas solicitações ao núcleo do DSRT.

Resolvemos então desenvolver um novo *broker*, capaz de fornecer, aos agentes de negociação, os serviços dos quais eles necessitavam e que pudesse ser um processo separado do núcleo do DSRT, permitindo uma melhora no desempenho da negociação. Desenvolvemos o nosso *broker* utilizando CORBA para facilitar a interação entre as aplicações e o DSRT.

A Figura 8.1 mostra a diferença entre o *broker* desenvolvido durante este trabalho e o *broker* original do DSRT. No *broker* original, as aplicações interagiam diretamente com o DSRT. Como a interface Java do DSRT foi desenvolvida utilizando a Java Native Interface (JNI), toda vez que um agente de negociação necessitasse interagir com o *broker* ele deveria carregar um módulo para a memória contendo as definições das funções disponibilizadas pela API do DSRT. Esta operação é cara em termos de desempenho.

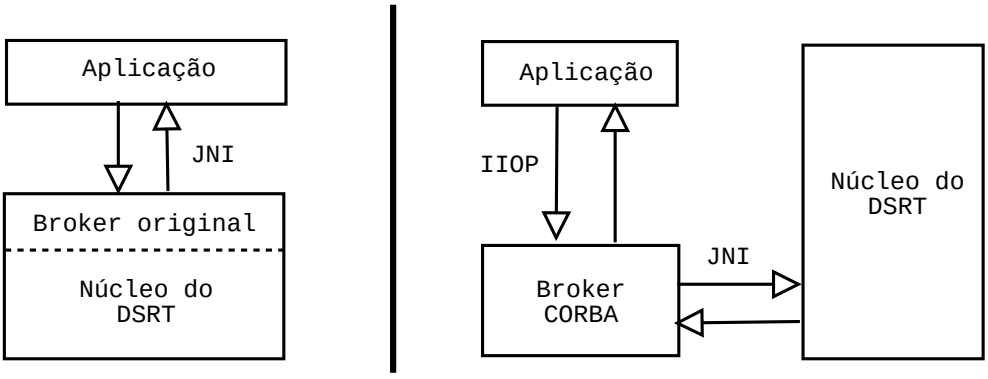

Figura 8.1: Funcionamento do *broker* original do DSRT comparado ao *broker* desenvolvido neste trabalho

No nosso *broker*, por outro lado, a aplicação não interage diretamente com o DSRT. As requisi-

ções de reserva de recursos são passadas para o *broker* CORBA, que se comunica com o núcleo do DSRT. Neste caso, o módulo com as funções da API é carregado apenas uma vez, quando o *broker* é iniciado. Os agentes de negociação não precisam do módulo, já que se comunicam apenas com o *broker* e não diretamente com o núcleo do DSRT.

O *broker* que desenvolvemos é responsável por três serviços: controle de admissão, sugestão de contratos e gerenciamento de reservas. O controle de admissão é feito de forma muito simples: o *broker* recebe a solicitação de requisitos da aplicação e tenta realizar a reserva junto ao DSRT. Se a reserva é feita com sucesso, o *broker* gera um *ticket* para a aplicação poder efetuar esta reserva posteriormente. Os recursos são reservados pelo DSRT durante a duração do *ticket*. Caso a aplicação não inicie sua reserva dentro do prazo de validade do *ticket*, os recursos reservados são liberados pelo *broker* e a aplicação não pode mais iniciar a reserva.

A sugestão de contratos pode ser solicitada pelo agente de negociação durante a negociação de requisitos de QoS. A sugestão de contrato sempre é feita baseada em um contrato passado pelo agente de negociação. A partir deste contrato, o *broker* realiza uma busca binária pelo maior valor de reserva aceito pelo DSRT. Quando este valor é encontrado, é passado à aplicação como sugestão de reserva.

Finalmente, o gerenciamento de reservas é realizado através dos *tickets*. Quando o *broker* aceita uma reserva, um *ticket* é gerado para a aplicação. Este *ticket* possui uma identificação e um período de validade, tipicamente de alguns segundos. Os *tickets* gerados pelo *broker* são armazenados em uma fila e quando o prazo de validade de um *ticket* expira, o próprio *ticket* notifica o *broker* para que as medidas necessárias sejam tomadas, como por exemplo liberar os recursos alocados para a aplicação. Por outro lado, se a aplicação inicia a utilização de sua reserva de recursos antes do *ticket* expirar, o *ticket* é retirado da fila, apagado e os recursos são garantidos para a aplicação.

Este *broker* também pode ser utilizado por outras aplicações que possuam necessidades semelhantes às do nosso sistema de negociação. Ele foi desenvolvido como um módulo separado da arquitetura principal de negociação e pode ser reutilizado e extendido por outras aplicações e outros projetos.

8.4 Artigos Publicados

Durante a elaboração deste trabalho, apresentamos dois artigos em eventos internacionais relevantes, relacionados à pesquisa que realizamos:

ACM OOPSLA 2002 Workshop on Pervasive Computing - Durante a série de workshops que fizeram parte da conferência OOPSLA 2002 em Seattle, EUA, tivemos a oportunidade de apresentar o artigo **Mobile Agents: A Key for Effective Pervasive Computing**. Neste artigo descrevemos as idéias principais de nossa arquitetura e como ela poderia contribuir para tornar mais eficiente as aplicações de computação ubíqua.

International Symposium on Distributed Objects and Applications (DOA) 2004 - Em 2004, apresentamos o artigo **A Mobile Agent Infrastructure for QoS Negotiation of Adaptive Distributed Applications** na trilha principal da conferência DOA, em Agya Napa, Chipre. Neste artigo, introduzimos a arquitetura, discutimos detalhes de sua implementação e apresentamos alguns resultados experimentais preliminares. Os anais da conferência foram publicados no volume 3291 da série *Lecture Notes in Computer Science*.

8.5 Trabalhos Futuros

Gostaríamos de pesquisar ainda outros aspectos relacionados ao sistema desenvolvido durante este trabalho. Mas em algumas vezes não tivemos tempo, em outras a pesquisa fugia ao escopo deste trabalho e em outras ainda motivos alheios à nossa vontade nos impediram de continuar. Nesta seção, descrevemos algumas idéias que tivemos ao longo da pesquisa, mas que não puderam ser levadas mais a fundo.

Estas idéias poderão ser estudadas com o intuito de tornar este trabalho mais completo e podem servir de base para projetos de iniciação científica, mestrado e até mesmo para uma tese de doutorado.

8.5.1 Reserva de Requisitos de Rede

Conforme argumentado na Seção 2.3, uma solução completa para gerenciamento de QoS em sistemas distribuídos deve considerar não só a disponibilidade e alocação de recursos nos nós do sistema, mas também deve considerar a utilização dos recursos de rede. O desempenho da rede influencia bastante a qualidade da saída gerada por uma aplicação distribuída e pode ser considerado um recurso sensível deste tipo de aplicação. Desta forma, seu uso deve ser disciplinado e o sistema de gerenciamento de QoS tem de ser capaz de alocar estes recursos e garantir sua disponibilidade para aplicações com requisitos de QoS.

Além disso, as aplicações também devem obter informações sobre a utilização dos recursos de rede, para poder decidir quando é necessário se adaptar às mudanças ocorridas no ambiente e onde a aplicação pode obter melhores condições de utilização deste recurso. Em outras palavras, seriam necessários sistemas para **monitoração** e **garantia** de recursos de rede. Eles desempenhariam as mesmas funções que o arcabouço para adaptação dinâmica de sistemas distribuídos e o DSRT desempenham em relação aos recursos dos nós.

Para a monitoração de recursos de rede, poderia ser utilizado o ReMoS, apresentado na Seção 2.3. Ele possui diversos mecanismos para obtenção de métricas de utilização da rede em um sistema distribuído e deve atender às necessidades da arquitetura. Já para alocação de recursos de rede e garantia de requisitos de rede, não encontramos um sistema que possua as características que desejamos. O problema de reserva de recursos em redes Ethernet é muito complexo, uma vez que o próprio modelo Ethernet não prevê mecanismos para garantia de qualidade de serviço, ao contrário, por exemplo, da arquitetura de redes ATM. Entretanto, esta é uma área de estudos em desenvolvimento e uma pesquisa mais cuidadosa, entre os trabalhos existentes e os que ainda surgirão, pode apontar um sistema capaz de abordar este problema de forma satisfatória.

Finalmente, seria necessário integrar os dois sistemas à arquitetura já existente. A complexidade desta integração pode variar de acordo com os sistemas que forem integrados, mas a tendência é que seja uma tarefa simples. Com isso, teríamos um sistema capaz de garantir a qualidade de serviço de ponta a ponta, no qual as aplicações pudessem monitorar os níveis de disponibilidade dos recursos da rede e dos nós, utilizando estas informações para tomar decisões de adaptação.

8.5.2 Definição do Posicionamento dos Agentes

Esta idéia surgiu a partir das perguntas realizadas pelos participantes da conferência DOA 2004, após a apresentação do nosso artigo. Imagine um cenário em que as aplicações distribuídas sejam compostas não por um, mas por vários agentes móveis diferentes. Estes agentes possivelmente trocariam mensagens ao longo do seu funcionamento para realizar algum trabalho.

A migração ou a clonagem desta aplicação envolveria todos os seus agentes. O processo de adaptação poderia levar isto em conta, por exemplo, definindo a melhor distribuição dos agentes pelos nós de acordo com os seus padrões de comunicação. Desta forma, seria possível saber quais agentes devem ser co-locados, quais precisam estar conectados por uma rede de alto desempenho e quais podem estar conectados através de redes mais lentas.

Esta disposição dos agentes influenciaria o próprio processo de negociação de requisitos, que se tornaria muito mais complexo. Os agentes de negociação não seriam mais responsáveis por negociar os requisitos de um agente em um nó. Eles teriam de negociar, ao mesmo tempo, os recursos de diversos agentes em diversos nós. Caso a disposição ótima de agentes não pudesse ser obtida, outras disposições alternativas poderiam ser negociadas.

Para que isso fosse possível, seria necessário monitorar as interações entre os agentes do sistema. O arcabouço para adaptação dinâmica de sistemas distribuídos já é capaz de monitorar as interações entre objetos CORBA [dSeSEK02] utilizando o mecanismo de interceptadores. Talvez este mesmo mecanismo pudesse ser utilizado para monitoração das interações entre agentes.

Portanto, a inclusão de um mecanismo capaz de definir a melhor distribuição dos agentes de uma aplicação no sistema dá margem ao estudo de diversos problemas interessantes. Além disso, este mecanismo adicionaria um serviço importante ao sistema, que traria ganhos de desempenho às aplicações que se utilizassem de nossa arquitetura.

8.5.3 Estratégia de Procura *peer-to-peer*

Nesta pesquisa, desenvolvemos duas estratégias de procura da melhor oferta de requisitos de QoS pelo sistema: a estratégia linear e a estratégia paralela. Na Seção 4.3.1, além de apresentar as duas estratégias, sugerimos a criação de uma estratégia *peer-to-peer*.

A idéia de uma procura por ofertas de requisitos que utilize a estratégia *peer-to-peer* é fazer com que um agente de negociação deixe o servidor de origem da aplicação sabendo apenas qual será seu primeiro nó de destino. Dependendo dos resultados da negociação neste primeiro nó, o agente poderia decidir de forma autônoma qual seria seu próximo destino.

Este tipo de procura poderia ser útil no cenário descrito na seção anterior, por exemplo. Imagine que um agente de negociação deve negociar os requisitos de uma aplicação formada por vários agentes, que possua algumas possibilidades de disposição dos agentes em vários nós. Dependendo do resultado das negociações com o primeiro nó, o agente de negociação pode definir qual das disposições será negociada e então determinar qual será o seu próximo destino.

8.5.4 Integração com o Gaia

Finalmente, uma de nossas intenções iniciais era de integrar nossa arquitetura com um sistema de computação ubíqua, conforme afirmamos no Capítulo 2. Dentre os sistemas de computação ubíqua pesquisados, o mais completo era o Gaia. Entretanto, apesar de operacional, o Gaia não possui ainda uma distribuição pública, a qual possamos alterar e acrescentar funcionalidades.

A integração da nossa arquitetura ao Gaia auxiliaria no processo de garantia de requisitos de QoS para aplicações de computação ubíqua. Esta classe de aplicações é sensível à qualidade de serviço, já que o principal objetivo de ambientes de computação ubíqua é tornar os computadores "invísiveis" e permitir que os usuários interajam com eles como com outras tecnologias cotidianas. Assim sendo, os estímulos dos usuários aos computadores espalhados pelo ambiente devem ser prontamente respondidos, pois caso contrário o objetivo de desaparecer com os computadores do

sistema não seria conquistado. Isto prejudicaria o desempenho do sistema e tornaria a experiência do usuário frustrante.

Os pesquisadores do Gaia pretendem lançar uma distribuição púlbica do código-fonte do sistema no futuro. Apenas após ter contato com os detalhes da implementação do Gaia será possível determinar a complexidade e a viabilidade da integração da nossa arquitetura com o software já desenvolvido. Até lá, devemos acompanhar os esforços dos pesquisadores em obter uma versão estável do sistema e torcer para que a distribuição pública seja lançada num futuro próximo.

8.6 Conclusões

Nesta tese apresentamos os resultados de nossa pesquisa na área de Qualidade de Serviço, principalmente os resultados relacionados à negociação de contratos de requisitos de QoS. Detalhamos nossa arquitetura para garantia de QoS que permite às aplicações especificarem seu requisitos, monitorarem o nível de QoS oferecido pelo sistema e se adaptarem às mudanças de disponibilidade de recursos, negociando e renegociando seus contratos de requisitos.

Apresentamos também duas aplicações que foram utilizadas para validar o funcionamento da arquitetura. Utilizando estas aplicações, realizamos alguns testes para avaliarmos com mais detalhes o mecanismo de negociação e apresentamos os resultados destes experimentos. Estes resultados mostraram que o mecanismo de negociação é flexível em termos de uso de recursos e tempo de resposta da negociação. Este mecanismo também é escalável, já que é possível obter resultados rápidos mesmo quando são envolvidos vários nós, ou então consumir poucos recursos do ambiente, dependendo da decisão da aplicação.

Acreditamos que os agentes móveis apresentam diversas vantagens que ainda não foram totalmente exploradas. Apesar do conceito de agentes móveis ter sido introduzido há quase dez anos atrás, as tecnologias relacionadas ainda não são populares entre os desenvolvedores. Entretanto, outro conceito que hoje está bem estabelecido - a orientação a objetos - levou cerca de 20 anos para que as tecnologias relacionadas fossem desenvolvidas e o conceito se popularizasse. Nós acreditamos que são necessárias mais pesquisas, melhores ferramentas e mais experiência com agentes móveis para que esta tecnologia finalmente se torne amplamente utilizada por um grande número de desenvolvedores.

Referências Bibliográficas

[BBK02] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A Scalable Peer-to-Peer Architect ure for Intentional Resource Discovery. *Pervasive 2002 - International Conference on Pervasive Computing*, Agosto 2002.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. In *Scientific American*, volume 284(5), pages 34–43, Maio 2001.

[BM03] Sanjit Biswas and Robert Morris. Opportunistic Routing in Multi-Hop Wireless Networks. *Proceedings of the Second Workshop on Hot Topics in Networking (HotNets-II)*, Novembro 2003.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, Agosto 1996.

[CC97] Andrew T. Campbell and Geoff Coulson. QoS Adaptive Transports: Delivering Scalable Media to the Desktop. *IEEE Network*, 11(2):18–27, Março 1997.

[CN99] Hao-hua Chu and Klara Nahrstedt. CPU Service Classes for Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (IEEE Multimedia Systems '99)*, 1999.

[DA00] Anind K. Dey and Gregory D. Abowd. The Context Toolkit: Aiding the development of context-aware applications. *Workshop on Software Engineering for Wearable and Pervasive Computing*, Junho 2000.

[DGL+98] Tony DeWitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Jaspal Subhlok, and Dean Sutherland. ReMoS: A Resource Monitoring System for Network-Aware Applications. Technical report, Carnegie Mellon University, Dezembro 1998.

[dSeS03] Francisco José da Silva e Silva. *Adaptação Dinâmica de Sistemas Distribuídos*. PhD thesis, Instituto de Matemática e Estatística - USP, Fevereiro 2003.

[dSeSEK01] Francisco José da Silva e Silva, Markus Endler, and Fabio Kon. Desenvolvendo software adaptável para computação móvel. In *Anais do Terceiro Workshop de Comunicação sem Fio (WCSF 2001)*, pages 93–101, Recife, Agosto 2001.

[dSeSEK02] Francisco José da Silva e Silva, Markus Endler, and Fabio Kon. Dynamic adaptation of distributed systems. In *12th ECOOP Workshop for PhD Students in Object-Oriented Systems*, Malaga, Spain, Junho 2002.

[FK98] Svend Frølund and Jari Koistinen. Quality of service specification in distributed object systems design. *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Abril 1998.

[GBH+00] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, , and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[GDH+01] Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetheral. Systems directions for pervasive computing. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Maio 2001.

[GDL+01] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Steven Gribble, Tom Anderson, Brian Bershad, Gaetano Borriello, and David Wetheral. Programming for pervasive computing environments. Technical report, University of Washington, Junho 2001.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice & Experience*, 16:449–459, Março 2004.

[GNY+02] Xiaohui Gu, Klara Nahrstedt, Wanghong Yuan, Duangdao Wichadakul, and Dongyan Xu. An XML-based Quality of Service Enabling Language for the Web. *Journal of Visual Language and Computing (JVLC), special issue on Multimedia Languages for the Web*, Fevereiro 2002.

[GP99] Roch Guérin and V. Peris. Quality-of-service in Packet Networks: Basic Mechanisms and Directions. *Computer Networks*, 31(3):169–179, Fevereiro 1999.

[Gut99] John V. Guttag. Communications Chameleons. In *Scientific American*, volume 281(2), page 49, Agosto 1999.

[GWS01] Krzysztof Gajos, Luke Weisman, and Howard Shrobe. Design Principles for Resource Management Systems for Intelligent Spaces. *Second International Workshop on Self-Adaptive Software (IWSAS'01)*, 2001.

[GWvB+00] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, Abril 1991.

[JN02] Jingwen Jim and Klara Nahrstedt. Classification and Comparison of QoS Specification Languages for Distributed Multimedia Applications. Technical Report UIUCDCS-R-2002-2302, University of Illinois at Urbana-Champaign, Novembro 2002.

[KCN01] Fabio Kon, Roy Campbell, and Klara Nahrstedt. Using Dynamic Configuration to Manage a Scalable Multimedia Distribution System. In *Computer Communications Journal*, volume 24, pages 105–123, 2001.

[KG99] David Kotz and Robert S. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):7–13, Julho 1999.

[KHR+00] Fabio Kon, Christopher Hess, Manuel Román, Roy H. Campbell, and M. Dennis Mickunas. A Flexible, Interoperable Framework for Active Spaces. *OOPSLA 2000 Workshop on Pervasive Computing*, Outubro 2000.

[KJ04] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley, Junho 2004.

[KLO97] Günther Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4):68–77, 1997.

[KN00] Kihun Kim and Klara Nahrstedt. A Resource Broker Model with Integrated Reservation Scheme. In *Proceedings of IEEE International Conference on Multimedia and Expo 2000 (ICME2000)*, pages 859–862, Julho 2000.

[KYH+01] Fabio Kon, Tomonori Yamane, Christopher Hess, Roy Campbell, and M. Dennis Mickunas. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, Fevereiro 2001.

[LA99] Danny B. Lange and Mitsuru Ashima. Seven Good Reasons for Mobile Agents. *Communications of the ACM*, 42(3):88–89, Março 1999.

[LCN98] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A soft real-time scheduling server on the Windows NT. *2nd USENIX Windows NT Symposium*, 1998.

[LLS+04] Christopher Lee, Neal Lesh, Candace Sidner, Louis-Philippe Morency, Ashish Kapoor, and Trevor Darrell. Nodding in Conversations with a Robot. *Proceedings of the Conference on Human Factors in Computing System (CHI)*, 2004.

[LO98] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, Agosto 1998.

[LOG99] Bruce Lowekamp, David R. O'Hallaron, and Thomas Gross. Direct Queries for Discovering Network Resource Properties in a Distributed Environment. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC99)*, Agosto 1999.

[LSZB98] Joseph P. Loyall, Richard E. Schantz, John A. Zinky, and David E. Bakken. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, Abril 1998.

[MK02a] Jeferson Roberto Marques and Fabio Kon. Gerenciamento de Recursos Distribuídos em Sistemas de Grande Escala. In *Proceedings of the 20th Brazilian Symposium on Computer Networks*, pages 800–813, Maio 2002.

[MK02b] Jeferson Roberto Marques and Fabio Kon. Gerenciamento de Recursos Distribuídos em Sistemas de Grande Escala. In *Proceedings of the 20th Brazilian Symposium on Computer Networks*, pages 800–813, Maio 2002.

[NCN99] Klara Nahrstedt, Hao-hua Chu, and Srinivas Narayan. QoS-Aware resource management for distributed multimedia applications. *Journal on High-Speed Networking*, 7:229–257, Março 1999.

[NXWL01] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-Aware Middleware for Ubiquitous Computing. In *IEEE Communications Magazine*, volume 39, Issue 11, pages 140–148, Novembro 2001.

[OMG04a] OMG. *Common Object Request Broker Architecture: Core Specification*, Março 2004. Version 3.03.

[OMG04b] OMG. *Common Object Request Broker Architecture: Core Specification*, Março 2004. Version 3.0.3.

[OMG04c] OMG. *Event Service Specification*, Outubro 2004. Version 1.2.

[RAMC+04] Anand Ranganathan, Jalal Al-Muhtadi, Shiva Chetan, Roy Campbell, and M. Dennis Mickunas. MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference (Middleware 2004)*, pages 397–416, Outubro 2004.

[RC00] Manuel Román and Roy H. Campbell. Gaia: Enabling active spaces. *9th ACM SIGOPS European Workshop*, Setembro 2000.

[RC02] Manuel Román and Roy H. Campbell. A User-Centric, Resource-Aware, Context-Sensitive, Multi-Device Application Framework for Ubiquitous Computing Environments. Technical report, University of Illinois at Urbana-Champaign, Julho 2002.

[RC03] Manuel Román and Roy H. Campbell. A Middleware-Based Application Framework for Active Space Applications. In *Proc. of ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, 2003.

[RHC+02] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, Out-Dez 2002.

[RHR+01] Manuel Román, Christopher K. Hess, Anand Ranganathan, Pradeep Madhavarapu, Bhaskar Borthakur, Prashant Viswanathan, Renato Cerqueira, Roy H. Campbell, and M. Dennis Mickunas. GaiaOS: An infrastructure for active spaces. Technical report, University of Illinois at Urbana-Champaign, Maio 2001.

[Sat96] Mahed Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), 1996.

[SBGP04] Adam Smith, Hari Balakrishnan, Michel Goraczko, and Nissanka Priyantha. Tracking Moving Devices with the Cricket Location System. *Proceedings of the 2nd USENIX/ACM MOBISYS Conference*, Junho 2004.

[SEK03] Francisco J. S. Silva, Markus Endler, and Fabio Kon. Developing Adaptive Distributed Applications: a Framework Overview and Experimental Results. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, pages 1275–1291, Novembro 2003.

[SGG04] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 7th edition, Dezembro 2004.

[SH00] Gerard J. M. Smit and Paul J. M. Havinga. Lessons learned from the design of a mobile multimedia system in the MOBY DICK project. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 85–99, Novembro 2000.

[SPP+03] Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, and Steve Ward. A Case for Goal-oriented Programming Semantics. *System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing (UbiComp '03)*, 2003.

[Sun99] Sun Microsystems, www.sun.com/jini/whitepapers. *Jini Architectural Overview*, Janeiro 1999.

[Vig98] Giovanni Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[VZL+98] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. In *Proceedings of Middleware'98*, Setembro 1998.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Outubro 2001.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American, 256(3)*, Setembro 1991.

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

[Zue99] Victor Zue. Talking with Your Computer. *Scientific American*, 281(2):40–41, Agosto 1999.